

Tong Xiao

Jingbo Zhu

# Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB

NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

<https://github.com/NiuTrans/NLPBook>

<https://niutrans.github.io/NLPBook>

Copyright © 2021-2025 Tong Xiao and Jingbo Zhu

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY  
&  
NIUTRANS RESEARCH

<https://github.com/NiuTrans/NLPBook>

<https://niutrans.github.io/NLPBook>

Licensed under the Creative Commons Attribution-NonCommercial 4.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/4.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*June 12, 2025*

---

## Preface

Natural language processing (NLP) is one of the core subfields of artificial intelligence (AI). For a long time, research in NLP primarily focused on solving specific problems in language understanding and generation, such as parsing and machine translation. This task-driven research approach dominated the development of NLP for several decades. However, with the rise of deep learning, the research paradigm of NLP has undergone a fundamental transformation. The application of deep neural networks has enabled us to tackle increasingly complex tasks. More importantly, researchers have discovered that by conducting large-scale pretraining on a base model with massive datasets, and then fine-tuning it with a small amount of task-specific data and knowledge, it is possible to construct general-purpose models capable of handling multiple tasks simultaneously. This new paradigm is greatly changing the research landscape of NLP, and even the broader field of AI.

This book focuses on modern NLP methods centered around neural networks and foundation models. It aims to provide a practical guide to understanding, building, and applying these powerful models. Unlike traditional NLP textbooks that organize chapters based on specific tasks, this book is structured from the perspective of constructing neural NLP models and is divided into three main parts:

- **Foundations of Machine Learning and Neural Networks** (Chapters 1-2): This part introduces the core concepts and methods of machine learning and neural networks, laying a foundation for the subsequent chapters. It is relatively self-contained and can be studied independently or used as background material when needed.
- **Basic Neural Models for Natural Language Processing** (Chapters 3-6): This part explains the neural networks used in NLP tasks, including word representation models, sequence models, and sequence-to-sequence models. In addition, Transformers are introduced in a dedicated chapter. These models are not limited to individual NLP tasks; rather, they serve as general-purpose tools across many applications.
- **Large Language Models** (Chapters 7-11): This part focuses on large language models (LLMs), covering topics such as pretraining, generative models, prompt engineering, alignment, and inference.

This book is intended for senior undergraduates, graduate students, researchers in related fields, and anyone interested in NLP. We strive for clear and accessible writing, aiming to introduce core concepts and fundamental methods rather than providing an in-depth exploration of all cutting-edge techniques. Therefore, this book can serve both as an introductory text for newcomers and as a reference manual for key concepts and methods in NLP.

The content of this book has gradually taken shape through our years of teaching and research experience. Initially, we only planned to write the first two parts. However, the rapid

rise and growing importance of LLMs led us to include this topic as a key part of the book. At the same time, we are delighted to witness the rapid development of NLP, and the writing of this book is also our response to this exciting trend.

Some chapters of this book have been previously published online, such as *Introduction to Transformers: An NLP Perspective* (Chapter 6) and *Foundations of Large Language Models* (Chapters 7-11), and we are grateful for the valuable feedback from many readers, which has greatly contributed to the refinement of the book. Furthermore, during the writing process, we drew significant inspiration from classic works, including *Machine Learning* by [Mitchell \[1997\]](#), *Foundations of Statistical Natural Language Processing* by [Manning and Schütze \[1999\]](#), *Pattern Recognition and Machine Learning* by [Bishop \[2006\]](#), and *Speech and Language Processing* by [Jurafsky and Martin \[2008\]](#). Many insights from these works profoundly influenced the writing approach of this book.

Lastly, we would like to express our heartfelt thanks to all those who provided suggestions and revisions to the content of this book. They are: Weiqiao Shan, Yongyu Mu, Chenglong Wang, Kaiyan Chang, Yuchun Fan, Hang Zhou, Chuanhao Lv, Xinyu Liu, Tao Zhou, Huiwen Bao, Tong Zheng, Junhao Ruan, Yingfeng Luo, Yuzhang Wu, and Yifu Huo.

Tong Xiao and Jingbo Zhu  
June, 2025

This book is dedicated to our families.



# Contents

I	Preliminaries	
<b>1</b>	<b>Foundations of Machine Learning</b>	<b>17</b>
<b>1.1</b>	<b>Math Basics</b>	<b>18</b>
1.1.1	Linear Algebra	18
1.1.2	Probability and Statistics	22
<b>1.2</b>	<b>Designing a Text Classifier</b>	<b>26</b>
1.2.1	Problem Statement	27
1.2.2	Documents as Feature Vectors	28
1.2.3	Linear Classifiers	29
1.2.4	Generative vs Discriminative	32
1.2.5	OOV Words and Smoothing	35
<b>1.3</b>	<b>General Problems</b>	<b>37</b>
1.3.1	Supervised and Unsupervised Models	37
1.3.2	Inductive Bias	38
1.3.3	Non-linearity	40
1.3.4	Training and Loss Functions	42
1.3.5	Overfitting and Underfitting	47
1.3.6	Prediction	49
<b>1.4</b>	<b>Model Selection and Evaluation</b>	<b>50</b>
1.4.1	Strategies for Model Selection	51
1.4.2	Training, Validation and Test Data	56
1.4.3	Performance Measure	57
1.4.4	Significance Tests	58
<b>1.5</b>	<b>NLP Tasks as ML Tasks</b>	<b>59</b>
1.5.1	Classification	59
1.5.2	Sequence Labeling	60
1.5.3	Language Modeling/Word Prediction	61
1.5.4	Sequence Generation	62
1.5.5	Tree Generation	63
1.5.6	Relevance Modeling	64

1.5.7	Linguistic Alignment	65
1.5.8	Extraction	67
1.5.9	Others	67
<b>1.6</b>	<b>Summary</b>	<b>68</b>
<b>2</b>	<b>Foundations of Neural Networks</b>	<b>71</b>
<b>2.1</b>	<b>Multi-layer Neural Networks</b>	<b>71</b>
2.1.1	Single-layer Perceptrons	71
2.1.2	Stacking Multiple Layers	73
2.1.3	Computation Graphs	75
<b>2.2</b>	<b>Example: Neural Language Modeling</b>	<b>78</b>
<b>2.3</b>	<b>Basic Model Architectures</b>	<b>83</b>
2.3.1	Recurrent Units	83
2.3.2	Convolutional Units	85
2.3.3	Gate Units	87
2.3.4	Normalization (Standardization) Units	88
2.3.5	Residual Units	89
<b>2.4</b>	<b>Training Neural Networks</b>	<b>90</b>
2.4.1	Gradient Descent	90
2.4.2	Batching	94
2.4.3	Parameter Initialization	96
2.4.4	Learning Rate Scheduling	97
<b>2.5</b>	<b>Regularization Methods</b>	<b>99</b>
2.5.1	Norm-based Penalties	100
2.5.2	Dropout	101
2.5.3	Early Stopping	102
2.5.4	Smoothing Output Probabilities	103
2.5.5	Training with Noise	105
<b>2.6</b>	<b>Unsupervised Methods and Auto-encoders</b>	<b>108</b>
2.6.1	Auto-encoders with Explicit Regularizers	111
2.6.2	Denosing Auto-encoders	113
2.6.3	Variational Auto-encoders	115
<b>2.7</b>	<b>Summary</b>	<b>119</b>

## II

## Basic Models

<b>3</b>	<b>Words and Word Vectors</b>	<b>123</b>
<b>3.1</b>	<b>Tokenization</b>	<b>124</b>
3.1.1	Tokenization via Rules and Heuristics	125
3.1.2	Tokenization as Language Modeling	126

3.1.3	Tokenization as Sequence Labeling	129
3.1.4	Learning Subwords	130
<b>3.2</b>	<b>Vector Representation for Words</b>	<b>137</b>
3.2.1	One-hot Representation	138
3.2.2	Distributed Representation	138
3.2.3	Compositionality and Contextuality	140
<b>3.3</b>	<b>Count-based Models</b>	<b>142</b>
3.3.1	Co-occurrence Matrices	142
3.3.2	TF-IDF	146
3.3.3	Low-Dimensional Models	147
<b>3.4</b>	<b>Inducing Word Embeddings from NLMs</b>	<b>153</b>
<b>3.5</b>	<b>Word Embedding Models</b>	<b>154</b>
3.5.1	Word2Vec	155
3.5.2	GloVe	157
3.5.3	Remarks	161
<b>3.6</b>	<b>Evaluating Word Embeddings</b>	<b>163</b>
3.6.1	Extrinsic Evaluation	163
3.6.2	Intrinsic Evaluation	164
3.6.3	Visualization	167
<b>3.7</b>	<b>Summary</b>	<b>168</b>
<b>4</b>	<b>Recurrent and Convolutional Sequence Models</b>	<b>171</b>
<b>4.1</b>	<b>Problem Statement</b>	<b>172</b>
<b>4.2</b>	<b>Recurrent Models</b>	<b>173</b>
4.2.1	An RNN-based Language Model	173
4.2.2	Training	175
4.2.3	Layer Stacking	178
4.2.4	Bi-directional Models	180
<b>4.3</b>	<b>Memory</b>	<b>181</b>
4.3.1	Memory as A System	182
4.3.2	Long Short-Term Memory	183
4.3.3	Gated Recurrent Units	185
<b>4.4</b>	<b>Convolutional Models</b>	<b>187</b>
4.4.1	Convolution	187
4.4.2	CNNs for Sequence Modeling	190
4.4.3	Handling Positional Information	193
<b>4.5</b>	<b>Examples</b>	<b>198</b>
4.5.1	Text Classification	198
4.5.2	End-to-End Speech Recognition	200
4.5.3	Sequence Labeling with LSTM and Graphical Models	203

4.5.4	Hybrid Models for Language Modeling	207
<b>4.6</b>	<b>Summary</b>	<b>207</b>
<b>5</b>	<b>Sequence-to-Sequence Models</b>	<b>211</b>
<b>5.1</b>	<b>Sequence-to-Sequence Problems</b>	<b>212</b>
<b>5.2</b>	<b>The Encoder-Decoder Architecture</b>	<b>213</b>
5.2.1	Encoding and Decoding	213
5.2.2	Example: Neural Machine Translation	215
<b>5.3</b>	<b>The Attention Mechanism</b>	<b>218</b>
5.3.1	A Basic Model	219
5.3.2	The QKV Attention	223
5.3.3	Multi-head Attention	226
5.3.4	Hierarchical Attention	229
5.3.5	Multi-layer Attention	232
5.3.6	Remarks	233
<b>5.4</b>	<b>Search</b>	<b>238</b>
5.4.1	The Length Problem	238
5.4.2	Pruning and Beam Search	242
5.4.3	Online Search	250
5.4.4	Exact Search	254
5.4.5	Differentiable Search	256
5.4.6	Hypothesis Diversity	258
5.4.7	Combining Multiple Models	260
5.4.8	More Search Objectives	262
<b>5.5</b>	<b>Summary</b>	<b>265</b>
<b>6</b>	<b>Transformers</b>	<b>269</b>
<b>6.1</b>	<b>The Basic Model</b>	<b>269</b>
6.1.1	The Transformer Architecture	269
6.1.2	Positional Encoding	273
6.1.3	Multi-head Self-attention	274
6.1.4	Layer Normalization	276
6.1.5	Feed-forward Neural Networks	277
6.1.6	Attention Models on the Decoder Side	278
6.1.7	Training and Inference	281
<b>6.2</b>	<b>Syntax-aware Models</b>	<b>283</b>
6.2.1	Syntax-aware Input and Output	284
6.2.2	Syntax-aware Attention Models	285
6.2.3	Multi-branch Models	287
6.2.4	Multi-scale Models	290
6.2.5	Transformers as Syntax Learners	291

<b>6.3</b>	<b>Improved Architectures</b> .....	<b>295</b>
6.3.1	Locally Attentive Models .....	295
6.3.2	Deep Models .....	299
6.3.3	Numerical Method-Inspired Models .....	305
6.3.4	Wide Models .....	308
<b>6.4</b>	<b>Efficient Models</b> .....	<b>312</b>
6.4.1	Sparse Attention .....	312
6.4.2	Recurrent and Memory Models .....	317
6.4.3	Low-dimensional Models .....	322
6.4.4	Parameter and Activation Sharing .....	327
6.4.5	Alternatives to Self-Attention .....	328
6.4.6	Conditional Computation .....	336
6.4.7	Model Transfer and Pruning .....	341
6.4.8	Sequence Compression .....	343
6.4.9	High Performance Computing Methods .....	344
<b>6.5</b>	<b>Applications</b> .....	<b>347</b>
6.5.1	Language Modeling .....	348
6.5.2	Text Encoding .....	349
6.5.3	Speech Translation .....	350
6.5.4	Vision Models .....	353
6.5.5	Multimodal Models .....	355
<b>6.6</b>	<b>Summary</b> .....	<b>357</b>

## III Large Language Models

<b>7</b>	<b>Pre-training</b> .....	<b>365</b>
<b>7.1</b>	<b>Pre-training NLP Models</b> .....	<b>366</b>
7.1.1	Unsupervised, Supervised and Self-supervised Pre-training .....	366
7.1.2	Adapting Pre-trained Models .....	368
<b>7.2</b>	<b>Self-supervised Pre-training Tasks</b> .....	<b>372</b>
7.2.1	Decoder-only Pre-training .....	372
7.2.2	Encoder-only Pre-training .....	373
7.2.3	Encoder-Decoder Pre-training .....	380
7.2.4	Comparison of Pre-training Tasks .....	386
<b>7.3</b>	<b>Example: BERT</b> .....	<b>388</b>
7.3.1	The Standard Model .....	388
7.3.2	More Training and Larger Models .....	393
7.3.3	More Efficient Models .....	393
7.3.4	Multi-lingual Models .....	394

<b>7.4</b>	<b>Applying BERT Models</b> .....	<b>396</b>
<b>7.5</b>	<b>Summary</b> .....	<b>401</b>
<b>8</b>	<b>Generative Models</b> .....	<b>403</b>
<b>8.1</b>	<b>A Brief Introduction to LLMs</b> .....	<b>404</b>
8.1.1	Decoder-only Transformers .....	405
8.1.2	Training LLMs .....	408
8.1.3	Fine-tuning LLMs .....	409
8.1.4	Aligning LLMs with the World .....	415
8.1.5	Prompting LLMs .....	419
<b>8.2</b>	<b>Training at Scale</b> .....	<b>425</b>
8.2.1	Data Preparation .....	425
8.2.2	Model Modifications .....	427
8.2.3	Distributed Training .....	430
8.2.4	Scaling Laws .....	433
<b>8.3</b>	<b>Long Sequence Modeling</b> .....	<b>436</b>
8.3.1	Optimization from HPC Perspectives .....	437
8.3.2	Efficient Architectures .....	438
8.3.3	Cache and Memory .....	441
8.3.4	Sharing across Heads and Layers .....	450
8.3.5	Position Extrapolation and Interpolation .....	452
8.3.6	Remarks .....	463
<b>8.4</b>	<b>Summary</b> .....	<b>466</b>
<b>9</b>	<b>Prompting</b> .....	<b>467</b>
<b>9.1</b>	<b>General Prompt Design</b> .....	<b>468</b>
9.1.1	Basics .....	468
9.1.2	In-context Learning .....	471
9.1.3	Prompt Engineering Strategies .....	473
9.1.4	More Examples .....	478
<b>9.2</b>	<b>Advanced Prompting Methods</b> .....	<b>489</b>
9.2.1	Chain of Thought .....	489
9.2.2	Problem Decomposition .....	492
9.2.3	Self-refinement .....	499
9.2.4	Ensembling .....	505
9.2.5	RAG and Tool Use .....	509
<b>9.3</b>	<b>Learning to Prompt</b> .....	<b>515</b>
9.3.1	Prompt Optimization .....	515
9.3.2	Soft Prompts .....	519
9.3.3	Prompt Length Reduction .....	528
<b>9.4</b>	<b>Summary</b> .....	<b>530</b>

<b>10</b>	<b>Alignment</b>	<b>533</b>
<b>10.1</b>	<b>An Overview of LLM Alignment</b>	<b>534</b>
<b>10.2</b>	<b>Instruction Alignment</b>	<b>535</b>
10.2.1	Supervised Fine-tuning	536
10.2.2	Fine-tuning Data Acquisition	541
10.2.3	Fine-tuning with Less Data	546
10.2.4	Instruction Generalization	547
10.2.5	Using Weak Models to Improve Strong Models	549
<b>10.3</b>	<b>Human Preference Alignment: RLHF</b>	<b>553</b>
10.3.1	Basics of Reinforcement Learning	553
10.3.2	Training Reward Models	560
10.3.3	Training LLMs	563
<b>10.4</b>	<b>Improved Human Preference Alignment</b>	<b>568</b>
10.4.1	Better Reward Modeling	568
10.4.2	Direct Preference Optimization	575
10.4.3	Automatic Preference Data Generation	578
10.4.4	Step-by-step Alignment	580
10.4.5	Inference-time Alignment	583
<b>10.5</b>	<b>Summary</b>	<b>584</b>
<b>11</b>	<b>Inference</b>	<b>587</b>
<b>11.1</b>	<b>Prefilling and Decoding</b>	<b>588</b>
11.1.1	Preliminaries	588
11.1.2	A Two-phase Framework	593
11.1.3	Decoding Algorithms	596
11.1.4	Evaluation Metrics for LLM Inference	607
<b>11.2</b>	<b>Efficient Inference Techniques</b>	<b>608</b>
11.2.1	More Caching	608
11.2.2	Batching	609
11.2.3	Parallelization	619
11.2.4	Remarks	619
<b>11.3</b>	<b>Inference-time Scaling</b>	<b>621</b>
11.3.1	Context Scaling	622
11.3.2	Search Scaling	623
11.3.3	Output Ensembling	623
11.3.4	Generating and Verifying Thinking Paths	624
<b>11.4</b>	<b>Summary</b>	<b>632</b>





# Preliminaries

<b>1</b>	<b>Foundations of Machine Learning</b> .....	<b>17</b>
1.1	Math Basics	
1.2	Designing a Text Classifier	
1.3	General Problems	
1.4	Model Selection and Evaluation	
1.5	NLP Tasks as ML Tasks	
1.6	Summary	
<b>2</b>	<b>Foundations of Neural Networks</b> .....	<b>71</b>
2.1	Multi-layer Neural Networks	
2.2	Example: Neural Language Modeling	
2.3	Basic Model Architectures	
2.4	Training Neural Networks	
2.5	Regularization Methods	
2.6	Unsupervised Methods and Auto-encoders	
2.7	Summary	



# Chapter 1

## Foundations of Machine Learning

*The goal of machine learning is to develop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future data or other outcomes of interest.*

– [Murphy \[2012\]](#)

*Machine learning can be broadly defined as computational methods using experience to improve performance or to make accurate predictions.*

– [Mohri et al. \[2018\]](#)

Data-driven NLP fits the above definitions<sup>1</sup>. It teaches computers to learn language *experience* from corpora, and to understand and utilize language based on that *experience*. Connecting machine learning (ML) with natural language processing is much more than a means that makes computers mimic human language intelligence from data. It is leading a revolution in both areas: natural language processing evolves by using a powerful tool of deriving meaning from corpora, and machine learning evolves by addressing the NLP challenges and testing on real-world data.

In this chapter, we present several basic concepts and models in machine learning. There are no tough bits but some preliminaries for the subsequent chapters. Here we focus on how to apply machine learning to NLP problems, in particular how to define an NLP problem as a statistical learning problem. To do this, we start with classification — one of the most widely-used examples in most introductory books. We then present several fundamental issues of machine learning. They are followed by a discussion on NLP problems from the machine learning perspective.

---

<sup>1</sup>We drop the term *data-driven* from now on and assume that all NLP models are data-driven in the remainder of this document.

## 1.1 Math Basics

In the remainder of this chapter and the following chapters, we will talk about machine learning problems using the tool of applied mathematics. Here are the math basics. If you find the details trivial, you can skip to Section 1.2 directly.

### 1.1.1 Linear Algebra

#### 1. Vectors and Matrices

**Scalar** may or may not be the simplest concept in linear algebra, but is surely the most common concept that one learns in high school or in university. A scalar is a number. It is a quantity that has a magnitude but has no direction. For example, height, weight, distance, temperature are all examples of scalars. Here we use an italic number to denote a scalar, for example,  $a$ ,  $b$ ,  $x$ ,  $A$ , and so on.

**Vector** and **matrix** are defined on top of scalar. A vector is an array of scalars, or simply a number list. A matrix is a rectangular array of scalars. In this book, we follow the convention of using bold letters to denote vectors and matrices. For example, an  $n$ -dimensional vector can be written as

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \quad (1.1)$$

where  $\{a_1, a_2, \dots, a_n\}$  are the elements (or entries) of the vector. Each indicates a dimension. For convenience of notation, we write  $a_i$  as  $a(i)$  sometimes. A vector is a real-valued vector only if all the elements are real numbers (i.e.,  $a_i \in \mathbb{R}$  for each  $i$ ), denoted as  $\mathbf{a} \in \mathbb{R}^n$ . Likewise, we can write an  $m \times n$  matrix as

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix} \quad (1.2)$$

where  $m$  is the number of rows and  $n$  is the number of columns.  $A_{ij}$  is the entry  $(i, j)$  of the matrix. A real-valued matrix is denoted as  $\mathbf{A} \in \mathbb{R}^{m \times n}$ . Occasionally, we use  $\mathbf{A}_{m \times n}$  to emphasize that the shape of the matrix is  $m \times n$ .

There are a few special matrices. For example, a matrix whose elements are all zeros is a **zero matrix**, denoted as  $\mathbf{0}$ . Another example is **identity matrix**, denoted as  $\mathbf{I}$ . It is a square matrix whose diagonal elements are all 1, and other elements are 0. Vectors can be treated as a special sort of matrices, too. For example, the vector in Eq. (1.1) is a matrix with only one row.

## 2. Matrix Transpose

The **transpose** of a matrix  $A_{m \times n}$  is a matrix  $B_{n \times m}$  subject to  $A_{ij} = B_{ji}$  for each pair of  $i$  and  $j$ . Often,  $\mathbf{A}$ 's transpose is denoted as  $\mathbf{A}^T$ . For example, for a matrix

$$\mathbf{A} = \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} \quad (1.3)$$

the transpose is

$$\mathbf{A}^T = \begin{bmatrix} 8 & 2 \\ 0 & 9 \\ 0 & 7 \end{bmatrix} \quad (1.4)$$

One can transpose a vector as well. For a vector

$$\mathbf{a} = [1 \ 9 \ 7 \ 3] \quad (1.5)$$

the transpose is

$$\mathbf{a}^T = \begin{bmatrix} 1 \\ 9 \\ 7 \\ 3 \end{bmatrix} \quad (1.6)$$

In general, a vector with only one row is called a **row vector** (as in Eq. (1.5)), and a vector with only one column is called a **column vector** (as in Eq. (1.6)). In this book, all vectors are row vectors by default.

## 3. Element-wise Operations on Matrices

Suppose  $\mathbf{A}$  and  $\mathbf{B}$  are two matrices of the same shape, say  $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ . The **matrix addition** of  $\mathbf{A}$  and  $\mathbf{B}$  is written as  $\mathbf{A} + \mathbf{B}$ .  $\mathbf{A} + \mathbf{B}$  is a matrix in  $\mathbb{R}^{m \times n}$  such that each element is the sum of the corresponding elements of  $\mathbf{A}$  and  $\mathbf{B}$ . Here is an example.

$$\begin{aligned} \mathbf{A} + \mathbf{B} &= \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 4 \end{bmatrix} \\ &= \begin{bmatrix} 9 & 1 & 1 \\ 3 & 9 & 11 \end{bmatrix} \end{aligned} \quad (1.7)$$

In a similar way, we can define element-wise minus ( $\mathbf{A} - \mathbf{B}$ ), product ( $\mathbf{A} \odot \mathbf{B}$ ), division ( $\mathbf{A} \oslash \mathbf{B}$ ) and other operations. A special case of element-wise product is that we multiply a matrix  $\mathbf{A}$  with another matrix whose elements are all the same (say  $k$ ). It is equal to scaling  $\mathbf{A}$  with a scalar  $k$ , denoted as  $k \times \mathbf{A}$  or  $k\mathbf{A}$ . This is also called **scalar product**. See below for an

example for  $k = 2$  and  $\mathbf{A} = \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix}$ .

$$\begin{aligned} k\mathbf{A} &= 2 \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} \\ &= \begin{bmatrix} 16 & 0 & 0 \\ 4 & 18 & 14 \end{bmatrix} \end{aligned} \quad (1.8)$$

Let  $\mathbf{A}$ ,  $\mathbf{B}$  and  $\mathbf{C}$  be matrices in  $\mathbb{R}^{m \times n}$ , and  $k$  and  $l$  be scalars in  $\mathbb{R}$ . Some properties of the matrix operations are:

- Property of the zero matrix:

$$\begin{aligned} \mathbf{A} &= \mathbf{A} + \mathbf{0} \\ &= \mathbf{0} + \mathbf{A} \end{aligned} \quad (1.9)$$

- **Commutativity:**

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \quad (1.10)$$

$$kl\mathbf{A} = lk\mathbf{A} \quad (1.11)$$

- **Associativity:**

$$(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C}) \quad (1.12)$$

$$(kl)\mathbf{A} = l(k\mathbf{A}) \quad (1.13)$$

- **Distributivity:**

$$k(\mathbf{A} + \mathbf{B}) = k\mathbf{A} + k\mathbf{B} \quad (1.14)$$

$$(k+l)\mathbf{A} = k\mathbf{A} + l\mathbf{A} \quad (1.15)$$

#### 4. Dot Product

The **dot product** of two same-sized vectors  $\mathbf{a} = [a_1 \ a_2 \ \dots \ a_n]$  and  $\mathbf{b} = [b_1 \ b_2 \ \dots \ b_n]$  is defined to be:

$$\mathbf{a} \cdot \mathbf{b} = [a_1b_1 \ a_2b_2 \ \dots \ a_nb_n] \quad (1.16)$$

In geometry, a real-valued vector  $\mathbf{a}$  can be seen as a geometric object having both magnitude (denoted as  $|\mathbf{a}|$ ) and direction. The dot product of  $\mathbf{a}$  and  $\mathbf{b}$  can also be defined as

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta) \quad (1.17)$$

where  $\theta$  is the angle between  $\mathbf{a}$  and  $\mathbf{b}$ .

## 5. Matrix Product

**Matrix product** (or **matrix-matrix product**) operates on two matrices. Given a matrix  $\mathbf{A} \in \mathbb{R}^{m \times p}$  and a matrix  $\mathbf{B} \in \mathbb{R}^{p \times n}$ , the matrix product of  $\mathbf{A}$  and  $\mathbf{B}$  produces a matrix  $\mathbf{C} \in \mathbb{R}^{m \times n}$  whose elements are defined as:

$$\begin{aligned} C_{ij} &= \sum_{k=1}^p A_{ik} \times B_{kj} \\ &= A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + \dots + A_{ip} \times B_{pj} \end{aligned} \quad (1.18)$$

Matrix product requires that the number of columns in  $\mathbf{A}$  is exactly the same as the number of rows in  $\mathbf{B}$ . In this book we use  $\mathbf{AB}$  to denote the matrix product of  $\mathbf{A}$  and  $\mathbf{B}$ . Here are a few properties of matrix product.

- **Distributivity.** For  $\mathbf{A} \in \mathbb{R}^{m \times p}$  and  $\mathbf{B}, \mathbf{C} \in \mathbb{R}^{p \times n}$ , the left distributivity is defined as

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC} \quad (1.19)$$

For  $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times p}$  and  $\mathbf{C} \in \mathbb{R}^{p \times n}$ , the right distributivity is defined as

$$(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC} \quad (1.20)$$

- **Associativity.** For  $\mathbf{A} \in \mathbb{R}^{m \times p}$ ,  $\mathbf{B} \in \mathbb{R}^{p \times q}$  and  $\mathbf{C} \in \mathbb{R}^{q \times n}$ , the associativity defines

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}) \quad (1.21)$$

- **Transpose.** For  $\mathbf{A} \in \mathbb{R}^{m \times p}$  and  $\mathbf{B} \in \mathbb{R}^{p \times n}$ , we have

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T \quad (1.22)$$

Matrix product is not commutative, i.e., we do not have  $\mathbf{AB} = \mathbf{BA}$  for all  $\mathbf{A}$  and  $\mathbf{B}$  even if  $\mathbf{A}$  and  $\mathbf{B}$  are square matrices with the same shape. Based on matrix-matrix product, we can define vector-matrix product and matrix-vector product accordingly. This is trivial because all we need is to see a vector as a matrix in multiplication.

## 6. Norm

In a vector space, **norm** is a measure of vector “length”. Given a vector  $\mathbf{a} = [a_1 \ a_2 \ \dots \ a_n] \in \mathbb{R}^n$ , the norm on  $\mathbf{a}$  is a function that maps from  $\mathbb{R}^n$  to  $\mathbb{R}$ . It is written as  $\|\cdot\|_p$ , or  $l_p$  for short.  $\|\mathbf{a}\|_p$  is defined as

$$\|\mathbf{a}\|_p = \left( \sum_{i=1}^n |a_i|^p \right)^{1/p} \quad (1.23)$$

It is called  **$p$ -norm** or  **$l_p$  norm**. The popular versions of  $p$ -norm are those when  $p = 1, 2$  and

$\infty$ :

$$\|\mathbf{a}\|_1 = \sum_{i=1}^n |a_i| \quad (1.24)$$

$$\|\mathbf{a}\|_2 = \sqrt{\sum_{i=1}^n |a_i|^2} \quad (1.25)$$

$$\|\mathbf{a}\|_\infty = \max\{|a_1|, |a_2|, \dots, |a_n|\} \quad (1.26)$$

2-norm and  $\infty$ -norm are also called **Euclidean norm** and **maximum norm**.  $p$ -norm can also be used in measuring the distance of two points in an  $n$ -dimensional space. Let  $\mathbf{b}$  be another vector in  $\mathbb{R}^n$ . The  $p$ -norm distance between  $\mathbf{a}$  and  $\mathbf{b}$  is given by the equation:

$$\|\mathbf{a} - \mathbf{b}\|_p = \left( \sum_{i=1}^n |a_i - b_i|^p \right)^{1/p} \quad (1.27)$$

## 1.1.2 Probability and Statistics

### 1. What is Probability

**Probability** is a matter of uncertainty. It is a quantity that describes how likely an event is to happen. For example, if the event is certain to happen, we will say that the probability is 1; if the event will never happen, we will say that the probability is 0.

Then, what is an event? In short, an event is an experimental outcome. It could be simply the result of everything. For example, an event could be the action that you raised your arms, the scene that you were seeing the sunset, the result that you figured out for a math quiz, and so on. A set of related events is described by a **random variable** or **variable** for short. For example, we can define the outcome of tossing a coin as a variable  $x$ . As there are two outcomes (heads or tails), we have two choices for the value of  $x$ . We can define that  $x = 1$  when the coin lands heads, and  $x = 0$  otherwise. Hence,  $x$  is a **binary variable** or more precisely a 0-1 variable. A variable choosing a value means that an event happens. For example,  $x = 1$  means the event of the coin landing heads.

In mathematics, probability is a measure on the probability space comprising events (call it a **probability measure**). As a measure, probability should satisfy certain properties, such as countable additivity [Ash and Doléans-Dade, 1999]. This means that not all functions defined on the interval  $[0, 1]$  could be a probability measure. Here we do not discuss the precise definition of probability measure. We just simply treat it as a function that outputs a real number in  $[0, 1]$ .

Usually, a probability measure is denoted as a function  $\Pr(\cdot)$ , called a **probability function**. When the input of  $\Pr(\cdot)$  is defined on a discrete set of events, the output of the function is the probability that an event happens. For example,  $\Pr(x = 1)$  means the probability of  $x$  equalling 1. Note that  $\Pr(x)$  is a function that varies its output by choosing different values of  $x$ . Suppose  $x_1$  is a value that  $x$  can take. When we write  $\Pr(x_1)$ , it means  $\Pr(x = x_1)$ . Because the probability over all events should be 1, any probability function should be subject

to:

$$\sum_{x_i \in X} \Pr(x_i) = 1 \quad (1.28)$$

where  $X$  is the set of all events. A probability function can be defined on two variables or more. Here are a few widely-used cases.

- **Joint Probability.** It is the probability that two events  $x_1$  and  $y_1$  happen at the same time, denoted as  $\Pr(x_1, y_1)$ . As a special case, the joint probability will be decomposed into the product of the probabilities of  $x_1$  and  $y_1$ , if  $x_1$  and  $y_1$  are independent of each other.

$$\Pr(x_1, y_1) = \Pr(x_1)\Pr(y_1) \quad (1.29)$$

- **Conditional Probability.** It is the probability that  $x_1$  happens in the presence of  $y_1$  happening, denoted as  $\Pr(x_1|y_1)$ .  $\Pr(x_1|y_1)$  can be defined as:

$$\Pr(x_1|y_1) = \frac{\Pr(x_1, y_1)}{\Pr(y_1)} \quad (1.30)$$

- **Marginal Probability.** It is another way to define the probability of a single variable. Given the joint probability on two variables, the marginal probability defines that

$$\Pr(x_1) = \sum_{y_j \in Y} \Pr(x_1, y_j) \quad (1.31)$$

where  $Y$  is the event space of  $y_j$ . Eq (1.31) says that  $\Pr(x_1)$  is unconditioned on  $Y$ .

Another note on joint probability. In some cases, one would like to use conditional probabilities to represent a joint probability. To this end, one can rewrite the joint probability by the **product rule** or the **chain rule**, like this

$$\Pr(x_1, y_1) = \Pr(x_1|y_1)\Pr(y_1) \quad (1.32)$$

So far, we have defined several kinds of probability on discrete variables. For continuous variables, we do not have a “probability” at a certain point. Instead, we have a density for that point. More formally, given a continuous variable  $x$ , a **probability density** of  $x$  is written as  $\Pr(x)$ . Suppose  $x \in \mathbb{R}$ . The probability of  $x$  lying in the interval  $[a, b]$  is defined via an integral:

$$\Pr(x \in [a, b]) = \int_a^b \Pr(x)dx \quad (1.33)$$

Obviously, we have

$$\int_{-\infty}^{+\infty} \text{Pr}(x) dx = 1 \quad (1.34)$$

For other properties, such as joint probability and conditional probability, the forms for continuous variables are almost the same as those for discrete variables. We just need to replace the sums in the formulas with the integrals.

## 2. Distribution and Expectation

A **probability distribution** (or **distribution** for short) is the probabilities of different values for a variable. It is defined by probability functions (for discrete variables) or probability density functions (for continuous variables). For example, a uniform distribution on a discrete variable  $x$  that chooses values from  $\{x_1, x_2\}$  can be described as  $\text{Pr}(x_i) = 1/2$  because  $\text{Pr}(x_1) = \text{Pr}(x_2)$  and  $\text{Pr}(x_1) + \text{Pr}(x_2) = 1$ ; A uniform distribution on a continuous variable in  $[-2, 2]$  can be described as  $\text{Pr}(x) = 1/4$  because  $\text{Pr}(x)$  is a constant for any  $x \in [-2, 2]$  and  $\int_{-2}^2 \text{Pr}(x) dx = 1$ . Statisticians have developed many distributions for describing the world we are living in, such as binomial distribution, Bernoulli distribution and Gaussian/normal distribution. One can find details of these distributions in most textbooks on statistics [McClave and Sincich, 2006; Freedman et al., 2007].

For describing properties of a variable, a popular means is to compute the **expected value** or **expectation** of the variable. Let  $x$  be a discrete variable that takes values from  $\{x_1, \dots, x_n\}$ , and  $\text{Pr}(x)$  be a distribution on  $x$ . The expected value of  $x$  is defined to be

$$\mathbb{E}_{x \sim \text{Pr}(x)}(x) = \sum_{i=1}^n x_i \cdot \text{Pr}(x_i) \quad (1.35)$$

where the subscript  $x \sim \text{Pr}(x)$  indicates that  $x$  follows the distribution  $\text{Pr}(x)$ . In many cases, we can drop the subscript and rewrite it as  $\mathbb{E}(x)$ .  $\mathbb{E}(x)$  is essentially the weighted average value of  $x$  under the distribution  $\text{Pr}(x)$ . It is a measure of central tendency, and is sometimes called the **mean** of a variable (denoted as  $\mu$ ).

Then, we can define the **variance** of a variable as the squared variation of the variable from the mean value, like this

$$\text{Var}(x) = \mathbb{E}[(x - \mathbb{E}(x))^2] \quad (1.36)$$

Informally, it describes how far the values are from the mean.  $\text{Var}(x)$  is usually written as  $\sigma^2$ , where  $\sigma$  is called **standard deviation**.

For a continuous variable  $x \in \mathbb{R}$ , the expected value is defined as:

$$\mathbb{E}(x) = \int_{-\infty}^{+\infty} x \cdot \text{Pr}(x) dx \quad (1.37)$$

where  $\text{Pr}(x)$  is a probability density function. For computing the variance of  $x$ , we just reuse

Eq. (1.36).

### 3. Entropy

Entropy is one of the most important tools of describing random variables and processes [Shannon, 1948b]. It is a measure of expected surprise. The more deterministic the events occur, the less surprise and less information there will be. For simplicity, we restrict the discussion on discrete variables here<sup>2</sup>. Let  $x$  be a variable and  $\Pr(x)$  be a distribution on  $x$ . The entropy is written as:

$$H(x) = - \sum_{i=1}^n \Pr(x_i) \cdot \log_b \Pr(x_i) \quad (1.38)$$

where  $b$  is the base of the logarithm function. The value of  $b$  is typically set to 2, 10 and  $e$ .

In addition to obtaining the entropy of a single distribution, we can determine the similarity of two distributions from the entropy point of view. Suppose  $p(x)$  and  $q(x)$  are distributions on  $x$ . The **relative entropy** of  $p$  with respect to  $q$  is defined to be:

$$D_b(p||q) = \sum_{i=1}^n p(x_i) \cdot \log_b \frac{p(x_i)}{q(x_i)} \quad (1.39)$$

We can treat  $p(x_i)$  as a weight to the log likelihood ratio  $\log_b \frac{p(x_i)}{q(x_i)}$ . Hence,  $D_b(p||q)$  is a weighted sum of the likelihood ratios over all possible values. A smaller value  $D_b(p||q)$  indicates that distributions  $p$  and  $q$  are closer. For example,  $p$  and  $q$  will be identical if  $D_b(p||q) = 0$ . The relative entropy is also called the **Kullback-Leibler (KL) divergence**. Note that the relative entropy is asymmetric, i.e., we cannot guarantee  $D_b(p||q) = D_b(q||p)$ .

Another concept that is popular in machine learning is **cross-entropy**. It is a measure of the information (in terms of the total number of bits) that we need to transit the events from a source in one distribution with another distribution. More formally, we write the cross-entropy of the distribution  $p$  with the distribution  $q$  as  $H_{\text{cross}}(p, q)$ . It is defined to be:

$$H_{\text{cross}}(p, q) = \sum_{i=1}^n p(x_i) \cdot \log_2 q(x_i) \quad (1.40)$$

Like relative entropy, cross-entropy is asymmetric too. Both relative entropy and cross-entropy are widely used in designing the objective of learning NLP systems although they are different quantities. The difference lies in that relative entropy calculates the average number of bits when replacing  $p$  with  $q$ , while cross-entropy calculates the total number of bits in the same process.

---

<sup>2</sup>For continuous variables, we have similar calculations.

## 1.2 Designing a Text Classifier

**Classification** is one of the most common problems in machine learning. It aims at automatically categorizing something into a set of classes. These classes are called **labels**, or **tags**, or **categories**. In general, a program of classification is called a **classifier** or **classification system**. There are a vast number of practical applications of classifiers. A simple example is spam filtering in that one needs to label an email as “spam” or “not-spam”. More challenging examples include classifying computed tomography images of organs into “normal” or “not-normal”, determining whether a piece of Chinese text is written by native speakers or not, labeling a patent application with a set of IPC codes it belongs to, and so on.

Many machine learning theories and algorithms are modeled and tested on classification tasks. Following this convention, we consider text classification as an example to get started. Assume that we have a corpus like this.

Text	Label
The game was wonderful.	Not-food
I’ve tried my best to recreate it in my kitchen. It tastes heavenly.	Food
For centuries seaweed was considered a food for normal people.	Food
Have you finished your coding work today?	Not-Food
I was wondering how you could miss the bus.	Not-Food
I like fruit because it is good for health.	Food
Natural language processing research is amazing.	Not-Food
...	...

Each line of the corpus is a tuple of a piece of text (we simply call it a document) and a label that indicates whether the text is about food or not. We call such tuples **samples**, or more precisely **labeled samples**. Labeled samples are essentially question-answer pairs although they do not strictly follow the general forms of questions and answers. For example, in the samples presented above, one can take a document as a question and take its label as the answer. In the next few chapters, we will show that such a form of describing machine learning problems is general and fits most problems in NLP.

Next, let us assume that we have a classifier that learns from those samples the way of labeling documents. The classifier is then used to label every new document as “Food” or “Not-Food”. For example, for the text

Fruit is not my favorite but I can enjoy it.

the classifier would categorize it as “Food”.

However, text classification, though seems simple on the surface, is much more than classifying or sorting unlabeled samples into classes. It presents a wide variety of issues, especially when considering the ambiguities and richness of language. Modern classifiers are not a system comprising a set of hand-crafted rules. They instead model the classification problem in a probabilistic manner, making it possible to learn the ability of classification from large-scale labeled data.

### 1.2.1 Problem Statement

Let  $x$  be a document and  $c$  be a label. Here we assume a probabilistic classifier which would estimate how likely we choose  $c$  as the label of  $x$ , denoted as  $\Pr(c|x)$ .  $\Pr(c|x)$  is in general a **classification model**. It describes a distribution over the set of all possible labels, satisfying

$$\sum_{c \in C} \Pr(c|x) = 1 \quad (1.41)$$

where  $C$  is the label set. For any document, we choose the most probable label as output via the classification model, like this

$$\hat{c} = \arg \max_{c \in C} \Pr(c|x) \quad (1.42)$$

where  $\hat{c}$  is the “best” label predicted by the model.  $\arg \max$  is the abbreviation of the arguments of the maxima. It returns the value of the argument that maximizes some function.

Eq. (1.42) is the fundamental equation of classification. It implies three problems

- **The modeling problem.**  $\Pr(c|x)$  is a computational challenge because it is not obvious how to obtain the value of  $\Pr(c|x)$  for each pair of  $x$  and  $c$ . To make an adequate model, one may need to represent  $x$  and  $c$  in some way that is easy to use, and may need to develop some mathematical form connecting  $x$  and  $c$  together with the algorithms necessary to compute the form.
- **The learning problem.** From a statistical learning point of view, the general form of  $\Pr(c|x)$  represents a range of models configured with different variables or parameters. These models are essentially of the same form but would behave differently if we choose different values of those parameters. Thus, we need to choose a “good” model among them. This is typically addressed by optimizing the parameters on labeled data by some criterion.
- **The prediction problem.** We are addressing a **binary classification** problem here. Predicting document class is thus trivial as we just need to determine which class is more probable than the other. However, one can hardly imagine how difficult the prediction problem is in the real world, especially when predicting tree or graph-like structures and other non-linear structures<sup>3</sup>. For many NLP problems, prediction needs effective and efficient search algorithms.

These problems are general and cover many machine learning and natural language processing tasks. Binary classification, though is one of the simplest cases, can fully complete the goal of getting familiar with machine learning. On the other hand, classification has several variants. Here are two examples.

- **Multi-class classification.** It is an updated version of binary classification. In multi-class

---

<sup>3</sup>Predicting trees or other structures is not recognized as a standard sub-problem of classification. It is typically referred to as **structure prediction**. We will show in the later sections that both classification and structure prediction can share a similar machine learning paradigm.

classification, one needs to classify samples into one of three or more classes.

- **Multi-label classification.** This might be confusing because multi-class classification and multi-label classification seem to be the same thing. By conventional use of the terms, multi-label classification is referred to as assigning multiple labels to a sample. By contrast, the problem presented in Eq. (1.42) is a **single-label classification** problem.

Classification would be more interesting and challenging if we extend it to the case of dealing with hierarchical data. For example, for biological and patent data, some classes can be grouped into a super-class. This makes a hierarchy of the classes and requires a hierarchical classification schema.

## 1.2.2 Documents as Feature Vectors

The first problem we confront in designing text classification models is how to represent a document. Treating  $x$  as a string is simply not a good solution. One may want a representation by which a human being can understand the text. For example, we can parse each sentence in a document into a syntax tree and use trees as a text representation. This, however, requires efforts for developing additional NLP tools (such as syntactic parsers).

Representation, of course, is a fundamental issue in NLP. We skip here those diverse, state-of-the-art models, but present a simple and effective model — **the bag-of-words (BOW) model**. The bag-of-words model is a feature-based model of representing documents. In machine learning, a **feature** is a property of a sample. One can define a feature not only as some concrete attribute, such as a name and a gender, but also as a quantity that is countable for machine learning systems, such as a real number.

In the bag-of-words model, a feature corresponds to the occurrence times of a word. Let  $V$  be a vocabulary. A document can be represented as a  $|V|$ -dimensional feature vector. Each dimension describes a word count feature. It counts the occurrence of the  $i$ -th word of  $V$  in the document. More formally, let  $\mathbf{x}$  be a feature vector. The  $i$ -th entry of  $\mathbf{x}$  is defined as:

$$x_i = \text{count}(V_i) \quad (1.43)$$

where  $\text{count}(\cdot)$  is a counting function. Consider, for example, the following lines of text<sup>4</sup>.

As I went to Bonner  
I met a pig  
Without a wig,  
Upon my word and honor.

---

<sup>4</sup>The text is from *Mother Goose rhymes*.

We then have a vocabulary extracted from the corpus<sup>5</sup>, like this

$$V = \{ \text{"a"}, \text{"and"}, \text{"as"}, \text{"Bonner"}, \text{"honor"}, \\ \text{"I"}, \text{"met"}, \text{"word"}, \text{"my"}, \text{"pig"}, \\ \text{"to"}, \text{"upon"}, \text{"went"}, \text{"wig"}, \text{"without"}, \\ \text{","}, \text{"."} \}$$

Each line of the text can be seen as a document and represented as a feature vector. See below for the feature vectors generated by using the bag-of-words model.

	a	and	as	Bonner	honor	I	met	word	my	pig	to	upon	went	wig	without	,	.
As I went to Bonner	[ 0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0 ]
I met a pig	[ 1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0 ]
Without a wig,	[ 1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0 ]
Upon my word and honor.	[ 0	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1 ]

The bag-of-words model defines a vector space<sup>6</sup>. In this space, the similarity of two vectors is measured in some way like dot-product. It helps when one wants to establish the relationship between documents — two documents with more overlapping words are more similar. This intuitive picture has guided many people when building classification systems.

The beauty of the bag-of-words model comes from its simple form in that any word is independent of other words. The independent assumption makes it possible to encode a document with almost infinite word relations as a countable, feasibly sized vector. On the other hand, representing a document as a feature vector of word counts is not the only option. As an improvement, one might take more context into account, and/or design more powerful features. This leads to an active line of research on text representation methods, ranging from heuristics-based methods to representation learning methods. We will see a few of them in the subsequent chapters.

### 1.2.3 Linear Classifiers

**Linear classifier** is one of the simplest classification models. Suppose we take a feature vector  $\mathbf{x} = [x_1 \dots x_n]$  as input, and take a weight vector  $\mathbf{w} = [w_1 \dots w_n]$  and a scalar  $b$  as parameters. A linear function has a form like this

$$\begin{aligned} s(\mathbf{x}, \mathbf{w}, b) &= \mathbf{w} \cdot \mathbf{x} + b \\ &= w_1x_1 + w_2x_2 + \dots + w_nx_n + b \end{aligned} \quad (1.44)$$

where  $b$  is a bias term. The dot product  $\mathbf{x} \cdot \mathbf{w}$  is a linear combination of  $[x_1 \dots x_n]$  where each  $x_i$  is weighted by  $w_i$ . For a more condensed formulation, we can define a new input vector

<sup>5</sup>We removed the case of the word at the beginning of each line.

<sup>6</sup>A vector space should be closed under vector addition and scalar multiplication.

$\mathbf{x}' = [x_1 \ \dots \ x_n \ 1]$  and a new weight vector  $\mathbf{w}' = [w_1 \ \dots \ w_n \ b]$ . We then rewrite Eq. (1.44) as:

$$\begin{aligned} s(\mathbf{x}', \mathbf{w}') &= s(\mathbf{x}, \mathbf{w}, b) \\ &= \mathbf{w}' \cdot \mathbf{x}' \end{aligned} \quad (1.45)$$

In the following, we drop the bias term  $b$  for simplicity and use  $s(\mathbf{x}, \mathbf{w})$  to denote a linear function. For classification, a linear function is used to describe class membership. Each class is assigned a score by the function equipped with a unique weight vector. Consider again the binary classification as an example. Let  $c_a$  and  $c_b$  be two classes. We can define two weight vectors  $\mathbf{w}_a$  and  $\mathbf{w}_b$  so that the function can discriminate between  $c_a$  and  $c_b$ .

For prediction, we can infer a class based on  $s(\mathbf{x}, \mathbf{w})$ . To achieve this, activation functions  $\psi(\cdot)$  are in general used for mapping the value of  $s(\mathbf{x}, \mathbf{w})$  to a class. For example, for binary classification, we can define an activation function like this,

$$\psi(x) = \begin{cases} c_a & x > 0 \\ c_b & \text{otherwise} \end{cases} \quad (1.46)$$

Then, we make a prediction by

$$\psi(s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b)) = \begin{cases} c_a & s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b) > 0 \\ c_b & \text{otherwise} \end{cases} \quad (1.47)$$

As  $s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b) = s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b)$ , the final prediction function is  $\psi(s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b))$ . We call it a **discriminant function**. Note that  $s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b)$  is linear. So this is a **linear discriminant function**.

A discriminant function assigns an input vector  $\mathbf{x}$  directly to a class but it does not describe how likely a class would appear given  $\mathbf{x}$ . There are other activation functions for generating a desirable output. For example, we may want a probability-like output (see Eq. (1.42)), and thus define  $\psi(\cdot)$  as a normalized function<sup>7</sup>. Then, the classification probabilities are given by the equation

$$\begin{aligned} \left[ \Pr(c_a|\mathbf{x}) \ \Pr(c_b|\mathbf{x}) \right] &= \psi \left( \left[ s(\mathbf{x}, \mathbf{w}_a) \ s(\mathbf{x}, \mathbf{w}_b) \right] \right) \\ &= \left[ \frac{s(\mathbf{x}, \mathbf{w}_a)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)} \ \frac{s(\mathbf{x}, \mathbf{w}_b)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)} \right] \end{aligned} \quad (1.48)$$

where  $\psi(\cdot)$  is a **vector function**<sup>8</sup>. It normalizes the entries of the input vector by the sum of these entries. The decision rule is simple: we predict  $c_a$  if  $\Pr(c_a|\mathbf{x}) > \Pr(c_b|\mathbf{x})$ , and  $c_b$  otherwise. Since  $\frac{s(\mathbf{x}, \mathbf{w}_a)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)}$  and  $\frac{s(\mathbf{x}, \mathbf{w}_b)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)}$  share the same denominator, the prediction can also be made by comparing the numerators, i.e., we are doing the same thing as that in Eq. (1.47). In subsequent chapters, we will show that the trick of transforming

<sup>7</sup>A normalized function is a function whose integral over its domain is equal to 1.

<sup>8</sup>A vector function reads a vector and returns a new vector.

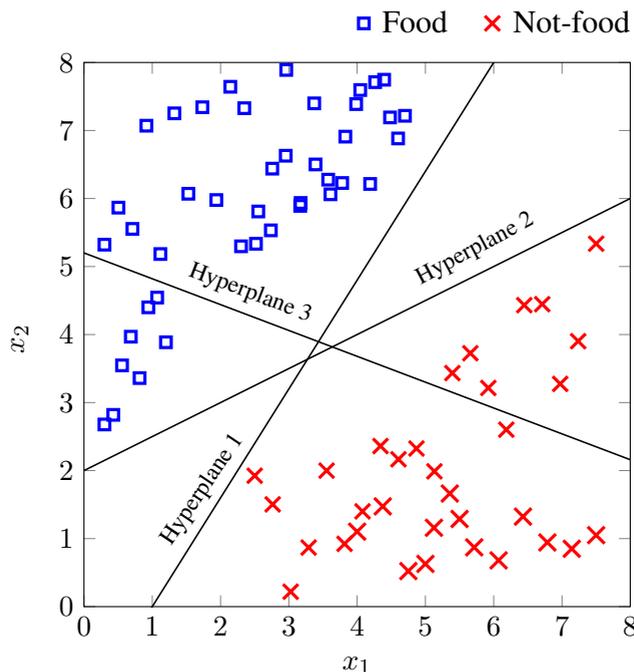


Figure 1.1: Data points and separating hyperplanes in two dimensions. There are two classes of data points (food and non-food). Both hyperplanes 1 and 2 separate the space into two sub-spaces where the two classes of data points are isolated. In this sense, the problem here is linearly separable. On the other hand, hyperplane 3 fails to separate the two classes, that is, the data points in the same class are classified into two different classes.

comparing probabilities to comparing real-valued scores is frequently used for addressing NLP problems.

For ease of understanding, one can see a linear classification model as a hyperplane (or **decision surface**, or **decision boundary**) that separates data points into different groups. Figure 1.1 shows example hyperplanes in a 2D space where each  $\mathbf{x}$  is a data point. Hyperplanes 1 and 2 successfully separate the data points into the correct classes, while hyperplane 3 fails to do so. In this sense, the task of classification is to find hyperplanes that make the correct separation of data points.

«««« HEAD It is worthy of note that linearity is the basis of many classifiers although most of them do not in the same form as Eqs. (1.44 - 1.45). A linear model can be nearly a perfect solution if the problem is linearly separable<sup>9</sup>. Even for non-linearly separable problems, linear models can be married to other models with a non-linear separation ability. A good example is that one can achieve non-linear classification by marrying a linear model with a non-linear activation function. ===== It is worthy of note that linearity is the basis of many classifiers although most of them do not exist in the same form as Eqs. (1.44 - 1.45). A linear model

<sup>9</sup>Linear separability checks if there is a way that we put a hyperplane to reside a group of data points from the remaining data points. For example, the problem shown in Figure 1.1 is linearly separable.

can be nearly a perfect solution if the problem is linearly separable<sup>10</sup>. Even for non-linearly separable problems, linear models can be married to other models with a non-linear separation ability. A good example is that one can achieve non-linear classification by marrying a linear model with a non-linear activation function. »»»» origin/master

## 1.2.4 Generative vs Discriminative

There are two ways, though not restricted to linear models, to make use of linearity in classification — **generative models** and **discriminative models**. While most statistical classifiers are of the modeling variety, they choose the backbone design from either or both of these two types of models.

### 1. Generative Models

A goal of classification is to learn  $\Pr(c|\mathbf{x})$ . Generative models do not explicitly model this conditional probability. Instead, they model the joint probability  $\Pr(\mathbf{x}, c)$ , and use the Bayes' rule to compute  $\Pr(c|\mathbf{x})$ . This is given by the following equation.

$$\begin{aligned}\Pr(c|\mathbf{x}) &= \frac{\Pr(\mathbf{x}, c)}{\Pr(\mathbf{x})} \\ &= \frac{\Pr(c)\Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})}\end{aligned}\tag{1.49}$$

where  $\Pr(\mathbf{x}, c)$  is rewritten as  $\Pr(\mathbf{x}|c)\Pr(c)$ . For an optimal class  $\hat{c}$ , we choose a class  $c$  by maximizing  $\Pr(c|\mathbf{x})$  (see Eq.(1.42)). Since the denominator  $\mathbf{x}$  is a constant for any  $c$ , we just need to maximize the numerator. Then, we rewrite Eq.(1.42) in the form

$$\begin{aligned}\hat{c} &= \arg \max_{c \in \mathcal{C}} \Pr(c|\mathbf{x}) \\ &= \arg \max_{c \in \mathcal{C}} \frac{\Pr(c)\Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})} \\ &= \arg \max_{c \in \mathcal{C}} \Pr(c)\Pr(\mathbf{x}|c)\end{aligned}\tag{1.50}$$

where  $\Pr(c)$  is the prior of  $c$ , and  $\Pr(\mathbf{x}|c)$  is the conditional probability of the input document vector  $\mathbf{x}$  given  $c$ . Computing  $\Pr(c)$  is easy. For example, the **maximum likelihood estimation (MLE)** defines  $\Pr(c)$  as a relative frequency:

$$\Pr(c) = \frac{\text{count}(c)}{\sum_{c' \in \mathcal{C}} \text{count}(c')}\tag{1.51}$$

where  $\text{count}(c)$  counts the occurrences of  $c$  in a corpus.

But computing  $\Pr(\mathbf{x}|c)$  is non-trivial as data sparseness prevents us from accurately estimating the probability of a high-dimensional document vector. Recall that the bag-of-words model defines  $x_i$  as the word frequency of  $V_i$  in the document. We assume here that the feature

<sup>10</sup>Linear separability checks if there is a way that we put a hyperplane to separate a group of data points from the remaining data points. For example, the problem shown in Figure 1.1 is linearly separable.

vector  $\mathbf{x} = [x_1 \ \dots \ x_n]$  is generated by a multinomial  $(p_1(c), p_2(c), \dots, p_n(c))$ , where  $p_i(c)$  is the probability of  $V_i$  occurring given  $c$ . Based on MLE, we can estimate  $p_i(c)$  by the relative frequency estimation:

$$p_i(c) = \frac{\text{count}(V_i, c)}{\sum_{1 \leq i' \leq n} \text{count}(V_{i'}, c)} \quad (1.52)$$

where  $\text{count}(V_i, c)$  is the number of occurrences of  $V_i$  in all documents labeled as  $c$ . Then,  $\Pr(\mathbf{x}|c)$  is given by

$$\Pr(\mathbf{x}|c) = \frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \cdot \prod_{i=1}^n p_i(c)^{x_i} \quad (1.53)$$

Substituting Eq. (1.53) into Eq. (1.50), we have

$$\hat{c} = \underset{c \in C}{\text{argmax}} \Pr(c) \cdot \frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \cdot \prod_{i=1}^n p_i(c)^{x_i} \quad (1.54)$$

Note that  $\frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!}$  is independent of any  $c$ . We drop it in  $\text{argmax}$ , and rewrite the right-hand side of the equation in log scale:

$$\hat{c} = \underset{c \in C}{\text{argmax}} \log(\Pr(c)) + \sum_{i=1}^n x_i \cdot \log(p_i(c)) \quad (1.55)$$

Obviously, this is a linear model. It defines the feature vector and weight vector as below

$$\mathbf{x} = \begin{bmatrix} 1 & x_1 & \dots & x_n \end{bmatrix} \quad (1.56)$$

$$\mathbf{w} = \begin{bmatrix} \log(\Pr(c)) & \log(p_1(c)) & \dots & \log(p_n(c)) \end{bmatrix} \quad (1.57)$$

Such a form is sometimes called a **log-linear** model, as the linearity comes from transforming the original problem via a logistic function.

In general, Eq (1.55) is called the **multinomial naive Bayes** approach. There are, of course, the naive Bayes variants for other types of feature vectors. For example, for binary value feature vectors, one can assume a Bernoulli distribution on each entry of a vector and design a **Bernoulli naive Bayes** classifier; for vectors with continuous features, one can assume a Gaussian distribution over continuous data and design a **Gaussian naive Bayes** classifier.

## 2. Discriminative Models

The model defined by  $\Pr(c|\mathbf{x}) = \frac{\Pr(\mathbf{x}, c)}{\Pr(\mathbf{x})} = \frac{\Pr(c) \Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})}$  (see Eq. (1.49)) is called generative because it assumes some way of generating data  $\mathbf{x}$  given label  $c$ . The idea is to use  $\Pr(\mathbf{x}, c)$  as a pivot to compute  $\Pr(c|\mathbf{x})$ . As an alternative possibility for modeling  $\Pr(c|\mathbf{x})$ , **discriminative models** do not try to model the distribution of  $\mathbf{x}$  but estimate  $\Pr(c|\mathbf{x})$  directly. An example is **logistic regression**. For binary text classification, a naive Bayes classifier predicts label  $c_a$  for

a given document  $\mathbf{x}$  only if the following function is positive:

$$f_a(\mathbf{x}) = \log \frac{\Pr(c_a|\mathbf{x})}{\Pr(c_b|\mathbf{x})} \quad (1.58)$$

One can assume that this quantity follows a linear model:

$$\log \frac{\Pr(c_a|\mathbf{x})}{\Pr(c_b|\mathbf{x})} = \mathbf{w} \cdot \mathbf{x} \quad (1.59)$$

Since  $\Pr(c_b|\mathbf{x}) = 1 - \Pr(c_a|\mathbf{x})$ , we have

$$\Pr(c_a|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x})} \quad (1.60)$$

This is a logistic function, or more precisely a Sigmoid function. With such a model, we predict  $c_a$  only if  $\mathbf{w} \cdot \mathbf{x}$  is positive. Eq. (1.60) is also called the logistic regression classifier. It is simply a discriminative analog of the naive Bayes classifier.

An advantage of discriminative models is that they offer flexibility in viewing classification (or other machine learning problems) from different angles. While generative models try to estimate the data distribution of  $\mathbf{x}$ , discriminative models try to find a good boundary between classes. Discriminative models, therefore, care more about which class is prioritized over another given  $\mathbf{x}$ , or in possibility language which class is more likely to appear, instead of making assumptions on individual data points. This makes it possible to learn a classifier by minimizing the number of some errors, not necessarily guaranteeing the maximum likelihood on those data points. This approach is generally called **error-driven learning**.

There are many ways to define errors. Like generative models, one can learn a discriminative model by fitting parameters  $\mathbf{w}$  to maximize the likelihood on the training data. Let  $\{(\mathbf{x}^{(1)}, c^{(1)}), \dots, (\mathbf{x}^{(K)}, c^{(K)})\}$  be a set of labeled documents, where  $\mathbf{x}^{(k)}$  is a document and  $c^{(k)}$  is the corresponding class. The best parameter vector  $\hat{\mathbf{w}}$  is given by the equation

$$\hat{\mathbf{w}} = \arg \max_{\mathbf{w}} \sum_{k=1}^K \log \Pr(c^{(k)}|\mathbf{x}^{(k)}) \quad (1.61)$$

Taking Eq. (1.60), the process can be seen as maximizing the likelihood

$$\begin{aligned} \hat{\mathbf{w}} &= \arg \max_{\mathbf{w}} \sum_{k=1}^K \log \delta(c_a, c^{(k)}) \Pr(c_a|\mathbf{x}^{(k)}) + \log \delta(c_b, c^{(k)}) \Pr(c_b|\mathbf{x}^{(k)}) \\ &= \arg \max_{\mathbf{w}} \sum_{k=1}^K \log \delta(c_a, c^{(k)}) \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x}^{(k)})} + \\ &\quad \log \delta(c_b, c^{(k)}) \left(1 - \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x}^{(k)})}\right) \end{aligned} \quad (1.62)$$

where  $\delta(\cdot, \cdot)$  is an **indicator function** that returns 1 if the two arguments are equal, and 0

otherwise.

Alternatively, we can train the model by minimizing 0-1 errors, that is, we count an error when the output is not the correct label. This process can be formulated as

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \sum_{k=1}^K \delta(\hat{c}^{(k)}, c^{(k)}) \quad (1.63)$$

where  $\hat{c}^{(i)}$  is the prediction made by the classifier. The 0-1 error can be further extended by taking the posterior into account:

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \sum_{k=1}^K \Pr(\hat{c}^{(k)} | \mathbf{x}^{(k)}) \cdot \delta(\hat{c}^{(k)}, c^{(k)}) \quad (1.64)$$

The training objective plays an important role in discriminative models. This topic, however, is so broad and beyond the scope of this section. We will present some in Section 1.3.4. As another bonus, discriminative models do not restrict features to forming a probabilistic generative story. In a broader sense,  $\mathbf{x}$  could be any feature vector that is designed by researchers and engineers. One does not even need to guarantee the probabilistic meaning for these features. For example, let  $g(\mathbf{x})$  be the output of another system, say some scores. Following Eq. (1.60), a new binary classification model can be designed in a logistic regression manner:

$$\Pr(c_a | \mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot g(\mathbf{x}))} \quad (1.65)$$

For learning  $g(\mathbf{x})$ , one can either pre-train it on some additional data, or train it jointly with  $\mathbf{w}$  (i.e., the parameters of the upper-level model  $\Pr(c_a | \mathbf{x})$ ).

### 1.2.5 OOV Words and Smoothing

The **out-of-vocabulary (OOV)** problem occurs when some of the words of a document are not found in the vocabulary. OOV words are common in NLP because new words are always there no matter how much text we have seen. Figure 1.2 gives two curves to illustrate this problem. As shown in Figure 1.2 (a), new words continuously appear when more data is available. When we fix the data that is used for testing the coverage of a vocabulary, OOV words remain even if we have an extremely large vocabulary (Figure 1.2 (b)).

For practical systems, OOV words are common because the vocabulary is often restricted to a “small” number of entries. A standard method is to keep the top- $n$  most frequent words and discard the rest. In this case, OOV words are treated as *unknown* words. For example, a new symbol  $\langle \text{unk} \rangle$  is introduced into the vocabulary so that all OOV words are denoted as  $\langle \text{unk} \rangle$ . A more aggressive idea is to build an **open-vocabulary** system that accepts every possible word, but it would require more sophisticated algorithms and probably a task-specific design of data structures. The  $\langle \text{unk} \rangle$  trick is still the de facto standard for the development of current NLP systems.

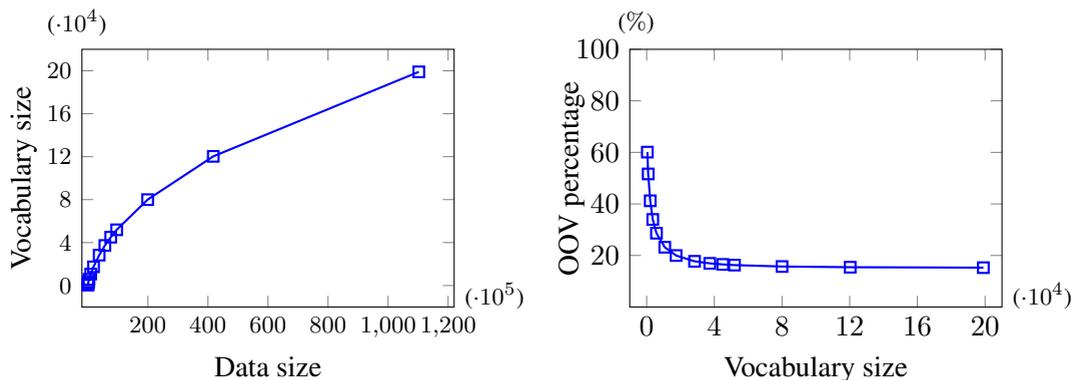


Figure 1.2: Data size (in number of words), vocabulary size and percentage of OOV. The statistics are collected on the English data provided in the WMT21 Zh-En translation task. The more data we use, the larger vocabulary we have. The increase in vocabulary size continues even if we build the vocabulary on data of more than 100 million words. However, the slope of the curve tends to be smaller as more data is involved. Interestingly, the OOV percentage converges to a certain level as the vocabulary size increases, indicating that new words always occur no matter how many words we have observed.

For words that are already in the vocabulary, there are also **unseen words** that are absent in the parameter estimation phase but appear in a new document. A naive implementation of the model described in the previous section would be tough when dealing with unseen words. For example, Eq. (1.52) will simply give a zero probability if the word  $V_i$  does not occur in any training example labeled with  $c$ . In consequence, for a new example containing  $V_i$ , Eq. (1.53) will assign it a zero probability. This result is obviously unreasonable. However, we should not simply attribute it to the model design itself. Instead, a primary reason for this is the insufficient data used for parameter estimation. This is also explained by **Zipf's Law**: a small number of words occur quite often, while a large number of words occur rarely.

However, we cannot suppose that we always have access to some data where every word occurs sufficiently. Alternatively, one could adopt smoothing techniques to redistribute the probability over the vocabulary. Consider Eq. (1.52) as an example. We can add a small number to each  $V_i$ , and rewrite the equation as:

$$p_i(c) = \frac{\text{count}(V_i, c) + \alpha}{\sum_{1 \leq i' \leq n} \text{count}(V_{i'}, c) + n \cdot \alpha} \quad (1.66)$$

where  $\alpha$  is the default count that we assign to each word. In this way,  $p_i(c)$  gives a non-zero probability even if  $\text{count}(V_i, c) = 0$ . Eq. (1.66) is doing something like subtracting word counts from high-frequency words and reassigning the subtracted counts to low-frequency words. This method is called **additive smoothing** or **add- $\alpha$  smoothing**. It is one of the simplest smoothing methods. For other methods, we refer the reader to language modeling papers where smoothing is in heavy use [Chen and Goodman, 1999].

## 1.3 General Problems

Building a simple text classifier is a good start but not enough for solving complicated, diverse real-world problems. For a more general picture of how modern machine learning systems work, we now discuss some problems that are important when designing such systems.

### 1.3.1 Supervised and Unsupervised Models

**Supervised learning** deals with labeled samples, by which we mean that an input  $x$  is associated with an output  $y$ . Given a set of input-output pairs  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(K)}, y^{(K)})\}$ , the task here is to learn a function  $f(\cdot)$  that maps each  $x^{(k)}$  to  $y^{(k)}$ :

$$y^{(k)} = f(x^{(k)}) \quad (1.67)$$

This process is called supervised because learning  $f(\cdot)$  is guided by the manually annotated answer  $y^{(k)}$  for  $x^{(k)}$ . In general,  $y^{(k)}$  is called the ground-truth or gold-standard label of  $x^{(k)}$ . After that, when a new input  $x_{\text{new}}$  comes, we use the learned function  $f(\cdot)$  to predict the output. We will say that the supervised learning succeeds if the prediction  $f(x_{\text{new}})$  is the same as the ground-truth  $y_{\text{new}}$ .

The vast majority of NLP can be framed as supervised learning problems. Assigning a class to a document is no doubt one of the simplest cases. Other NLP tasks include but are not limited to producing a sequence of labels, a piece of text, a syntax tree, and a graph.

In contrast to supervised learning, **unsupervised learning** deals with unlabeled samples, in other words, for each sample, we have an input  $x$  in the absence of the correct output  $y$ . In this case, we need an algorithm that learns from the unlabeled data  $\{x^{(k)}\}$  the mapping function from  $x^{(k)}$  to some output. Since there is no human intervention on the output of the function, algorithms of this kind need to discover patterns in  $\{x^{(k)}\}$  and optimize the way we represent  $\{x^{(k)}\}$  and function outputs by some criteria. These criteria are typically inspired by human prior knowledge so that the resulting function could output something that we expect.

A common example in unsupervised learning is **word clustering**. It groups a set of words by assigning similar words into the same cluster<sup>11</sup>. Based on such a criterion, we can bias  $f(\cdot)$  towards outputting the same cluster for similar words. A more difficult case is **unsupervised bilingual dictionary induction**. It learns a word-level mapping between two languages without the need of parallel data. The problem is usually addressed, in part, by making use of the isomorphism of word representation spaces of different languages.

A halfway between supervised learning and unsupervised learning is **semi-supervised learning**. It deals with the case in which some of the input data is labeled and the rest is unlabeled. Thus, it can receive benefits from both supervised and unsupervised learning. For example, in machine translation, we may have a certain amount of parallel data (i.e., labeled data) and orders of magnitude larger monolingual data (i.e., unlabeled data). Often, a base system is learned on the parallel data in a supervised learning fashion. On top of it, improvements can be made from training components of the base system on large-scale

<sup>11</sup>It might be difficult and ambiguous to determine if two words are similar or not. We leave this issue to Chapter 3.

monolingual data. This is implemented by either combining a translation model learned on the bilingual data and a language model learned on the target-language monolingual data, or pre-training parts of the translation model on monolingual data and fine-tuning the entire model on the bilingual data.

Broadly speaking, all learning algorithms need supervision. This sounds weird because unsupervised learning seems to not be signaled by any ground-truth data. However, from a general learning perspective, we should not restrict ourselves to labeled data for receiving supervision signals. Even for an unsupervised learning problem, we still need to supervise the learning process by prior knowledge and hidden patterns in the input data  $\{x^{(k)}\}$ . In this sense, unsupervised learning is not “learning without supervision”.

Taking “supervision” as a concept in a broader sense, more paradigms can be seen as instances of machine learning, though not necessarily belonging to either supervised learning or unsupervised learning. An example is **reinforcement learning**. It models how a system makes a sequence of decisions. This is achieved by operating an agent in an environment. The agent receives a feedback (or a reward) from the environment when making a decision (or taking an action). The goal of reinforcement learning is to learn a decision model that maximizes the reward along the steps the agent takes. The real reward here is available only when the agent reaches some state, such as the end of a game. As such, reinforcement learning can describe problems where the reward is over a longer period (call it **distant reward**). This differentiates reinforcement learning sharply from standard supervised learning, traditionally concerned with instant supervision signals that are encoded in labeled data in advance. This characteristic fits many NLP problems. For example, a text generation system generally generates a sequence of words from left to right, but it is hard to determine if a word is properly predicted until the whole text has been generated.

Another example is **self-supervised learning**. It addresses unsupervised learning problems in a supervised learning manner. A general idea is to frame the unsupervised learning task as a pretext task that can be used in solving the original problem. In the pretext task, ground truth can be generated from input data. The learning therefore receives supervision from self-made signals instead of manual labels.

Self-supervised learning has indeed been quite successful in NLP. Perhaps **pre-training** is one of those which have made the most incredible progress. In pre-training, one can train some model (such as a language model) via self-supervised learning, and then apply parts of the model to some downstream system (such as a sentiment analysis system). It offers two advantages. First, the self-supervised, pre-trained models can be scaled to a huge amount of data because they require no labeled data. Second, pre-training is general itself and can be applied to a wide range of downstream tasks. In Chapter 7, we will see a few examples of applying self-supervised learning to NLP models.

### 1.3.2 Inductive Bias

We informally describe (supervised) machine learning problems as an **inductive reasoning** (or **inductive inference**) process: we use specific observations (such as labeled documents) to make a generalized model (such as a classifier). For example, we may observe that word

*cooking* frequently occurs in some documents talking about food. We would say that, based on inductive reasoning, *cooking* is an important indicator for all food documents. This “specific-to-general” method is also called **induction** sometimes. Once we have an induced model, we can apply it to describe new observations, as a **deduction** process.

Induction is the most widely-used principle in designing learners of modern machine learning systems<sup>12</sup>. Imagine that there is a hypothesis space (or model space, or learnable function space) consisting of all possible models that we could make. Learning a model is thus the same as selecting a model in the hypothesis space by inducing from the given training samples. However, we cannot simply assume an oracle model that works well on all unseen samples. A reason for this is that searching for the “best” model in a huge hypothesis space is computationally infeasible. There would be an infinite number of dimensions along which we can design models if the hypothesis space is unconstrained. This problem is essentially some sort of **the curse of dimensionality**<sup>13</sup>. Another reason is that many models in the hypothesis space can fit training samples well, but only some of them can fit unseen samples. There is a risk that we select a “weak” model for test data although it is “strong” for training data. This is relevant to **overfitting**, a concept that we will discuss later.

A natural solution is to define priors on the hypothesis space in a way that allows some models to be more preferable than others. A simple example is that we restrict classifiers to linear models (see Section 1.2.3). It is doing something like we impose a prior that excludes all non-linear models from the hypothesis space.

Such a prior is generally called an **inductive bias**. In a nutshell, an inductive bias is a set of assumptions on the problem<sup>14</sup>. For example, one can design models in certain mathematical forms (i.e., model bias); one can choose specific algorithms for learning a model (i.e, algorithm bias); one can assume the way of generating samples (i.e, sample bias), and so on<sup>15</sup>.

Inductive biases try to tell in what way we should describe a problem. Better results are generally favorable when inductive biases meet what really happens. This explains why solutions to some problems prefer certain model architectures (or model biases). Of course, more and stronger inductive biases could make it easier to solve a problem. However, inductive biases are not always helpful, especially when they are not close to the reality.

Let us consider a dice rolling game. Suppose you have a 6-sided dice. Before rolling the dice, you guess a side (say a number from 1 to 6). You will win if the dice lands on the same side you guess. You are a gambler and try to win as many times as possible. In your experience, a random guess is the best choice in this game because all sides should have an equal chance of appearing (i.e. a chance of 1/6). This is true when you play fair dice. However, one day,

---

<sup>12</sup>Not all machine learning methods should follow an induction process for learning a model. There are other options for different types of problems, including deductive reasoning, abductive reasoning, analogical reasoning (or transduction), and so on [Hurley, 2011].

<sup>13</sup>The curse of dimensionality refers to the problems that generally appear as the dimensionality of the hypothesis space increases. For example, data sparseness is a common problem that arises when processing high-dimensional data, and is thus a kind of the curse of dimensionality.

<sup>14</sup>A more formal definition can be found in machine learning textbooks [Mitchell, 1997]

<sup>15</sup>As an aside it is worth noting that the term *bias* is used in many different ways, and there are other meanings for *bias* in certain contexts. We will make it clear when a different meaning is used.

we played weighted dice, and it was not easy to win as before. You found that the appearance of different sides did not follow a uniform distribution. Then, you assumed a multinomial distribution (because you had the experience of developing naive Bayes classifiers). Before the game started, you rolled the dice 100 times. You chose the most frequent side for new games, and you won more. In this example, you made an initial assumption that all six of the sides are equally likely to occur. This is a very strong inductive bias because your model has 0 degrees of freedom. It seems to be obvious but does not work for weighted dice. The second inductive bias, though seems more complicated, is actually a weaker assumption, because a multinomial distribution defines a larger family of models and gives room to finding appropriate models.

In general, all machine learning models need some sort of inductive bias. Many of them are implicit assumptions. Sometimes, we are even not aware that we are making these assumptions because they are so “obvious” and “logical”. On the other hand, if the assumption is wrong then it is harmful to problem-solving. So we still need some experience to avoid easily neglected mistakes.

### 1.3.3 Non-linearity

Non-linearity is the nature of most real-world problems, whereas it is not easy to use a linear model to solve a non-linear problem. See Figure 1.3 for examples of varying degrees of classification difficulty. In Figure 1.3 (a), the two classes can be separated by a hyperplane. In this case, the problem is **linearly separable** because the decision boundary can be represented as a linear function. In contrast, in Figure 1.3 (b), we cannot draw hyperplanes to perfectly separate the two classes. Instead, we need some non-linearity for better separation, such as hyperspheres. A more difficult case is shown in Figure 1.3 (c) where the decision boundary is highly complex.

Although the theory of non-linear systems still has not been fully studied, there are several methods that help us introduce non-linearity into machine learning systems.

- **Feature mapping and kernel methods.** Recall that a linear classifier can be formulated as a function  $f(\mathbf{w} \cdot \mathbf{x})$ , where  $\mathbf{w}$  is the weight vector,  $\mathbf{x}$  is the feature vector, and  $f(\cdot)$  is the function that returns one class (say  $c_a$ ) if its argument  $> 0$  and the other class (say  $c_b$ ) otherwise. The idea of feature mapping is that we map the feature vector  $\mathbf{x}$  into a higher-dimensional space so that the problem is linearly separable in the new space. For example, let  $\phi(\cdot) : \mathbb{R}^2 \rightarrow \mathbb{R}^4$  be a mapping function and  $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$  be a 2-dimensional vector. We assume that

$$\phi(\begin{bmatrix} x_1 & x_2 \end{bmatrix}) = \begin{bmatrix} x_1^2 + x_2^2 & x_1 & x_2 & 1 \end{bmatrix} \quad (1.68)$$

By choosing  $\mathbf{w} = \begin{bmatrix} 1 & -8 & -8 & 28 \end{bmatrix}$ , we get a new classifier:

$$\begin{aligned} f(\mathbf{w} \cdot \phi(\mathbf{x})) &= f\left(\begin{bmatrix} 1 & -8 & -8 & 28 \end{bmatrix} \cdot \begin{bmatrix} x_1^2 + x_2^2 & x_1 & x_2 & 1 \end{bmatrix}\right) \\ &= f((x_1 - 4)^2 + (x_2 - 4)^2 - 4) \end{aligned} \quad (1.69)$$

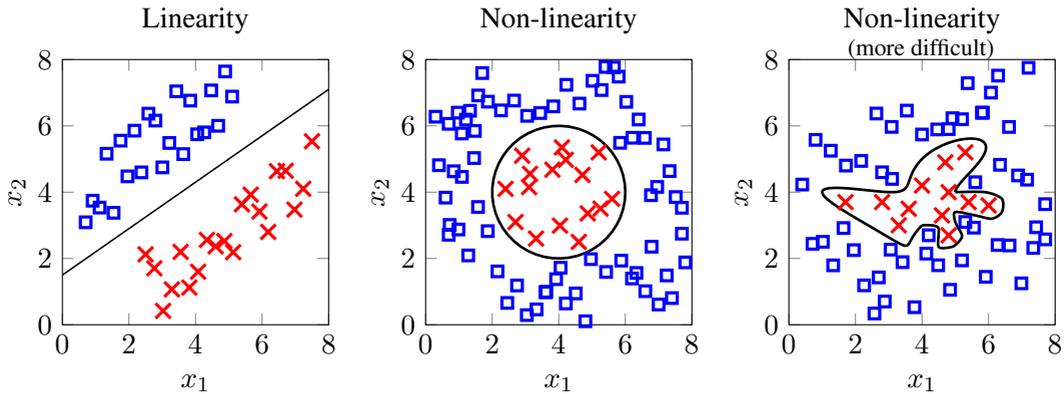


Figure 1.3: Linearity and non-linearity in binary classification. The first problem (left) is linearly separable because there exists (at least) a hyperplane that perfectly separates the data points in the two classes. The property of linear separability does not hold in the second problem (middle). Rather, we need a circle-like decision boundary. The decision boundary would be more complex if there are areas where the two classes of the data points are mixed and more or less indistinguishable (right).

It defines a decision boundary (i.e., a hypersphere  $(x_1 - 4)^2 + (x_2 - 4)^2 = 4$ ) that perfectly classifies the samples in Figure 1.3 (b). In other words, we use a linear model on the mapped feature space to create a non-linear model. However, computing the mapping function might be inefficient. This is typically addressed by using kernel methods. In kernel methods, the calculation of vector dot-product in the new space is performed efficiently by using a kernel function in the old space. This method is called **the kernel trick**. It has been successfully adopted in classification and other machine learning models, such as **support vector machines** [Cortes and Vapnik, 1995].

- **Non-linear activation functions.** Another way to add non-linearity is to use activation functions. A common method is to stack a non-linear activation function on top of a linear model. For example, the function  $f(\cdot)$  used in the above example is itself a non-linear function. There are many kinds of non-linear activation functions. We can choose from them, depending on what form of the output we want. For more sophisticated models, more activation functions can be inserted into the intermediate computing steps to develop a more powerful and expressive model. For example, a **deep neural network** is a stack of sub-models (call them layers) where each sub-model may involve one or more activation functions.
- **Non-parametric methods.** Non-parametric is a term that is originated from statistics. In non-parametric statistics, statistical inferences are made without any assumption on underlying distributions of data. In machine learning, non-parametric methods follow the same idea. They do not assume any mapping function from input to output as Eq. (1.69). This differentiates them from **parametric methods** that explicitly learn a mathematical form of variables (or parameters) to describe the problem. An example of

non-parametric methods is ***k*-nearest neighbors**. It makes a prediction for a new sample based on the  $k$  nearest neighboring samples in training data. Note that non-parametric does not mean parameter-free. Rather, it means that parameters can change. On another hand, non-parametric methods do not ensure a fixed model. They grow in model size as more training samples are available. As a reward, they can handle highly non-linear problems when training samples are sufficient.

Still, non-linear methods do not work alone. Linearity is surely an important component for most practical machine learning systems. This has two flavors. First, more non-linearity is not always better. We do not need to complicate the modeling if a linear model is enough for solving the problem. An example is that most state-of-the-art machine learning models are a combination of linear and non-linear sub-models. This is also an instance of **Occam’s Razor** — *the simplest solution is almost always the best*. The second flavor is the linear approximation of non-linear behaviors. Linear models are a good alternative if the non-linearity of the problem is not obvious. In such cases, using linear functions to approximate precise solutions is probably more efficient for practical purposes.

### 1.3.4 Training and Loss Functions

Almost all machine learning algorithms involve a training step. Typically, it refers to the process of estimating the mapping function and the associated parameters from data. Here we follow a conventional definition of the training problem: given a model or mapping function, we improve some objective by evaluating the model through some training experience [Mitchell, 1997]. For example, training a naive Bayes text classifier requires maximizing a likelihood function (i.e., the objective) on a number of labeled documents (i.e., the training experience).

Often, the training problem can be framed as an **optimization** problem. As such, we optimize some **objective function** via some training algorithm. Although an ideal objective function is a performance measure on test samples, we cannot take it in optimization since the test samples and corresponding labels are assumed to be inaccessible in the training phase. Practical objective functions are instead defined as a surrogate for the measure on test data. On the other hand, these objective functions are not necessarily some sort of performance measure, but some metrics that are assumed to correlate with the performance on test data.

Let us consider a general case. Suppose  $\mathbf{y}_\theta = f_\theta(\mathbf{x})$  is a model that reads a feature vector  $\mathbf{x}$  and produces an  $n$ -dimensional vector  $\mathbf{y}_\theta$ . For example, in text classification,  $\mathbf{x}$  is the bag-of-words representation of a document, and  $\mathbf{y}_\theta$  is a distribution over a set of classes.  $\theta$  is the parameters of the model. The subscript emphasizes that the model is determined by  $\theta$ . We further suppose that  $\mathbf{y}_{\text{gold}}$  is the gold-standard vector. Then, we define the objective function as a function that counts errors in  $\mathbf{y}_\theta$  with respect to  $\mathbf{y}_{\text{gold}}$ , denoted as  $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$ . It measures how bad it would be if we predict  $\mathbf{y}_\theta$  instead of  $\mathbf{y}_{\text{gold}}$ . Given a model, the training problem can be described as finding the “best” parameters  $\hat{\theta}$  so that  $L(\mathbf{y}_{\hat{\theta}}, \mathbf{y}_{\text{gold}})$  is minimized:

$$\hat{\theta} = \underset{\theta}{\operatorname{arg\,min}} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \quad (1.70)$$

This formulation can be easily extended to the case of  $K$  training samples:

$$\hat{\theta} = \arg \min_{\theta} \frac{1}{K} \sum_{k=1}^K L(\mathbf{y}_{\theta}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)}) \quad (1.71)$$

Once we obtain  $\hat{\theta}$ , we can use  $f_{\hat{\theta}}(\mathbf{x})$  as a fixed model for prediction.

$L(\mathbf{y}_{\theta}, \mathbf{y}_{\text{gold}})$  and  $\frac{1}{K} \sum_{k=1}^K L(\mathbf{y}_{\theta}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)})$  are usually called **loss functions** (or **cost functions**). A loss function can be defined in many ways, depending on what type of problem we address and what prior we want to impose upon training. Here, we first consider the case in which  $\mathbf{y}_{\theta}$  is a probability distribution. It is quite common in NLP, e.g.,  $\mathbf{y}_{\theta}$  could be a distribution over a vocabulary, a distribution over a list of documents, a distribution over a set of syntactic labels. For such a type of model output, the most commonly-used loss functions are measures of divergence:

- **Divergence-based Loss.** Divergence-based loss functions compute the degree of difference between the two distributions  $\mathbf{y}_{\theta}$  and  $\mathbf{y}_{\text{gold}}$ . For example, cross-entropy (see Section 1.1.2) is one of the most popular loss functions used in NLP. One can, of course, choose other divergence-based measures, such as the KL divergence and the Jensen-Shannon (JS) divergence, which can be found in most statistics textbooks. Note that MLE is also a special instance of the divergence-based objective. It is the same as the cross-entropy loss if  $\mathbf{y}_{\text{gold}}$  is a one-hot vector where the entry of the correct label is 1 and other entries are all 0.

However, machine learning systems are not always restricted to distribution-like output. Rather,  $\mathbf{y}_{\theta}$  could be a vector in  $\mathbb{R}^n$ . An example is the discriminant functions used in classification (see Section 1.2.3). They assign a score to each class, indicating how strong the model believes it is the answer. One way to define the loss functions on real-valued vectors is to transform them into distribution-like forms<sup>16</sup>, and resort to the divergence-based loss. However, normalization is not always necessary, especially when we need a score out of the range of  $[0, 1]$ . It is more common to compute losses on the raw output of these models. Here are some examples.

- **Distance-based Loss.** It is natural to take loss as some sort of distance in geometry. A general example is the  $p$ -norm distance (see Section 1.1.1):

$$L(\mathbf{y}_{\theta}, \mathbf{y}_{\text{gold}}) = \left( \sum_{i=1}^n |y_{\theta}(i) - y_{\text{gold}}(i)|^p \right)^{1/p} \quad (1.72)$$

For example, we would have a Euclidean distance-based loss function if  $p = 2$ . The distance-based loss intrinsically describes a **curve fitting** problem: we learn a curve  $\mathbf{y}_{\theta} = f_{\theta}(\mathbf{x})$  to fit the points  $\{(\mathbf{x}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)})\}$ . It is also called **regression**<sup>17</sup>. A simple

<sup>16</sup>For example, we can normalize the entries of a vector by the sum of these entries.

<sup>17</sup>When the model output is a vector with two or more dimensions, the problem is called **multivariate regression**.

example is quality estimation of machine translation<sup>18</sup>. It learns to predict translation quality (i.e.,  $y_\theta$ ) for any pair of source and target sentences (i.e.,  $\mathbf{x}$ ). We would say that the prediction is accurate if the predicted score is close to that made by humans. By using Eq. (1.72), one can design many loss functions for regression models. For example, **mean square error (MSE)** is a popular regression loss function. It is the sum of squared Euclidean distances between the prediction and the gold standard:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_{i=1}^n |y_\theta(i) - y_{\text{gold}}(i)|^2 \quad (1.73)$$

Another example is **mean absolute error (MAE)**. It is precisely the form of Eq. (1.72) when  $p = 1$ .

- **0-1 Loss.** The 0-1 loss is widely used in classification problems. It chooses a value of either 1 (penalty) or 0 (no penalty), and penalizes the case in which the predicted label and the gold-standard label are not the same. Let  $c_\theta = \arg \max_c y_\theta(c)$  be the label that is predicted by selecting the entry in  $\mathbf{y}_\theta$  with the highest value. Likewise, let  $c_{\text{gold}} = \arg \max_c y_{\text{gold}}(c)$  be the gold-standard label. The 0-1 loss is defined to be:

$$\begin{aligned} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= L_{0-1}(c_\theta, c_{\text{gold}}) \\ &= \begin{cases} 1 & c_\theta \neq c_{\text{gold}} \\ 0 & c_\theta = c_{\text{gold}} \end{cases} \end{aligned} \quad (1.74)$$

- **Margin-based Loss.** A **margin** is the difference between the predicted scores of the correct label  $c_{\text{gold}}$  and an incorrect label  $c$ :

$$\text{margin}(c, c_{\text{gold}}) = y_\theta(c_{\text{gold}}) - y_\theta(c) \quad (1.75)$$

It indicates a distinction between  $c_{\text{gold}}$  and  $c$ . So, a natural idea is to ensure that the margin is sufficiently large, or at least exceeds a minimum. This is called **large-margin training**. Let  $\Delta(c, c_{\text{gold}})$  be a predefined cost of replacing label  $c_{\text{gold}}$  with label  $c$ , satisfying  $\Delta(c, c_{\text{gold}}) \geq 0$ , and  $\Delta(c, c_{\text{gold}}) = 0$  only if  $c = c_{\text{gold}}$ . Our goal is to enlarge  $\text{margin}(c, c_{\text{gold}}) - \Delta(c, c_{\text{gold}})$ , in other words, the larger this value is, the smaller the loss is. Then, the margin-based loss is given by:

$$\begin{aligned} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= \max_c \left( 0, \max_c - (\text{margin}(c, c_{\text{gold}}) - \Delta(c, c_{\text{gold}})) \right) \\ &= \max_c \left( 0, \max_c (y_\theta(c) - y_\theta(c_{\text{gold}}) + \Delta(c, c_{\text{gold}})) \right) \\ &= \max_c \left( 0, y_\theta(c) - y_\theta(c_{\text{gold}}) + \Delta(c, c_{\text{gold}}) \right) \end{aligned} \quad (1.76)$$

<sup>18</sup>In machine translation, quality estimation comprises several different tasks (see <https://www.statmt.org/wmt21/quality-estimation-task.html>). Here we use the term to refer to the task that predicts an evaluation score directly.

Designing  $\Delta(c, c_{\text{gold}})$  depends on the problem. A simple choice is  $\Delta(c, c_{\text{gold}}) = 1$  for  $c \neq c_{\text{gold}}$ . This makes Eq. (1.76) a type of the **hinge loss**. Another variant of Eq. (1.76) is using a sum instead of a max:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_c \max\left(0, y_\theta(c) - y_\theta(c_{\text{gold}}) + \Delta(c, c_{\text{gold}})\right) \quad (1.77)$$

- **Ranking-based Loss.** Ranking-based loss (or ranking loss) is used in several different areas, such as information retrieval, classification and metric learning. It deals with the problem where we want to order a set of scored items. Suppose the model output  $\mathbf{y}_\theta$  corresponds to a set of  $n$  items  $\{c_i\}$ , each for an entry of  $\mathbf{y}_\theta$ . We define  $\{\psi_\theta(c_i)\}$  as the order of  $\{c_i\}$  by  $\{y_\theta(i)\}$ . For example, given  $\{y_\theta(i)\} = \{0.3, -2, 1\}$ , we have  $\psi_\theta(c_1) = 2$ ,  $\psi_\theta(c_2) = 3$  and  $\psi_\theta(c_3) = 1$ . Likewise, we can define  $\{\psi_{\text{gold}}(c_i)\}$  as the gold-standard ranks. Note that  $\{\psi_{\text{gold}}(c_i)\}$  can be induced in some way without the need of  $\mathbf{y}_{\text{gold}}$  if the problem only requires orders, rather than scores. An idea of ranking-based loss is to model the ranking mistakes in  $\{\psi_\theta(c_i)\}$  with respect to  $\{\psi_{\text{gold}}(c_i)\}$ . There are many ways to “count” the mistakes. A simple method is to penalize the case in which a pair of items are ordered incorrectly. As such, the ranking-based loss somewhat shares the same spirit of that used in binary classification — we categorize a pair of items as correct or incorrect. Let  $\Omega$  be a set of ordered item pairs:

$$\Omega = \{(i, j) | \phi_{\text{gold}}(i) < \phi_{\text{gold}}(j)\} \quad (1.78)$$

The loss function is given by the equation:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_{(i,j) \in \Omega} L_{\text{pair}}(y_\theta(i), y_\theta(j)) \quad (1.79)$$

where  $L_{\text{pair}}(y_\theta(i), y_\theta(j))$  is a classification loss, such as the hinge loss used in [Collobert and Weston, 2008]:

$$L_{\text{pair}}(y_\theta(i), y_\theta(j)) = \max(0, y_\theta(i) - y_\theta(j) + 1) \quad (1.80)$$

This method is called the **pairwise method**. Also, one can define the ranking-based loss in a pointwise or listwise manner. These loss functions are extensively used in developing systems to rank objects.

- **Contrastive Loss.** Contrastive loss is typically used in **contrastive learning**. It assumes that, given a sample, there is a similar sample that is labeled as “positive”, and there are a number of dissimilar samples that are labeled as “negative”. A natural idea is to minimize the distance between similar samples and simultaneously maximize the distance between dissimilar samples. Return to the formulation here. For a model output  $\mathbf{y}_\theta$ , let  $\mathbf{y}^+$  be the positive output and  $\mathbf{Y}^- = \{\mathbf{y}^-\}$  be the set of negative outputs. Also, we use  $\mathbf{y}_{\text{gold}}$  to denote the tuple of  $\mathbf{y}^+$  and  $\mathbf{Y}^-$  instead of a single gold-standard vector.

A form of the contrastive loss function is given by the equation:

$$\begin{aligned}
 L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= L(\mathbf{y}_\theta, \mathbf{y}^+, \{\mathbf{y}^-\}) \\
 &= \log D(\mathbf{y}_\theta, \mathbf{y}^+) - \log \sum_{\mathbf{y}^- \in \mathbf{Y}^-} D(\mathbf{y}_\theta, \mathbf{y}^-) \\
 &= \log \frac{D(\mathbf{y}_\theta, \mathbf{y}^+)}{\sum_{\mathbf{y}^- \in \mathbf{Y}^-} D(\mathbf{y}_\theta, \mathbf{y}^-)} \tag{1.81}
 \end{aligned}$$

where  $D(\alpha, \beta)$  is a measure of the distance between  $\alpha$  and  $\beta$ . For example, we can define  $D(\alpha, \beta)$  as the Euclidean distance (see Eq. (1.72)). A problem here is how to generate positive and negative model outputs. In the supervised learning setup, one can simply treat the gold-standard vector as the positive output. For negative outputs, the model  $f(\mathbf{x})$  can output a number of  $\mathbf{y}$  through accepting different  $\mathbf{x}$ . In the unsupervised learning setup,  $\mathbf{y}^+$  and  $\mathbf{Y}^-$  are often defined based on some “natural” annotation. For example,  $f(\cdot)$  can be a function that maps  $\mathbf{x}$  to something and back to  $\mathbf{x}$  (call it **auto-encoding**). Then,  $\mathbf{y}^+$  is  $\mathbf{x}$  itself or some neighbors of  $\mathbf{x}$ , and  $\mathbf{Y}^-$  is a set of randomly generated vectors.

- **Error-based Loss.** Evaluation metrics, as generally used in counting errors in system output, can also be taken to be part of a loss function. For example, in machine translation, a popular evaluation metric is BLEU<sup>19</sup>. Thus, we can take minimizing  $1 - \text{BLEU}$  as the objective. Let  $\text{Error}(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$  be the “number” of errors in comparing  $\mathbf{y}_\theta$  with  $\mathbf{y}_{\text{gold}}$ . The error-based loss is just the same as this number:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \text{Error}(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \tag{1.82}$$

So far we have presented several loss functions for a wide variety of problems, such as classification, regression, and ranking. As we will see in this book, different loss functions have different effects on model behavior. However, testing all possible loss functions is simply impractical because there are so many of them. Users instead need to choose or design the most suitable loss functions for their own problems. This may take time but is necessary.

On another hand, there are general methods to improve the design of loss functions. For example, we can assume that the model output  $\mathbf{y}_\theta$  is not a single vector but a variable with some probability. The loss  $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$  is thus treated as a variable too. Then, we redefine the loss function as the expectation of  $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$  under the distribution of  $\mathbf{y}_\theta$ :

$$\mathbb{L}(\{\mathbf{y}_\theta\}, \mathbf{y}_{\text{gold}}) = \mathbb{E}_{\mathbf{y}_\theta \sim \text{Pr}(\mathbf{y}_\theta|\mathbf{x})} [L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \cdot \text{Pr}(\mathbf{y}_\theta|\mathbf{x})] \tag{1.83}$$

where the use of  $\{\mathbf{y}_\theta\}$  means that  $\mathbf{y}_\theta$  is not fixed. By accessing the space of possible  $\mathbf{y}_\theta$ , it offers a better estimation of the loss. This is essentially an instance of **the Bayesian approach**.  $\mathbb{L}(\{\mathbf{y}_\theta\}, \mathbf{y}_{\text{gold}})$  is called **the Bayesian risk** or **risk** for short, sometimes.

Another way to improve training is introducing priors into the objective. A typical method

<sup>19</sup>BLEU is a precision-like score between 0 and 1. The higher the better.

is to add a regularization term  $R$  to the objective, like this:

$$\hat{\theta} = \operatorname{argmin}_{\theta} L(\mathbf{y}_{\theta}, \mathbf{y}_{\text{gold}}) + \alpha \cdot R \quad (1.84)$$

where  $R$  could be another function that describes some aspect of the problem, such as the number of parameters.  $\alpha$  is a hyperparameter controlling how much we respect  $R$  in training. The design of  $R$  is itself an important problem for many practical machine learning systems. Although we do not discuss them here, we will look at a few later in this book.

Once the objective is determined, we need some training algorithm to perform optimization. This is a very broad topic in machine learning, such that we do not even try to describe any of them in detail in this chapter. Anyway, one should not expect a universal algorithm that can solve all training problems, and there are indeed some algorithms that are suitable for certain types of problems. For example, we can use gradient descent to train a neural language model with the cross entropy-based loss [Bengio et al., 2003a], can use quadratic programming to train an SVM model with the hinge loss [Cortes and Vapnik, 1995], and can use **minimum error-rate training (MERT)** to train a statistical machine translation model with the 1 – BLEU loss [Och and Ney, 2002].

### 1.3.5 Overfitting and Underfitting

The standard process of (supervised) machine learning comprises a training step and a test step. While one may try to minimize the loss on training samples, the learned model is used to deal with new samples that are never seen before. It is like what we experienced in our lives, for example, a student studies hard and wishes to get good grades in final exams. Yes, *studying hard = good grades* should always be true, but it does not mean that memorizing all the questions and answers in textbooks is a good way to perform well in exams. It always happens that the test questions are something different from what we learned. We therefore need some ability of **generalization**.

In machine learning, generalization is used to describe how well a model learned through experience predicts on new data. A system is thought to be of excellent generalization performance if it learns little from training data but forms its prediction ability based on some “god” inductive biases on the problem. However, good generalization does not mean less training. Instead, practitioners would like to train a machine learning model on more training data to prevent it from memorizing all the things. Generalization is a very complex issue determined by several factors, including problem complexity, model architecture, amount of training data, training algorithm and so on. While there are no standard rules to ensure good generalization, researchers always try to address it somehow.

To describe how well a model generalizes to new data, there are two important terms, **underfitting** and overfitting. Underfitting refers to the phenomenon that a model does not learn sufficiently from the training data and thus has poor performance on new data. For example, we interrupt training accidentally and deploy the immature model for prediction. The model cannot perform well on either the training data or the test data. If a model underfits the training data, then one could improve it in some simple ways. For example, one could train the model

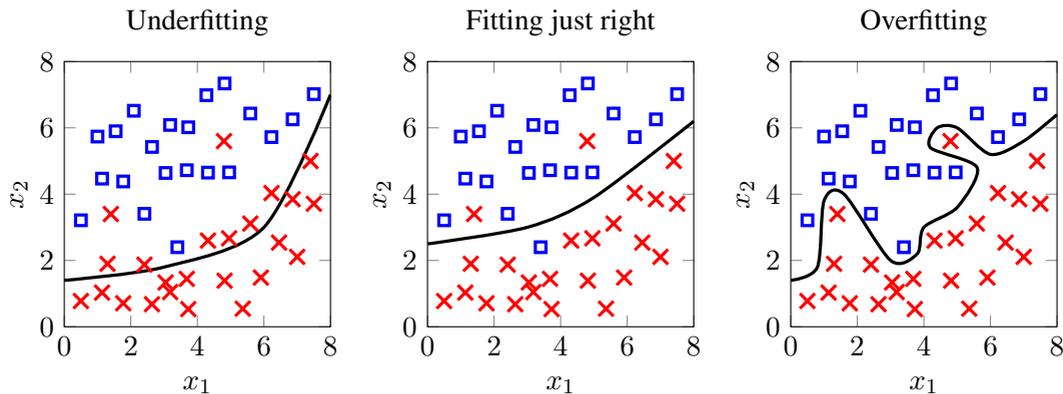


Figure 1.4: Decision boundaries of a binary classification problem. left = underfitting, right = overfitting, and middle = fitting just right. In the underfitting case, there are several obvious mistakes that are made in separating the two classes of data points. By shifting the decision boundary up a bit (middle), we obtain a satisfactory separation result, where most of the data points belonging to the same class are placed on the same side of the decision boundary. By contrast, a perfect separation requires a highly complex decision boundary instead (right).

for a longer time; one could remove unimportant portions from the training data; one could use a model with a simpler architecture instead.

In contrast to underfitting, overfitting refers to the phenomenon that a model fits the training data well but generalizes poorly on the test data (see Figure 1.4). A simple example of overfitting here is the OOV problem (see Section 1.2.5). It would be a disaster if a text classification model just fits those words that have been seen but gets stuck when new words appear.

The causes of overfitting are diverse. An example is learning a complex model on a small training dataset. The model complexity often matters when we design a machine learning model. If the model is complex and has many parameters, then it would be much easier to overfit a small number of samples (see Figure 1.4). The problem would be more difficult if there is noisy data, because of the errors of “garbage in, garbage out” in training. In addition, excessive training is another cause of overfitting. For example, we can heavily tune a system to enforce it to model the data with no errors. The system would be fragile for new samples, even when there are small fluctuations in input.

Overfitting can be alleviated in many ways. Here are some commonly-used techniques.

- **Using more (high-quality) training data.** Large-scale training helps the model capture the true patterns in data. However, adding noisy data would do this in a negative way.
- **Using validation data.** Validation data is some test data but used in training. For example, a dataset can be divided into **held-out data** and training data. One can simply **early stop** the training process when the performance drops on the held-out data.
- **Using simpler model architectures.** As noted previously, Occam’s Razor is a principle

we can follow in model design. Models with more complex architectures, though powerful, would be more likely to fit the noisy data points if the problem is not so difficult itself. Using a simpler model architecture instead could make it easier to model the dominant patterns in the data.

- **Regularization.** Regularization is another way to control the model complexity. Typically, it regularizes model parameters by priors. An example is smoothing (see Section 1.2.5). It re-estimates the distribution of words after training. A more general method is regularized training (see Eq. (1.84)). For example, we can define the regularization factor as the  $l_1$  norm of the parameters, and bias the model to those whose parameters are not in large absolute values.
- **Combining multiple models.** A better prediction can also be made by ensembling multiple models. These models (call them **component models**) are in general of different parameters or architectures, and/or are trained with different portions of the data. The variance in models can reduce the risk that all these models overfit the data in exactly the same manner. These models are, therefore, less likely to make similar mistakes in prediction.

### 1.3.6 Prediction

Although we restricted our discussion to classification in previous sections, (supervised) machine learning is not just a task of predicting a label for an input object. There are many types of machine learning problems, depending on what form of the prediction is defined.

- **Classification.** Classification is perhaps one of the most common machine learning problems. A classification system is required to assign one or more classes to an input object.
- **Regression.** In statistics, regression studies the relationship between a **dependent variable** (or an outcome) and an **independent variable**. While regression has many applications, it is often framed as score prediction in NLP. For example, taking a movie review as input (i.e., an independent variable), the regression model learns to predict a recommendation score (i.e., a dependent variable).
- **Ranking.** A ranking model is to predict the order of a set of input objects. For example, a model ranks a number of translations in terms of translation quality.
- **Structure prediction.** Many machine learning models are required to output not only a real value or a class but a tree or a sequence. The task of predicting structured outputs is called structure prediction. For example, a syntactic parser is a structure prediction system, as its output is a tree structure.

In addition to these, **mining** is a term that is frequently used in the community, although it is somehow not a standard machine learning problem. The problem of mining refers to discovering unknown patterns in the data. An example we would like to categorize into this is word clustering. Given a number of words, the clustering system “predicts” the cluster for each word. The output of such systems is not pre-defined. Patterns in data are themselves hard

to describe. Thus, the term “mining” could cover a range of problems. To avoid confusion, we will use more specific terms (such as word clustering) to refer to mining-related problems.

Despite a fundamental aspect of machine learning, prediction is conventionally assumed to be trivial, given that many models and methods are tested on standard classification and regression tasks. On the other hand, prediction is non-trivial in structure prediction, such as parsing and machine translation, which are very common in NLP. Essentially, predicting a tree or a sequence is a **search problem**. For example, there exist a theoretically infinite number of translations given a source-language sentence. Even if we have a model to evaluate every translation, finding the optimal translation in the search space is obviously a computational challenge. In such cases, we need some way to make it feasible to perform search. This is implemented by either resorting to the general search algorithms in artificial intelligence or developing new algorithms for specific problems. As an aside, the study on the search problem offers a new view on the mistakes made by a machine learning model: some of the errors are due to inaccurate modeling (call them **model errors**), and the rest are due to inaccurate search (call them **search errors**). For prediction, eliminating search errors is a goal but often at the cost of a considerably large amount of search effort. We sometimes must trade off between efficiency and accuracy if a machine learning model is deployed for practical purposes. We will see a few examples in Chapter 5.

## 1.4 Model Selection and Evaluation

For most machine learning problems, the goal is to find a model that would perform the best on new data. Two problems can be separated out from this goal [Hastie et al., 2009]:

- **Model selection.** Selecting the best model on training data by some criteria.
- **Model evaluation.** Estimating the performance of a given model on new data.

As noted in Section 1.3.4, loss functions (or error functions) are common ways of measuring errors in a prediction  $\mathbf{y}_\theta = f_\theta(\mathbf{x})$  with respect to a gold-standard  $\mathbf{y}_{\text{gold}}$ . Given  $K$  labeled training samples  $\{(\mathbf{x}^{(1)}, \mathbf{y}_{\text{gold}}^{(1)}), \dots, (\mathbf{x}^{(K)}, \mathbf{y}_{\text{gold}}^{(K)})\}$ , the **training error** is given by

$$\text{Err}_{\text{train}} = L(\{\mathbf{y}_\theta^{(k)}\}, \{\mathbf{y}_{\text{gold}}^{(k)}\}) \quad (1.85)$$

where  $\{\mathbf{y}_\theta^{(k)}\}$  are the predictions over the training dataset, and  $\{\mathbf{y}_{\text{gold}}^{(k)}\}$  are the corresponding gold-standards.  $L(\{\mathbf{y}_\theta^{(k)}\}, \{\mathbf{y}_{\text{gold}}^{(k)}\})$  is in general defined as the averaged loss over all training samples:

$$L(\{\mathbf{y}_\theta^{(k)}\}, \{\mathbf{y}_{\text{gold}}^{(k)}\}) = \frac{1}{K} \sum_{k=1}^K L(\mathbf{y}_\theta^{(k)}, \mathbf{y}_{\text{gold}}^{(k)}) \quad (1.86)$$

or defined as a single measure on the entire set of training samples. Likewise, we can define the **test error** on the test dataset, denoted as  $\text{Err}_{\text{test}}$ .  $\text{Err}_{\text{test}}$  is also called **generalization error**. It indicates how well a model generalizes to new data.

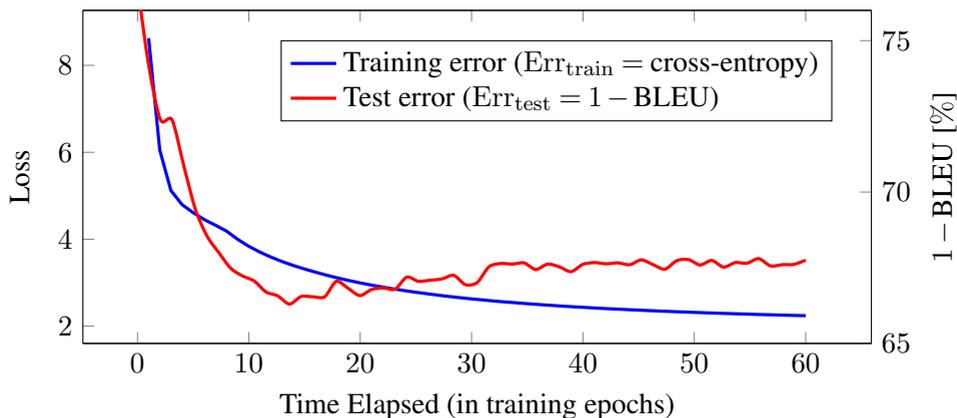


Figure 1.5: Curves of training error and test error for a machine learning system. The training error is measured in terms of the cross-entropy loss, and the test error is measured in terms of  $1 - \text{BLEU}$ . All statistics are collected by running a neural machine translation system on the IWSLT De-En benchmark. The training error continues to drop as more training epochs are involved. The test error, on the other hand, follows a trend of first going down and then going up. When the test error starts to increase, the model is likely to overfit the training data.

In the preceding sections we assumed that minimizing  $\text{Err}_{\text{train}}$  is the objective of training, i.e.,  $\hat{\theta} = \arg \min_{\theta} \text{Err}_{\text{train}}$ . However, we cannot assume that  $f_{\hat{\theta}}(\cdot)$  can obtain the minimum  $\text{Err}_{\text{test}}$  in the same way. See Figure 1.5 for learning curves of a machine translation system. Clearly,  $\text{Err}_{\text{test}}$  does not correlate with  $\text{Err}_{\text{train}}$  well. The training error keeps reducing as training proceeds. However, the test error goes up after following the same trend as the training error for a period of time, indicating overfitting of the model. This makes the problem a bit more complicated, as we cannot always trust  $\text{Err}_{\text{train}}$  although it is and should be the measure of the goodness of training. Surely, we need some way to select a better model, in addition to looking at  $\text{Err}_{\text{train}}$  only.

### 1.4.1 Strategies for Model Selection

Choosing the optimal model on the training data is challenging because the motivation here is “greedy” itself — we hope that a machine learning model can generalize from a finite, even a “small” number of samples. From the statistical learning point of view, the challenge is due to the way we define the learning problem. An implicit assumption in machine learning is that all data is generated by some distribution. Thus, the learning problem is determined by generating the training data via a data-generation distribution and the test data via another distribution.

For example, if both the training and test datasets are sufficiently large and obtained via the same data-generation distribution, then the learned model can perform on the test data as well as on the training data. In this case, it is easy to generalize the model from the training data to the test data. By contrast, if all training and test data is generated in an arbitrary manner (say a uniform distribution over the entire space of data points), then the model will fail to generalize, as everything learned on the training data does nothing with the test data.

It will be more interesting if we consider all possible problems. **The no free lunch theorem** states that all learning algorithms will perform equally well if we average the test error over all problems<sup>20</sup>. In other words, all learning will make no sense if there is no preference for certain problems. However, developing a universally good machine learning model on all problems is idealistic. In real-world applications, the training and test data is always assumed to at least in part follow some distribution. Therefore, there are indeed some ways to capture this distribution and improve the generalization ability of a model. Two scenarios are generally considered in improving machine learning systems:

- Given the model design and the training algorithm, how to develop or select training data to reduce the test error.
- Given the training data, how to develop or select a model to reduce the test error.

The first scenario is complicated and relates to many practical issues, e.g., annotation, data cleaning, data quality estimation and so on. Since these issues are not the focus for model selection, we do not discuss them but leave some to subsequent sections. Here, we focus on the model selection problem in the second scenario.

## 1. Model Complexity

The simplest method of model selection might be testing the models on validation data. Typically, this data does not overlap with either training or test data, but is assumed to be generated in the same way as the test data. However, such data is not always available. In some cases, we do not even know anything about the test data. So many model selection methods are validation-free.

A common way is to use **model complexity** (or **model capacity**) as an indicator of the selection. In machine learning, model complexity can be interpreted in several different ways. For example, a non-linear model is intuitively more complex than a linear model. Also, a model with more parameters is more complex than a model with fewer parameters under the same model architecture. More formal definitions could be found in the theoretical part of machine learning, such as **the Vapnik-Chervonenkis dimension** or **the VC dimension** [Vapnik and Chervonenkis, 1971]. Here we simply treat model complexity as a measure of the expressive power of a model, i.e., a higher model complexity indicates more hypotheses that the model can express.

While complex models are usually assumed to be more powerful, higher model complexities are not always helpful. In fact, complex models are more likely to overfit the data, especially when a small dataset is used for training. By contrast, too simple models are often prone to underfitting. We therefore need to seek an “optimal” level of model complexity. Figure 1.6 plots training and test errors against model complexity. An “optimal” complexity can be chosen when the training error tends to convergence. While Figure 1.6 shows an intuitive example, it is still hard to say at what point we can choose the model. The common practice, though not formally described in most cases, is to choose among those “good” models by using

---

<sup>20</sup>The no free lunch theorem was originally presented in a classification scenario [Wolpert, 1996], and was further extended to search and optimization problems [Wolpert and Macready, 1997].

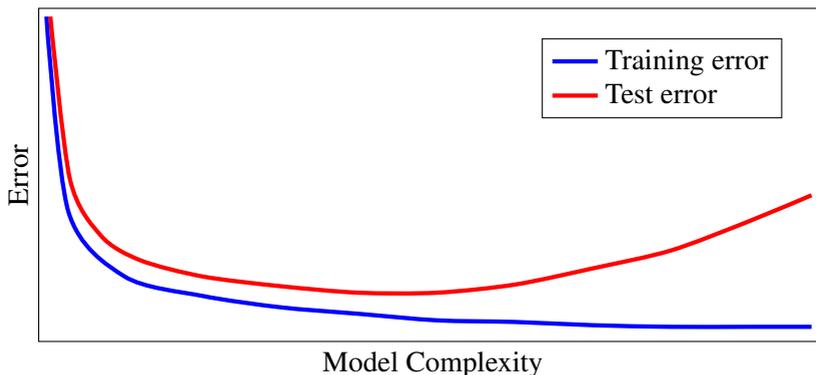


Figure 1.6: Curves of training error and test error under different model complexities. Complex models help in reducing the training error as they can compute complex functions in fitting data points. However, a too large model complexity is more likely to lead to overfitting and is harmful to the generalization ability of the models. For example, the test error increases as more complexity is added.

Occam’s Razor. Suppose we have a set of models that perform comparably well on the training data but are of different complexities. According to Occam’s Razor, the simplest model is the “best” choice. Many criteria are available to measure the model complexity. For example,

- **Number of parameters.** Though very simple, counting the number of parameters is the most intuitive yet effective method. It can be extended to counting the **effective number of parameters** which is defined to be the trace of the matrix used to transform  $\mathbf{y}_{\text{gold}}$  to  $\mathbf{y}_{\theta}$ .
- **$p$ -norm of parameters.** The  $p$ -norm of a parameter matrix is also an indicator of how complex a model is (see Section 1.1.1). For example, according to the  $l_1$  norm, a model with larger absolute values for parameters is more complex.
- **Description length.** Description length is a term used in data compression. For example, it could be the number of bits used to store a model. Thus, the **minimum description length** (or MDL) indicates the most compressed model.
- **The VC dimension.** It is originally from computational learning theory. In short, the VC dimension can be defined as the maximum number of data points that can be shattered by the classifier.

In addition, there are other choices for defining the criterion, including **the Akaike information criterion (AIC)**, **the Bayesian information criterion (BIC)**, **the minimum message length (MML)** and so on. They can be found in most textbooks on statistics and/or statistical learning [Burnham and Anderson, 2002; Konishi and Kitagawa, 2007; Hastie et al., 2009].

## 2. Bias-Variance Tradeoff

Controlling model complexity to avoid overfitting and underfitting is also linked to the tradeoff between bias and variance. Bias (or prediction bias) is the amount that the model prediction differs from the true value. In statistics, bias is a **systematic error** that cannot cancel out even if we run a large number of repeated experiments. In general, bias error results from the wrong assumptions about the problem, such as approximating a non-linear problem via a linear model. This is very interesting! We can establish the connection of the bias error here with the inductive bias used in mode design (see Section 1.3.2). For example, given training data, a large bias model is usually due to the fact that there are more assumptions and the model is not complex enough. To make it simple, we would say that more (or stronger) inductive biases can result in a lower model complexity and more bias error in prediction. Occasionally, the term *bias* is used as a short for both bias in prediction (from a statistics perspective) and inductive bias (from a model design perspective), although they are considered to have different meanings<sup>21</sup>.

Variance, on the other hand, describes how spread the prediction is when there are variations in training data. The variance error also correlates with model complexity. For example, a complex model tends to exhibit higher variance.

Both bias and variance are sources of errors of a system. A common example is the bias-variance decomposition of mean squared error. Here we use some notation that differs slightly from that used in previous sections. Let  $D$  be a set of  $K$  training samples and  $f_{\hat{\theta}(D)}(\cdot)$  be a model leaned on  $D$ . Further, given a new sample  $\mathbf{x}$ , let  $\mathbf{y}_{\hat{\theta}(D)} = f_{\hat{\theta}(D)}(\mathbf{x})$  be the model prediction and  $\mathbf{y}_{\text{gold}}$  be the “true” prediction. The bias and variance are defined as:

$$\text{bias} = \mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}] - \mathbf{y}_{\text{gold}} \quad (1.87)$$

$$\text{variance} = \mathbb{E}_D [(\mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}] - \mathbf{y}_{\hat{\theta}(D)})^2] \quad (1.88)$$

where  $\mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}]$  is the mean of  $\mathbf{y}_{\hat{\theta}(D)}$  over all possible  $K$  sample training datasets. Thus, the bias is some sort of difference between the mean and the true value, and the variance is some sort of difference between the mean and the predicted value. Taking the mean squared error as the error measure, we can write the expected error as:

$$\begin{aligned} \text{error} &= \mathbb{E}_D [(\mathbf{y}_{\hat{\theta}(D)} - \mathbf{y}_{\text{gold}})^2] \\ &= \text{bias}^2 + \text{variance} \end{aligned} \quad (1.89)$$

For lower mean squared error, reducing both bias and variance simultaneously is obviously an ideal goal. However, it is difficult to make a model that exhibits both low bias and variance. When one of the two decreases, the other increases (see Figure 1.7). Researchers must choose the optimal level of model complexity while preventing training from overfitting and underfitting. This also depends on the problem we intend to solve. For example, a simple

<sup>21</sup>Bias is more often used in statistics to describe some aspect of an estimator.

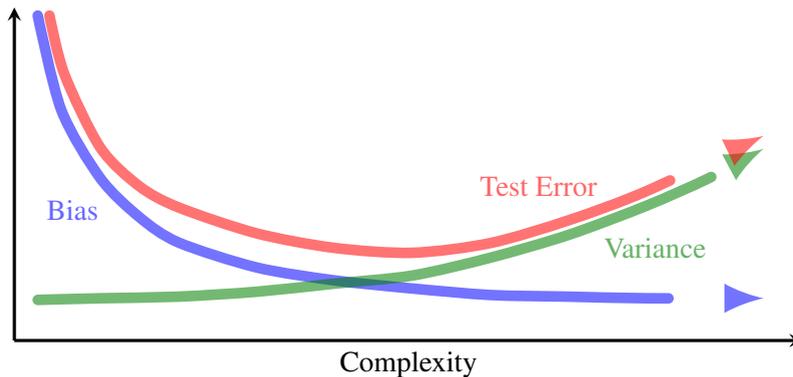


Figure 1.7: Bias and variance against model complexity [Goodfellow et al., 2016]. The curves show a conflict in reducing the bias error and the variance error simultaneously. By varying the model complexity, one can obtain either a low-bias, high-variance model or a high-bias, low-variance model. Both of the two cases exhibit high test error. For example, a high-variance model is often of a larger model complexity. While such a model is able to deal with complex problems, it is more likely to overfit the data. On the other hand, a high-bias model often means a simpler model but tends to underfit the data. To improve the generalization on test data, one can seek a tradeoff between bias and variance. For example, there is low test error when a “middle sized” model is chosen.

model generally has low variance but high bias. However, if we use the simple model (say a linear model) to describe a complex problem (say a non-linear problem), then underfitting would probably occur because the problem is too “hard” for the model.

Returning to the model selection problem, the bias-variance tradeoff is not a rule for model selection, but a principle we must keep in mind. Often, one needs to make compromises to create a model that makes reasonably good predictions. It is also worth noting that, in many applications, complex models are usually accompanied with the inefficiency problem. An appropriate method might be to start with a simple model and only add complexity when it is needed.

### 3. Model Combination

Selecting from a set of models is not the only way to reduce generalization error. Alternatively, one can do this in the opposite way, and combine these models for a “stronger” model. Such a method is called **ensemble learning** [Seni et al., 2010; Zhou, 2012a]. A key idea of ensemble learning is to create a set of component models (or ensemble models), such that they can vote for a better prediction. The simplest of these is a **mixture model** that averages the predicted scores of multiple component models (call it **model averaging**), whereas a more sophisticated method can combine the sub-structures of these models.

Component models are in general generated in some way that they can exhibit some diversity. For example, they can be learned on different portions of the training data, or by using different initializations for model parameters. Interestingly, it is found that such methods

can guarantee the reduction of generalization error somehow. For example, bagging helps to lower variance [Breiman, 1996], and boosting helps to lower bias [Schapire, 1990]. These are linked back to what we presented in Section 1.4.1: the generalization error can be reduced by either reducing the bias error or reducing the variance error.

But discussing how to combine models is beyond the scope of this chapter. While it is even not appropriate to categorize model combination as a topic related to model selection, it can be seen as a means of improving the generalization ability. In this sense, both model combination and model selection address problems on a similar theme. In fact, model combination is remarkably effective for many NLP tasks. For example, most state-of-the-art systems in NLP are based on the combination of multiple models.

## 1.4.2 Training, Validation and Test Data

We turn now to the data problem. As discussed in the previous sections, in the training stage, a training dataset is used to fit the parameters of the model. In the test stage, a test dataset is used to evaluate the learned model. Closely related to test data is validation data, which has come up a few times in this chapter. A validation dataset is a test dataset as well but can be used in the training stage. It is commonly used for model selection and tuning hyperparameters.

In many cases, one may imagine that there is some data for training and some additional data for validation and test. This assumption, however, is not realistic in many real-world applications. For example, developers cannot always access the data of system use after deploying a system. From a scientific point of view, there is no “real” new data for test — when you see new data, it is not new anymore. Therefore, what we address is essentially an analogue of the problem.

A simple method, as in many research papers, is to verify machine learning models on benchmark tasks. In these tasks, all data is prepared in advance, and all you need is to run your models on the data. Such a method makes it easy to compare different systems directly, as all these systems are trained and tested on the same datasets. Occasionally, we are just given a number of samples but not told which are for training and which are for test. In such cases, the data can be divided into parts each of which is used for some purposes. For example, a split could be 60% for training, 20% for validation, and 20% for test.

While data splitting provides a way to assess the performance of a model, the assessment result is not always stable due to sampling bias. Sometimes, the performance varies greatly across different runs of data splitting. The problem is more obvious when the dataset is too small to perform sufficient training or test.

A common way to weaken the effect of this bias is **cross-validation**. Cross-validation is a resampling method. Each round of cross-validation is a new split of data and the result is the combination of the assessment over the rounds. A simple method is random subsampling that repeats random partition of the data and averages the performance over runs. Another method is  $k$ -fold cross-validation. It divides the data into  $k$  parts. In each round of cross-validation, some parts are used as training data, and other parts are used as validation and test data. For example, in 10-fold cross-validation, a model can be trained and validated/tested for 10 times, each choosing one of the ten parts as the test dataset.

Another note on the scale of data. For practitioners, one of the most frequent questions is how many samples are enough for learning a good model. This may be the most difficult question on which different people can have consensus answers. There are many theoretical results that can tell the bound of errors given a certain amount of data, whereas in most cases we just simply follow the “the more the better” idea. In another line of thought, a system could be **sample efficient**. In general, a sample efficient system can reach a good level of performance by using fewer samples or seeing the same sample for fewer times. For example, tuning a pre-trained model is sample efficient because the samples are not used for learning from scratch but a modest update of the model. Another example is **few-shot learning**. It aims to generalize from observing very few samples for a task.

### 1.4.3 Performance Measure

As an essential part of every machine learning problem, a performance measure describes how well a system performs given some data. Usually it is used in either designing the training objectives or evaluating the result of the final system. For example, all those loss functions described in Section 1.3.4 are some kinds of performance measures.

As for evaluating the performance on test data, a measure is often designed in a way that we can count the real errors. Thus, re-using the loss functions in training might not be a good choice for reporting the final score. For example, the widely-used measures for classification problems are precision, recall and  $F_1$  score. They are proposed to quantify the ability of a classification system in certain aspects: given a class  $c$ , precision computes the fraction of correct predictions in predicting  $c$ , and recall computes the fraction of correct predictions on all samples labeled as  $c$ . The  $F_1$  score is a measure that combines precision and recall.

Notice that performance measures are not necessarily designed for optimization. In this sense, they may not guarantee some mathematical properties, such as differentiable and continuous functions. An example is the BLEU metric used in machine translation. BLEU is a function combining precision scores and a penalty score [Papineni et al., 2002]. This in turn makes the metric non-differentiable and discontinuous. In NLP, there are many such evaluation measures that are ad-hoc for certain tasks. These raise an interesting problem that the loss function used in training may differ from what we actually use in evaluating the final model. Thus, one sometimes needs to take into account the discrepancy between the objectives of training and test.

Another problem with performance measures in NLP is that there might be two or more “answers” for the same “question”. For example, there are generally multiple good translations for a source-language sentence. One solution is to take multiple gold-standards into account when designing a performance measure. BLEU is such a case. It counts the maximum number of the correct translation segments over all reference translations. The second solution involves human evaluation. Such a way of evaluation is more accurate but of course is more expensive. When developing practical systems, practitioners usually train and tune the systems using automatic measures, and call for human evaluations for the final test.

### 1.4.4 Significance Tests

Now, assuming you are improving a system in some way, you might be wondering if the improvement is significant enough or not. All you have is a performance measure. So you can tell the performance difference between any two points in developing the system, but you cannot tell if the difference is real or happens by chance.

In this example, you implicitly try to reject or accept a claim that a system is better than another system (or not). In statistics, **significance tests** are a method to model this problem. Suppose we have two systems  $A$  and  $B$ . And there are a number of datasets on each of which we evaluate the two systems via the same performance measure. Then, we make two hypotheses

$H_0$ : System  $A$  performs worse than or equally well as system  $B$ .

$H_1$ : System  $A$  performs better than system  $B$ .

where  $H_0$  is the **null hypothesis**, and  $H_1$  is the **alternative hypothesis** that is contradictory to the null hypothesis. By testing these hypotheses, we can claim that system  $A$  is significantly better than system  $B$  (i.e., reject  $H_0$  and accept  $H_1$ ) or not (i.e., accept  $H_0$  and reject  $H_1$ ). We probably make errors in the test, for example, incorrectly rejecting a true null hypothesis (type I error), or incorrectly accepting a false null hypothesis (type II error). The two types of errors are at odds with each other. A decrease of one may lead to an increase of the other. Alternatively, we can decrease one while guaranteeing that the other is upper bounded. For example, we can reduce the type II error as much as possible, and keep the type I error below a constant  $\alpha$ .  $\alpha$  is typically called the **significance level** of a test. It is standard practice to choose the significance level in the interval  $[1\%, 5\%]$ . When conducting statistical testing, we can obtain the probability of the type I error (call it a  $p$ -value). A  $p$ -value that is lower than the significance level can make a rejection of the null hypothesis. For example, in the above example, with a significance level of 5%, a  $p$ -value = 3% means that the improvement is statistically significant. For more information about the  $p$ -value, we refer the reader to other books on statistics [McClave and Sincich, 2006; Freedman et al., 2007; Freedman, 2009].

Note that the conclusion of significance tests depends on several factors, such as the number of experiments and the variance in the results of experiments. A problem with applying significance tests to NLP tasks is that there are often very few datasets for running the experiments [Dror et al., 2020]. Ideally, we know the true data distribution and can consider it in the test. This method is called the **parametric test**. If we cannot find the true data distribution, then, as a **non-parametric** test method, we can generate a number of experiments by sampling over a dataset or adding randomness into the test.

Significance tests are important for drawing convincing conclusions in developing machine learning systems, although they are often ignored unintentionally. Figure 1.8 shows evaluation results of three models. Each of them is run for several times with different initial parameters. While system  $A$  is superior to system  $B$  in terms of the averaged performance, there are large variances in their results. The significance test indicates that the difference is not significant. By contrast, the difference between system  $A$  and system  $C$  is significant because their performance differs greatly enough in most cases. On the other hand, researchers have found

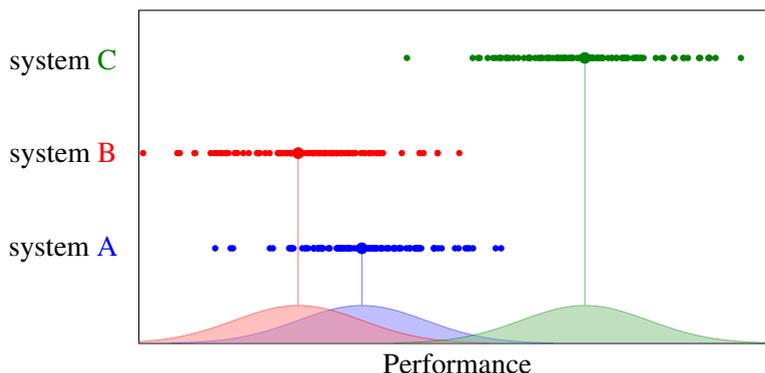


Figure 1.8: Performance of three machine learning systems. For each system, there are many different results because we introduce some randomness into training (e.g., data shuffling, random starting points, etc). Although it seems that System A outperforms System B, there is no real distinction between them, because they overlap a lot in the distributions of the performance (see the bottom of the figure). When comparing System C with System A or B, the difference in performance is significant because we could accept the  $H_1$  hypothesis (i.e., System C outperforms System A or B) given a large number of experiments.

that there are indeed some thresholds of performance gain to indicate significance under certain circumstances. For example, we would say that the significance can be roughly indicated by a certain metric gain if we compare similar systems [Berg-Kirkpatrick et al., 2012].

## 1.5 NLP Tasks as ML Tasks

While there are a wide variety of NLP tasks, many of them can be formulated as the same machine learning problem. This enables a universal solution to a group of NLP problems by using a general machine learning approach. Typically, an NLP task can be described as learning to map language units to some output. Following the notation used in this chapter, we use  $\mathbf{x}$  to denote the input feature vector (or matrix) of an NLP task, and use  $f(\mathbf{x})$  to denote the function that is learned to process  $\mathbf{x}$ . Here are some of the common tasks in NLP.

### 1.5.1 Classification

Suppose there are a set of classes or labels  $C$ . Each class is represented by a distinct integer in  $\{1, \dots, |C|\}$ . A classification model is a function that maps the input  $\mathbf{x}$  to a  $|C|$ -dimensional vector  $\mathbf{y}$ , i.e.,  $\mathbf{y} = f(\mathbf{x})$ . Each entry of  $\mathbf{y}$  is a score corresponding to class  $i$ , denoted by  $y(i)$ . The task here is to assign  $\mathbf{x}$  to one or more classes having the highest scores. Consider single-label classification as an example. The prediction is given by the equation

$$\hat{c} = \underset{1 \leq i \leq |C|}{\operatorname{argmax}} y(i) \quad (1.90)$$

where  $\hat{c}$  is the “best” class assigned to  $\mathbf{x}$ . Sometimes, one needs a probability-like output (see

Section 1.2.1). Let  $\psi(\cdot)$  be a function that normalizes a vector into a distribution<sup>22</sup>. We then obtain a probabilistic classifier:

$$\mathbf{y} = \psi(f(\mathbf{x})) \quad (1.93)$$

Classification may be the most common problem in NLP. There are many applications in addition to categorizing documents into predefined classes. Among them are choosing a sense for a word [Yarowsky, 1994], determining the polarity of a sentence [Pang et al., 2002], checking whether two entities should be linked [Krebs et al., 2018], classifying the way of associating a semantic argument with a verb [Gildea and Jurafsky, 2002], and so on. When adapting a classification model to these tasks, all you need is to design the form of  $\mathbf{x}$  and the set of classes.

## 1.5.2 Sequence Labeling

An extension to standard classification is to classify a set of samples simultaneously. **Sequence labeling** is an example of such a problem. In sequence labeling, the input is a sequence of  $n$  tokens, such as a sequence of  $n$  words. A sequence labeling system is required to assign each input token  $\mathbf{x}(i)$  a label  $l(i)$ . Here the boldface in  $\mathbf{x}(i)$  is used to emphasize that the token is represented as a feature vector. For convenience, we write  $\mathbf{x}(i)$  as  $\mathbf{x}_i$  and  $l(i)$  as  $l_i$ . The function  $f(\cdot)$  maps the sequence  $\mathbf{x}_1 \dots \mathbf{x}_n$  into another sequence  $\mathbf{y}_1 \dots \mathbf{y}_n$ , where  $\mathbf{y}_i$  is the output vector corresponding to  $\mathbf{x}_i$ . This can be formulated as:

$$\begin{bmatrix} \mathbf{y}_1 & \dots & \mathbf{y}_n \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 & \dots & \mathbf{x}_n \end{bmatrix}\right) \quad (1.94)$$

For vector  $\mathbf{y}_i$ , each entry  $y_i(c)$  corresponds to the prediction score of a class  $c \in C$ . Note that  $f(\cdot)$  allows for the use of a larger context. For example, one can condition the prediction  $\mathbf{y}_i$  on the entire input sequence [Lafferty et al., 2001]. The final output of the system can be defined as the “optimal” label sequence induced from  $\mathbf{y}_1 \dots \mathbf{y}_n$ . A simple method is to choose the label sequence that maximizes the sum of the scores over all positions, like this

$$\begin{bmatrix} \hat{l}_1 & \dots & \hat{l}_n \end{bmatrix} = \arg \max_{l_1, \dots, l_n \in C} \sum_{i=1}^n y_i(l_i) \quad (1.95)$$

A straightforward application of sequence labeling to NLP is to tag each token of the input sequence, such as **part-of-speech tagging** (or **POS tagging**). Furthermore, sequence labeling

<sup>22</sup>A similar idea can be found in Eq. (1.48). Given a vector  $\mathbf{a} = [a(1) \dots a(n)]$ , the normalization function has the form:

$$\psi(\mathbf{a}) = \left[ \frac{a(1)}{\sum_{i=1}^n a(i)} \quad \dots \quad \frac{a(n)}{\sum_{i=1}^n a(i)} \right] \quad (1.91)$$

Another way is using the Softmax function:

$$\psi(\mathbf{a}) = \left[ \frac{\exp(a(1))}{\sum_{i=1}^n \exp(a(i))} \quad \dots \quad \frac{\exp(a(n))}{\sum_{i=1}^n \exp(a(i))} \right] \quad (1.92)$$

Tokens :	Most	are	expected	to	fall	below	previous-	levels	.
							month		
POS tags :	JJS	VBP	VBN	TO	VB	IN	JJ	NNS	.
Chunk tags :	<u>B-NP</u>	<u>B-VP</u>	<u>I-VP</u>	<u>I-VP</u>	<u>I-VP</u>	<u>B-PP</u>	<u>B-NP</u>	<u>I-NP</u>	O
	NP		VP			PP		NP	

Figure 1.9: An example of sequence labeling for POS tagging and chunking. The example is from the training data of the CoNLL 2000 shared task. Each token is labeled with a POS tag and a chunk tag. A chunk tag has an initial character chosen from {B, I, O}, where B = beginning of a chunk, I = inside a chunk, and O = outside a chunk. So, a chunk always starts with a “B” tag, optionally followed by “I” tags. For example, the VP (verb phrase) chunk in the example spans over the chunk tag sequence “B-VP I-VP I-VP I-VP”.

is able to deal with more complex problems by using labels in a clever way. A well-known example is the use of the “IOB” label format in identifying chunks spanning multiple tokens (call it **chunking**). In this method, “I”, “O” and “B” stand for a token inside a chunk, a token outside a chunk, and the leftmost token of a chunk [Ramshaw and Marcus, 1995]. As such, a chunk always starts with a “B” and ends just before the next “B” or a new “O”. See Figure 1.9 for POS tagging and chunking results on an example sentence. As sequence labeling allows the labeling of both tokens and spans, it has been applied with strong results to many tasks, including POS tagging [Bahl and Mercer, 1976], chunking [Tjong Kim Sang and Buchholz, 2000], **named entity recognition (NER)** [Tjong Kim Sang, 2002], and so on.

### 1.5.3 Language Modeling/Word Prediction

**Statistical language modeling** (or **language modeling** for short) is a task of assigning a probability  $\Pr(w_1, \dots, w_n)$  to a sequence of words  $w_1 \dots w_n$ . This joint probability is generally decomposed into a product of conditional probabilities, by using the chain rule:

$$\begin{aligned} \Pr(w_1, \dots, w_n) &= \Pr(w_1) \cdot \Pr(w_2|w_1) \cdots \Pr(w_n|w_1, \dots, w_{n-1}) \\ &= \prod_{i=1}^n \Pr(w_i|w_1, \dots, w_{i-1}) \end{aligned} \quad (1.96)$$

Eq. (1.96) describes a procedure that generates a word sequence from left to right (call it **auto-regressive** generation). Estimating  $\Pr(w_i|w_1, \dots, w_{i-1})$  is essentially a missing word prediction problem: we mask out the last word of a sequence and guide the language model to predict the correct word at that position. See below for a word sequence where the last word is missing.

Pride and prejudice is one of the best known \_\_\_\_

We can reuse the idea in classification to model the probability distribution  $\Pr(\_ |$

*Pride, and, ..., known*). Let  $\mathbf{x}_i$  be the vector representation of  $w_i$ . We can define a function that reads  $\mathbf{x}_1 \dots \mathbf{x}_{i-1}$  and produces a vector  $\mathbf{h}_i$ :

$$\mathbf{h}_i = f(\mathbf{x}_1, \dots, \mathbf{x}_{i-1}) \quad (1.97)$$

where  $\mathbf{h}_i$  is the intermediate states of the word distribution at position  $i$ . For a sounding distribution, we normalize  $\mathbf{h}_i$  by some normalization function  $\psi(\cdot)$ . Thus, the distribution at position  $i$  would be

$$\begin{aligned} \mathbf{y}_i &= \psi(\mathbf{h}_i) \\ &= \psi(f(\mathbf{x}_1, \dots, \mathbf{x}_{i-1})) \end{aligned} \quad (1.98)$$

Obviously,  $y_i(w_i)$  is the probability of  $w_i$  given previous words, i.e.,  $y_i(w_i) = \Pr(w_i | w_1, \dots, w_{i-1})$ . Note that Eq. (1.96) only considers the left context when predicting a word. A natural extension to this is to condition the prediction on all available context. Consider, for example, a sentence with a masked word in the middle.

Pride and \_\_\_ is one of the best-known novels

In this example, we can predict the masked word by using both the left context (*Pride and*) and the right context (*is one of the best known novels*):

$$\mathbf{y}_i = \psi(f(\mathbf{x}_1, \dots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \dots, \mathbf{x}_n)) \quad (1.99)$$

This is a bidirectional model, and is commonly used in **auto-encoding** methods for learning sequence representation models [Devlin et al., 2019].

## 1.5.4 Sequence Generation

Sequence generation covers a range of NLP problems, including machine translation, summarization, question answering, dialogue systems, and so on. Usually, it refers to mapping some data to a sequence. Here we focus on the **sequence-to-sequence** problem, in that a source-side sequence is transformed to a target-side sequence, although sequence generation is not specialized to work with chain structures on the source-side.

For notation convenience, we use boldface variables to denote sequences from now on. For example,  $\mathbf{a}$  is a sequence of size  $n$ . It can be written as either  $[a_1 \dots a_n]$  or  $a_1 \dots a_n$ . Let  $\mathbf{s} = s_1 \dots s_m$  and  $\mathbf{t} = t_1 \dots t_n$  be the sequences to transform from and to. The sequence-to-sequence problem can be described as finding a target-side sequence that maximizes  $\Pr(\mathbf{t}|\mathbf{s})$ :

$$\hat{\mathbf{t}} = \underset{\mathbf{t}}{\operatorname{argmax}} \Pr(\mathbf{t}|\mathbf{s}) \quad (1.100)$$

Like language modeling,  $\Pr(\mathbf{t}|\mathbf{s})$  can be formalized in an auto-regressive fashion:

$$\begin{aligned}\Pr(\mathbf{t}|\mathbf{s}) &= \Pr(t_1, \dots, t_n|\mathbf{s}) \\ &= \Pr(t_1|\mathbf{s}) \cdot \Pr(t_2|\mathbf{s}, t_1) \cdots \Pr(t_n|\mathbf{s}, t_1, \dots, t_{n-1}) \\ &= \prod_{i=1}^n \Pr(t_i|\mathbf{s}, t_1, \dots, t_{i-1})\end{aligned}\tag{1.101}$$

Eq. (1.101) differs from Eq. (1.96) only in the additional condition (i.e.,  $\mathbf{s}$ ) introduced to these probabilities. In this sense, we can use Eqs. (1.97-1.98) to solve  $\Pr(t_i|\mathbf{s}, t_1, \dots, t_{i-1})$ . On the other hand, involving  $\mathbf{s}$  makes the problem more difficult, as we need to model the cross-sequence relationship between  $\mathbf{s}$  and  $t_i$ . A recent trend in sequence generation is to formulate  $\Pr(t_i|\mathbf{s}, t_1, \dots, t_{i-1})$  in the **encoder-decoder** paradigm. There are two steps: an encoder is first used to represent  $\mathbf{s}$  as some intermediate form (e.g., a vector), and a decoder is then used to model both the target-side words and the correlation between the encoder output and the target-side words. Putting these together, the output of the encoder-decoder model can be defined to be

$$\mathbf{y}_i = \text{Dec}(\text{Enc}(\mathbf{s}), t_1, \dots, t_{i-1})\tag{1.102}$$

where  $\text{Enc}(\cdot)$  is the encoder, and  $\text{Dec}(\cdot)$  is the decoder.  $\mathbf{y}_i$  is a distribution of the target-side word at position  $i$ , i.e.,  $y_i(t_i) = \Pr(t_i|\mathbf{s}, t_1, \dots, t_{i-1})$ . Chapter 5 will provide a detailed description of the encoder-decoder model.

### 1.5.5 Tree Generation

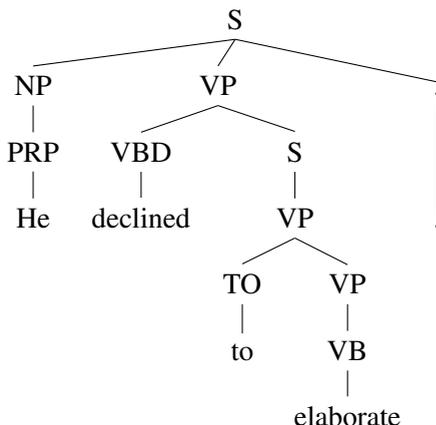
In NLP, trees are usually used to represent the structures or meanings of sequential data. For example, a **syntactic parser** analyzes a sentence to form a **syntax tree** or **parse tree**. More formally, given a sequence of words  $\mathbf{s} = s_1 \dots s_m$ , the parsing problem can be defined as:

$$\hat{d} = \underset{d \in D}{\text{arg max}} \Pr(d|\mathbf{s})\tag{1.103}$$

where  $d$  is a parse tree, and  $D$  is the set of all parse trees yielding  $s_1 \dots s_m$ . Computing  $\Pr(d|\mathbf{s})$  is challenging, as the modeling complexity increases exponentially when moving from sequences to trees. In **statistical parsing**, a solution is to model  $d$  as a derivation of syntactic rules. In this way,  $\Pr(d|\mathbf{s})$  can be formulated as a product of rule probabilities. Figure 1.10 presents an example of parsing with context-free grammar (CFG) rules. Alternatively,  $\Pr(d|\mathbf{s})$  can be modeled in an end-to-end manner. For example, some recent approaches perform parsing by defining a neural network over the parse tree. The probability of a sub-tree rooting at a node is computed by considering the interaction between this node and child nodes.

Another idea is to frame parsing as sequence generation. For example, one can linearize a parse tree and represent it as a sequence of words and syntactic labels, or transform the tree generation process as a sequence of actions. This allows the use of sequence-to-sequence techniques in addressing a sequence-to-tree problem.

Parse Tree:



CFG Rules:

- $r_1$  : PRP  $\rightarrow$  He
- $r_2$  : VBD  $\rightarrow$  declined
- $r_3$  : TO  $\rightarrow$  to
- $r_4$  : VB  $\rightarrow$  elaborate
- $r_5$  : .  $\rightarrow$  .
- $r_6$  : NP  $\rightarrow$  PRP
- $r_7$  : VP  $\rightarrow$  VB
- $r_8$  : VP  $\rightarrow$  TO VP
- $r_9$  : S  $\rightarrow$  VP
- $r_{10}$  : VP  $\rightarrow$  VBD S
- $r_{11}$  : S  $\rightarrow$  NP VP .

$$\begin{aligned}
 P(d|s) &= P(\text{PRP} \rightarrow \text{He}) \cdot P(\text{VBD} \rightarrow \text{declined}) \cdot P(\text{TO} \rightarrow \text{to}) \cdot P(\text{VB} \rightarrow \text{elaborate}) \cdot \\
 &\quad P(. \rightarrow .) \cdot P(\text{NP} \rightarrow \text{PRP}) \cdot P(\text{VP} \rightarrow \text{VB}) \cdot P(\text{VP} \rightarrow \text{TO VP}) \cdot P(\text{S} \rightarrow \text{VP}) \cdot \\
 &\quad P(\text{VP} \rightarrow \text{VBD S}) \cdot P(\text{S} \rightarrow \text{NP VP} .) \\
 &= \prod_{i=0}^{11} P(r_i)
 \end{aligned}$$

Figure 1.10: An example parse tree and CFG rules. The sentence is from the training data of the CoNLL 2000 shared task. The parse tree is represented as a derivation of CFG rules. The probability of the parse tree is defined as the product of rule probabilities.

In linguistics and NLP, tree structures are in heavy use for syntactic analysis. In addition to parsing sentences, they are also attributed to words, phrases, and discourses. On the other hand, trees are not the only way of visualizing complex non-linear structures. A more general concept is a graph. While trees can be thought of as special graphs, there are cases that trees cannot handle [Fellbaum, 2005; Singhal, 2005; Banarescu et al., 2013]. For example, in the semantic representation of a sentence, we often need a graph to connect verbs and arguments. While learning general graphs is harder than parsing a sentence into a tree, we can reuse many of the methods developed in sequence and tree generation.

## 1.5.6 Relevance Modeling

Generally speaking, relevance is referred to as how well a thing relates to another. The concept of relevance is used in many different sub-fields of NLP and information science. For example, in information retrieval, relevance is used to describe to what extent a retrieved document meets the query. Additional uses of this concept can be found in question answering, dialogue systems, semantic analysis, and all other tasks that require a matching or retrieval process.

Let us consider a more general description. Assume that we have a query *query* and a key *key* that represents something we intend to match with *query*. Then, we define the feature

vectors of *query* and *key* as

$$\mathbf{q} = Q(\text{query}) \quad (1.104)$$

$$\mathbf{k} = K(\text{key}) \quad (1.105)$$

$Q(\cdot)$  and  $K(\cdot)$  are feature extractors. The relevance between *query* and *key* is given by the function:

$$r = f(\mathbf{q}, \mathbf{k}) \quad (1.106)$$

$f(\cdot)$  could be on one hand simply a distance measure if  $\mathbf{q}$  and  $\mathbf{k}$  are in the same vector space, and on the other hand a more complex model that performs some non-linear transformations. In fact, the way of defining relevance can be adopted in several different scenarios. Sometimes, relevance is also termed as similarity or correlation. A general example is how similar two objects are. Let  $x$  and  $y$  be two samples (say two words). The similarity of  $x$  and  $y$  is given by

$$r = f(g(x), g(y)) \quad (1.107)$$

where  $g(\cdot)$  is a feature extractor, and  $f(\cdot)$  is a **similarity function**. Learning both  $g(\cdot)$  and  $f(\cdot)$  is called **similarity learning**. In one setup of similarity learning, we fix  $f(\cdot)$  and learn  $g(\cdot)$  in a way that similar samples exhibit similar outputs of  $g(\cdot)$ . The learning of the feature extractor is not even required to work with the similarity function. For example, for obtaining the similarity between words, we can learn  $g(\cdot)$  in a language model and use it together with various similarity functions. This puts the problem in a larger topic of machine learning: the learning of a sub-model is independent of the problem where we use it. Such an idea is widely adopted in pre-training, advancing the recent state-of-the-art on many NLP tasks.

In another setup of similarity learning, we can learn  $f(\cdot)$  directly. This can be performed by either jointly learning  $f(\cdot)$  and  $g(\cdot)$ , or learning  $f(\cdot)$  on top of fixed  $g(\cdot)$ . The problem is also related to **metric learning**. Typically, metric learning is framed as a supervised problem [Kulis, 2013]. A desired similarity function could be learned with the supervision regarding some gold-standard similarity. However, in practice there is usually no such supervised information in NLP. In this case, one could take relative distance as some supervision. For example, the similarity function can be learned by optimizing a contrastive loss (see Section 1.3.4).

Measuring the similarity between objects plays an important role in many machine learning methods, such as clustering and nearest neighbor classification. On the side of NLP, it is useful for exploring the relationship between words, phrases, sentences, and documents, e.g., similarity is a way to examine how word vectors correspond to our understanding of word meanings [Mikolov et al., 2013c; Pennington et al., 2014].

### 1.5.7 Linguistic Alignment

Linguistic alignment is a set of problems where we establish some correspondence between two sets of linguistic units. In NLP, the sequence-to-sequence and sequence-to-tree problems are typically linguistic alignment problems, as they both connect two linguistic units. However,

by convention, the term *alignment* is referred to as aligning multiple objects simultaneously<sup>23</sup>.

As an example, consider the well-known **word alignment** task: we align the words of a sentence to the words of another sentence. We reuse the notation in Section 1.5.4 as both the sequence-to-sequence and word alignment tasks perform on a pair of sequences. Given a source-side word sequence  $\mathbf{s} = s_1 \dots s_m$  and a target-side word sequence  $\mathbf{t} = t_1 \dots t_n$ , the word alignment between the two sequences is denoted as an  $m \times n$  matrix  $\mathbf{A}$ .  $A(i, j) = 1$  if there is an **alignment link** between  $s_i$  and  $t_j$ , and  $A(i, j) = 0$  otherwise. The optimal alignment can be defined as:

$$\hat{\mathbf{A}} = \underset{\mathbf{A}}{\operatorname{argmax}} \operatorname{Pr}(\mathbf{A}|\mathbf{s}, \mathbf{t}) \quad (1.108)$$

where  $\operatorname{Pr}(\mathbf{A}|\mathbf{s}, \mathbf{t})$  is the word alignment probability. Like in other machine learning problems, we can model  $\operatorname{Pr}(\mathbf{A}|\mathbf{s}, \mathbf{t})$  in either a generative or discriminative manner (see Section 1.2.4). For example, in Brown et al. [1993]’s work, the word alignment model is factored into several generative steps, each accounting for some assumptions about the problem<sup>24</sup>.

A 0-1 alignment matrix indicates a hard way of word alignment. A problem here is that the hard model may not describe well the highly ambiguous word alignments. We therefore can represent  $\mathbf{A}$  as a real-valued matrix (call it a **soft word alignment matrix** or a **word alignment weight matrix**). Assume that the source-side words are represented as a sequence of feature vectors  $\mathbf{x} = [\mathbf{x}_1 \ \dots \ \mathbf{x}_m]$ . Likewise, the target-side words are represented as  $\mathbf{y} = [\mathbf{y}_1 \ \dots \ \mathbf{y}_n]$ . A soft word alignment model is given by:

$$\mathbf{A} = a(\mathbf{s}, \mathbf{t}) \quad (1.109)$$

where  $a(\cdot)$  is an word alignment function that computes the alignment weight  $A(i, j)$  for each pair of  $\mathbf{x}_i$  and  $\mathbf{y}_j$ . In fact, all the methods discussed in Section 1.5.6 are applicable to the design of  $a(\cdot)$ . This somehow links the modeling of word alignment with the modeling of similarity, and makes it possible to address different NLP problems by using the same machine learning approach.

Eq. (1.109) offers a very general way to discover the underlying connection over pairs of variables. In addition to aligning words in sequences, it is useful for aligning unordered objects. For example, in bilingual dictionary induction, we can learn such a weight matrix to estimate how strong a word in one language corresponds to a word in another language.

Here is another note on linguistic alignment models. While linguistic alignment could be thought of as an independent NLP task, it is commonly used in designing sub-models of some downstream systems. Many systems that model word-level relationships involve implicit representation of linguistic alignment. As a consequence, linguistic alignment is treated as some latent states, and is a by-product of these systems. For example, in the early

<sup>23</sup>The concept of alignment is wide-ranging. We use the term *linguistic alignment* here to differentiate it from the *alignment* of large language models discussed in subsequent chapters.

<sup>24</sup>More precisely, Brown et al. [1993] model  $\operatorname{Pr}(\mathbf{A}, \mathbf{s}|\mathbf{t})$  which is a surrogate of  $\operatorname{Pr}(\mathbf{A}|\mathbf{s}, \mathbf{t})$ , as  $\operatorname{Pr}(\mathbf{A}|\mathbf{s}, \mathbf{t}) = \frac{\operatorname{Pr}(\mathbf{A}, \mathbf{s}|\mathbf{t})}{\operatorname{Pr}(\mathbf{s}|\mathbf{t})} = \frac{\operatorname{Pr}(\mathbf{A}, \mathbf{s}|\mathbf{t})}{\sum_{\mathbf{A}'} \operatorname{Pr}(\mathbf{A}', \mathbf{s}|\mathbf{t})}$

age of statistical machine translation, word alignment is a hidden variable used in modeling the mapping between sequences. The word alignment result can be easily induced from a machine translation model. More recently, neural sequence-to-sequence models — most notably attentional models [Bahdanau et al., 2014] — have attempted to do something similar to word alignment by computing attention weights among words.

### 1.5.8 Extraction

In NLP, *extraction* is not a kind of task but a kind of behavior that a system exhibits. Informally, it denotes a process of gathering, distilling structured information from some information sources. So, the term extraction generally appears together with other terms to form a specific task, such as **keyword extraction**, **event extraction**, and **relation extraction**. Many of these tasks can be categorized into an area — **information extraction**. Information extraction is perhaps the broadest topic in NLP. There is even no exhaustive list of information extraction tasks. According to Jurafsky and Martin [2008]’s book, it includes but is not limited to named entity recognition, reference resolution, relation extraction, event extraction, **template filling**, and so on.

However, since information extraction is a “miscellany” of many different problems, it cannot be formulated as a single machine learning problem. Fortunately, most of these problems can be framed as standard machine learning problems, such as classification and sequence labeling, and can be solved by using the off-the-shelf tools. In some cases, it may require a slight update of existing models for adaptation to a new task. For example, extracting a specific segment from text may require the system to produce a span that indicates the beginning and ending positions of the extracted segment (call it **span prediction**).

On the practical side, machine learning is not always necessary in extracting information from text. Many problems can be solved by using hand-crafted rules. An example is using regular expressions to identify locations and dates in text. In practice real-world systems are usually combinations of heuristic methods and automatic machine learning methods.

### 1.5.9 Others

Figure 1.11 shows illustrative examples of the above NLP tasks. Note that many of the discussions here are still preliminary and incomplete. For example, we only talked about NLP problems in the supervised learning paradigm. Many unsupervised tasks are important for NLP research as well. For example, it is common to cluster unlabeled words or documents to ease the processing in downstream systems. Several methods are directly applicable to this task [Murphy, 2012]. A recent trend in NLP is that it is not necessary to set a strict boundary between the use of supervised learning and the use of unsupervised learning. In many cases, unsupervised methods help supervised tasks, and vice versa. A notable example is that we learn a pre-trained feature extractor on unlabeled data and build a supervised classifier on top of it. This leads to another trend running towards improving representation models (i.e., feature extractors) without the need of accessing downstream supervised tasks.

## 1.6 Summary

This chapter has given the basic ideas of machine learning and its applications to NLP problems. In particular, we have presented a simple text classification problem to get started with machine learning. Also, we have discussed several general problems on machine learning, e.g., types of machine learning methods, inductive biases, loss functions, overfitting and so on. They are followed by a discussion on model selection and assessment. In addition, we have described how model NLP problems are framed as machine learning problems.

However, machine learning is a huge research field. There are several interesting topics we left out. One topic that we said little about is reinforcement learning. In general, reinforcement learning is very powerful. It should be and has been considered as an approach to addressing NLP problems, e.g., training a sequence-to-sequence by using a risk-based loss function. A reinforcement learning textbook will offer the general ideas of reinforcement learning [Sutton and Barto, 2018]. Another topic we missed here is Bayesian learning [Gelman et al., 2020; McElreath, 2020; Downey, 2021]. It opens up a notable strand of research in statistical learning, and has been successfully used in NLP tasks. Moreover, there are many other topics that are specialized in certain aspects of machine learning and are of interest to NLP researchers and engineers. Some of them are efficient machine learning [Tay et al., 2020b], multi-task learning [Zhang and Yang, 2021], few-shot/zero-shot learning [Wang et al., 2019b; 2020c], and so on.

A final point to wrap up this chapter. We skip the detailed discussion on certain machine learning models and algorithms, such as classification and regression models, because the reader interested in them can find several excellent, comprehensive introductions [Bishop, 2006; Hastie et al., 2009; Murphy, 2012; Mohri et al., 2018]. In the next chapter we will discuss a bit more about artificial neural networks which are the basis of deep learning and recent state-of-the-art NLP models.

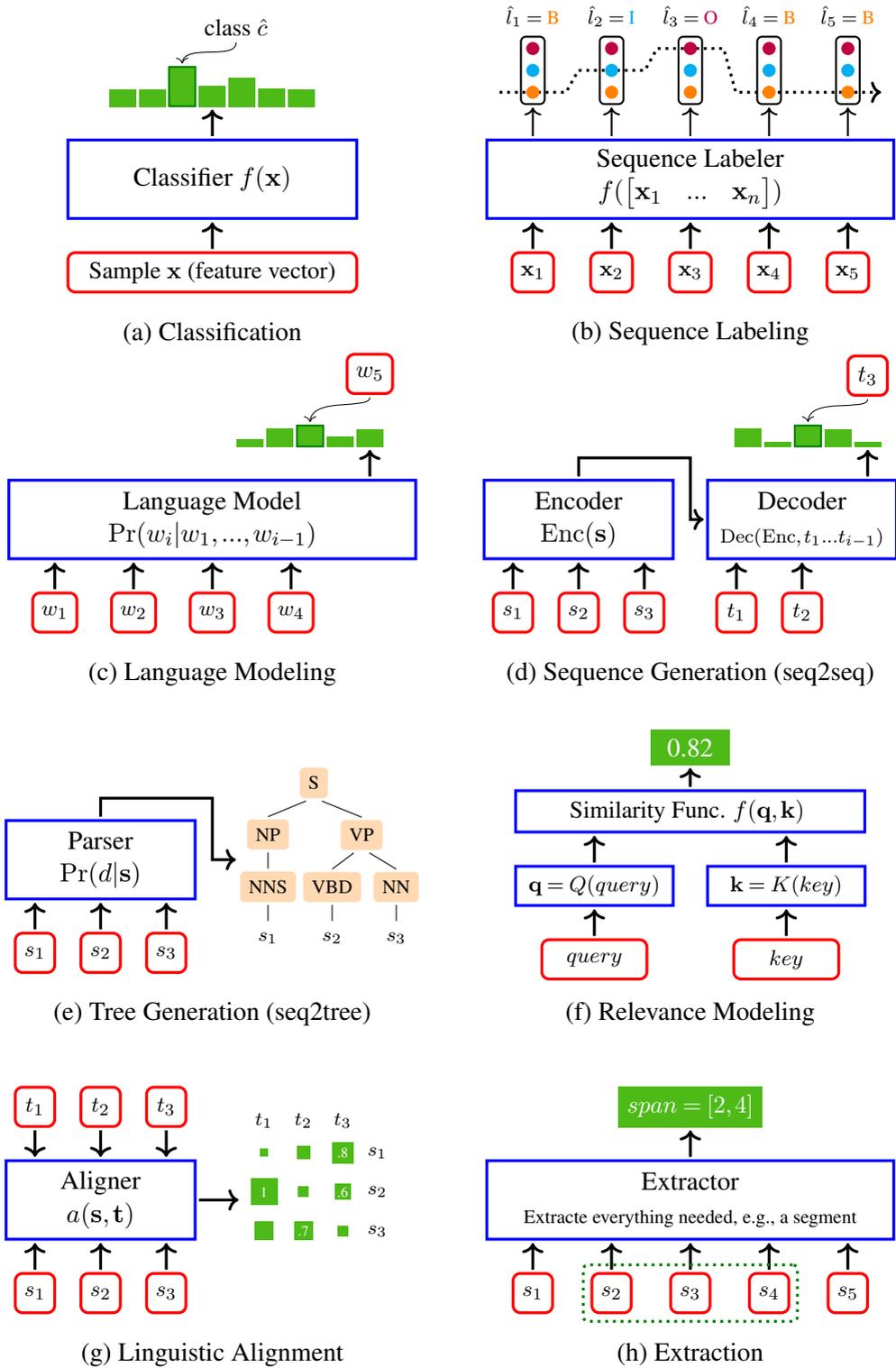


Figure 1.11: Natural language processing tasks from a machine learning perspective.



# Chapter 2

## Foundations of Neural Networks

**Artificial neural networks** (or **neural networks**, or **neural nets** for short) are powerful machine learning tools that have advanced the previous state-of-the-art in NLP in recent years. However, although the history of neural networks can be traced back to the 1940s [McCulloch and Pitts, 1943], for quite a long time neural networks have not been found to consistently outperform other machine learning counterparts. The change began around 2006 when “new” ideas were developed to learn **deep neural networks** [Hinton et al., 2006; Hinton, 2007]. Such methods have since been known as **deep learning**. To date, deep learning has no doubt become one of the most active, influential areas in artificial intelligence, while it has received benefits from not only “deep” model architectures but also many, many techniques which help to learn and use such models.

In this chapter, we will present the basic ideas of neural networks and deep learning. The chapter is not cutting-edge but covers several important concepts and techniques that are widely used in implementing neural systems. This includes basic model architectures of neural networks, training and regularization methods, unsupervised learning methods, and auto-encoders. We will also present an example of using neural networks to solve the language modeling problem.

### 2.1 Multi-layer Neural Networks

To get started, we give a quick introduction to **single-layer perceptrons**, and extend them to a more general case where multiple neural networks are stacked to form a more complex one.

#### 2.1.1 Single-layer Perceptrons

Single-layer perceptrons (or **perceptrons** for short) may be the simplest neural networks that have been developed for practical uses [Rosenblatt, 1957; Minsky and Papert, 1969]. Often, it is thought of as a biologically-inspired program that transforms some input to some output. A perceptron comprises a number of **neurons** connecting with input and output variables. Figure 2.1 shows a perceptron where there is only one neuron. In this example, there are two real-valued variables  $x_1$  and  $x_2$  for input and a binary variable  $y$  for output. The neuron reads

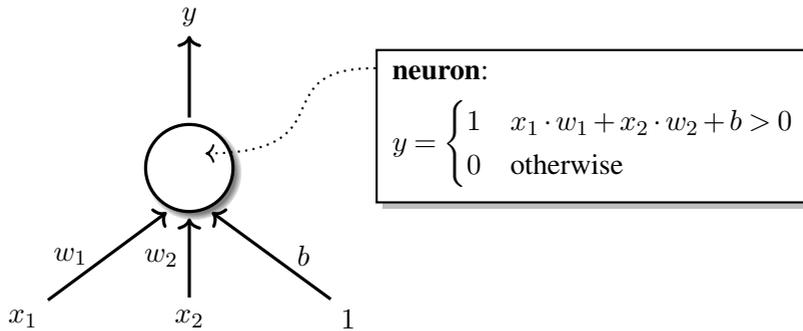


Figure 2.1: A perceptron with two input variables  $\{x_1, x_2\}$  and an output variable  $y$ . There are two weights  $\{w_1, w_2\}$ , each corresponding to an input variable. The output depends on the sum of the weighted input variables and the bias term  $b$ , say,  $y = 1$  if  $x_1 \cdot w_1 + x_2 \cdot w_2 + b > 0$ , and  $y = 0$  otherwise.

the input variables and determines which output value is chosen. This procedure is like what a biological neuron does — it receives electrochemical inputs from other neurons and determines if the electrochemical signal is passed along.

In a mathematical sense, a perceptron can be described as a mapping function. Let  $\mathbf{x}$  be a vector of input variables (i.e., a **feature vector**). An **affine transformation** of  $\mathbf{x}$  is given by<sup>1</sup>:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{x} \cdot \mathbf{w} + b \\ &= \sum_i x_i \cdot w_i + b \end{aligned} \quad (2.1)$$

where  $\mathbf{w}$  is a weight vector and  $b$  is a bias term. Then, a standard perceptron can be defined to be:

$$\begin{aligned} y &= \psi(f(\mathbf{x})) \\ &= \begin{cases} 1 & f(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (2.2)$$

where  $\psi(\cdot)$  is a binary **step function**. Another name for  $\psi(\cdot)$  is activation function. This links the perceptron to the classification models discussed in Section 1. In other words, Eq. (2.2) is a classifier itself:  $\psi(\cdot)$  is a discriminate function defined on each input  $\mathbf{x}$ , followed by an activation function  $\psi(\cdot)$  used for producing a desirable output<sup>2</sup>.

In case there are two or more neurons, we can group these neurons into a **layer**. As shown in Figure 2.2, all the neurons in a layer receive signals from the same input feature vector but are weighted in different ways. The output of the layer is a new feature vector, each entry of

<sup>1</sup>In mathematics, a **linear transformation** maps each vector  $\mathbf{v}$  in a space to  $f(\mathbf{v})$  in another space, satisfying for any vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and scalars  $\alpha$  and  $\beta$ , we have  $f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$ . An affine transformation is a linear transformation followed by a translation, often written in the form  $f(\mathbf{x}) + \mathbf{b}$ .

<sup>2</sup>Since the step function is a linear combination of indicator functions, the perceptron is a linear classifier.

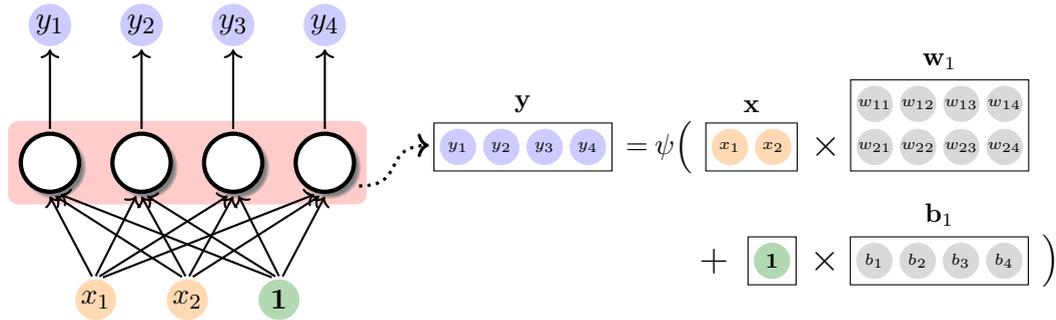


Figure 2.2: A single-layer perceptron involving four neurons. All these neurons receive information from the input variables  $\{x_1, x_2\}$ . The perceptron describes a process in that 1) we first transform the input vector of variables by an affine transformation  $f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w} + \mathbf{b}$ ; 2) and then compute the output by feeding  $f(\mathbf{x})$  into the activation function  $\psi(\cdot)$ .

which corresponds to a neuron. More formally, taking  $\psi(\cdot)$  and  $f(\cdot)$  as vector functions, the mathematical form of the single-layer perceptron is given by the equations:

$$\mathbf{y} = \psi(f(\mathbf{x})) \quad (2.3)$$

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w} + \mathbf{b} \quad (2.4)$$

where  $\mathbf{x} \in \mathbb{R}^m$ ,  $\mathbf{y} \in \mathbb{R}^n$ ,  $\mathbf{w} \in \mathbb{R}^{m \times n}$  and  $\mathbf{b} \in \mathbb{R}^n$ .

Another note on the activation function. The step function, though extensively used, is not the only form of the activation function. There are many different ways to perform activation. For example, we can use the Softmax function if we want a probability distribution-like output; we can use the Sigmoid function if we want a monotonic, continuous, easy-to-optimize output; we can use the ReLU function if we want a ramp-shaped output. Table 2.1 shows several commonly used activation functions. Note that, although a layer of neurons equipped with these activations can be loosely called a single-layer perceptron, it can be categorized as a more general concept, called a **single-layer neural network**. If not specified otherwise, we will use the term *single-layer neural network* throughout this document.

## 2.1.2 Stacking Multiple Layers

A next obvious step is to create a neural network comprising multiple layers. To do this, all we need is to stack multiple single-layer neural networks to form a **multi-layer neural network**. See Figure 2.3 for an example. In this multi-layer neural network, the output of every neuron of a layer is connected to all neurons of the following layer. So the network is **fully connected**. Essentially, a multi-layer neural network describes a composition of functions. For example, we can formulate the neural network in Figure 2.3 as a function yielded by composing a few simple functions:

$$\mathbf{y} = \text{Softmax}(\text{Sigmoid}(\text{ReLU}(\mathbf{x} \cdot \mathbf{w}_1) \cdot \mathbf{w}_2) \cdot \mathbf{w}_3 + \mathbf{b}_3) \quad (2.5)$$

Name	Formula (for entry $i$ of a vector)
Identity	$y_i = s_i$
Binary Step	$y_i = \begin{cases} 1 & s_i > 0 \\ 0 & s_i \leq 0 \end{cases}$
Hyperbolic Tangent	$y_i = \frac{\exp(s_i) - \exp(-s_i)}{\exp(s_i) + \exp(-s_i)}$
Hard Tangent	$y_i = \begin{cases} 1 & s_i > 1 \\ s_i & -1 \leq s_i \leq 1 \\ -1 & s_i < -1 \end{cases}$
Sigmoid (Logistic)	$y_i = \frac{1}{1 + \exp(-s_i)}$
ReLU (Rectified Linear Unit)	$y_i = \begin{cases} s_i & s_i > 0 \\ 0 & s_i \leq 0 \end{cases}$
Softplus	$y_i = \ln(1 + \exp(s_i))$
Gaussian	$y_i = \exp\left(-\frac{1}{2} \cdot \frac{(s_i - \mu_i)^2}{\sigma_i^2}\right)$
Softmax	$y_i = \frac{\exp(s_i)}{\sum_{i'=1}^n \exp(s_{i'})}$
Maxout	$y_i = \max(s_1, \dots, s_n)$

Table 2.1: Activation functions ( $\mathbf{y} = \psi(\mathbf{s})$ , where  $\mathbf{s}, \mathbf{y} \in \mathbb{R}^n$ ). All these functions are vector functions. We show formulas for entry  $i$  of the input and output vectors.  $\mu_i$  and  $\sigma_i^2$  are the mean and variance respectively.

where  $\mathbf{w}_1 \in \mathbb{R}^{3 \times 4}$ ,  $\mathbf{w}_2 \in \mathbb{R}^{4 \times 3}$ ,  $\mathbf{w}_3 \in \mathbb{R}^{3 \times 3}$ , and  $\mathbf{b}_3 \in \mathbb{R}^3$  are parameters.

Usually, the **depth** of a neural network is measured in terms of the number of layers. It is called **model depth** sometimes. For example, taking the input vector as an additional layer, the depth of the example network in Figure 2.3 is 4. A related concept is **model width**, which is typically defined on a layer, rather than on the entire network. A common measure for the width of a layer is the number of neurons in the layer. For example, the width of the output layer in Figure 2.3 is 3. If all layers of a neural network are of the same width  $n$ , then we can simply say that the model width is  $n$ . Both model depth and model width have important implications for the properties of the resulting neural network. For example, it has been proven that even a neural network with two layers of neurons and the Sigmoid activation function can compute any function [Cybenko, 1989]. For stronger systems, promising improvements are generally favorable when deepening neural networks.

Stacking layers results in a very common kind of neural network — **feed-forward neural networks (FFNNs)**. These networks are called “feed-forward” because there are no cycles in connections between layers and all the data moves in one direction. We will see in this book that most of today’s neural networks are feed-forward. A few exceptions will be presented in Section 2.3.

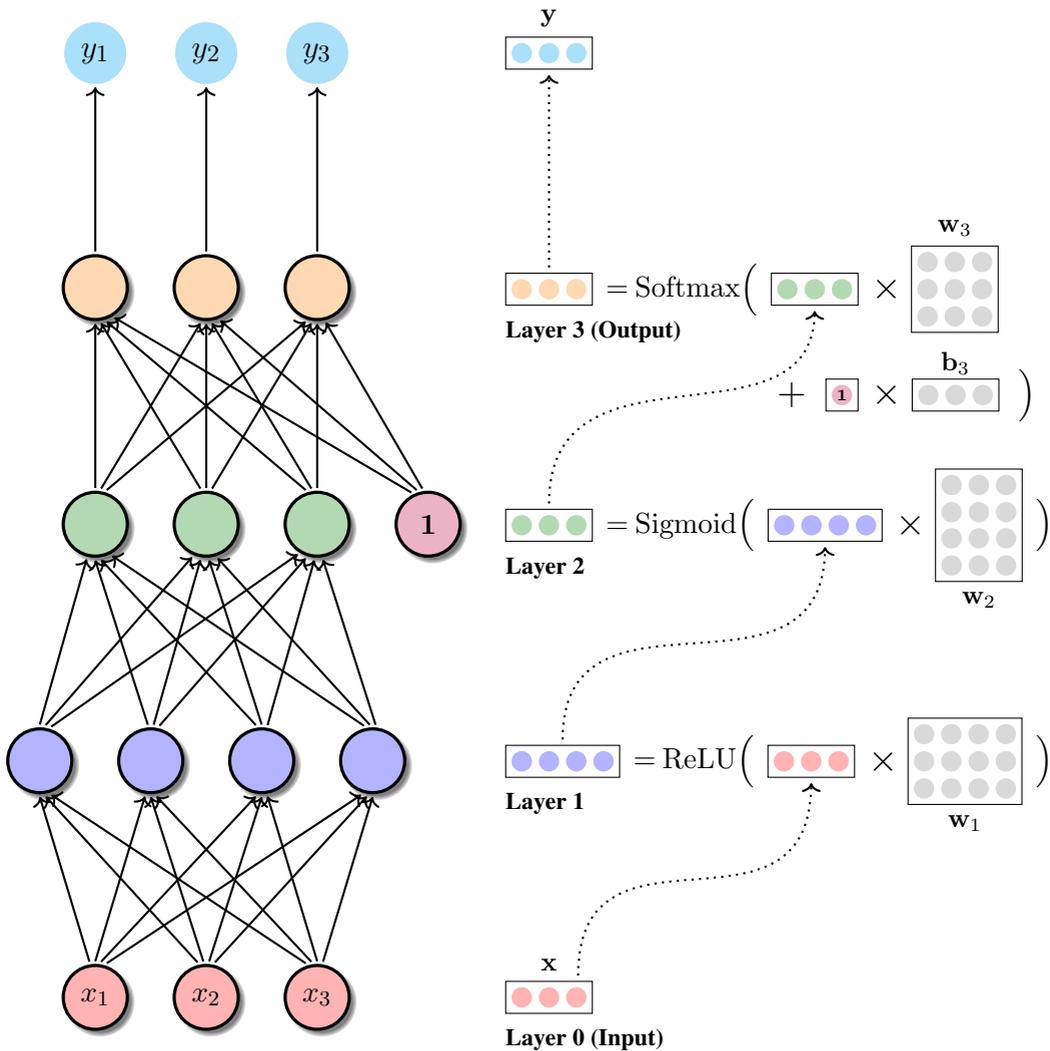


Figure 2.3: A multi-layer neural network. The input layer consists of three variables  $\{x_1, x_2, x_3\}$ . These variables are fully connected to all neurons of layer 1. The output of layer 1 is a new vector  $\mathbf{h}_1 = \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_1)$ . It is then fully connected to layer 2, performing the mapping  $\mathbf{h}_2 = \text{Sigmoid}(\mathbf{h}_1 \cdot \mathbf{w}_2)$ . Its output  $\mathbf{h}_2$  is fed into layer 3, which generates the final output  $\mathbf{y} = \text{Softmax}(\mathbf{h}_2 \cdot \mathbf{w}_3 + \mathbf{b}_3)$ . The parameters of this neural network are  $\mathbf{w}_1$ ,  $\mathbf{w}_2$ ,  $\mathbf{w}_3$  and  $\mathbf{b}_3$ .

### 2.1.3 Computation Graphs

**Computation graphs** are a common way of representing neural networks. As graphs in mathematics, a computation graph is made up of nodes and edges between nodes. Each node represents either a mathematical operation or a variable, and each edge represents the data flow from one node to another. So computation graphs are directed<sup>3</sup>. Consider, for example, three

<sup>3</sup>While a number of machine learning models can be represented as undirected computation graphs, they are not the focus of this document.

functions:

$$\mathbf{y} = \mathbf{x} + \mathbf{w} \quad (2.6)$$

$$\mathbf{y} = \text{Softmax}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}) \quad (2.7)$$

$$\mathbf{y} = \text{Sigmoid}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) - \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_2) \quad (2.8)$$

Figure 2.4 shows the computation graphs of these functions. From the parsing point of view, all neural networks can be viewed as mathematical expressions. A computation graph is therefore the representation of the result when parsing a mathematical expression. In this way, each node of the graph yields a sub-expression, and the root node yields the whole expression.

In a computation graph, a node can be connected to multiple nodes beneath it and/or above it. This enables the reuse of sub-graphs in representing complex functions. For example, in Eq. (2.8), the variable  $\mathbf{x}$  is used twice and the corresponding node has two outgoing edges. In fact, organizing neural networks into computation graphs resembles the compositional nature of neural networks — typically, a large network is built by composing small networks. Take Eq. (2.8) as an instance. It can be rewritten as a system of three equations:

$$\mathbf{y} = \mathbf{h}_1 - \mathbf{h}_2 \quad (2.9)$$

$$\mathbf{h}_1 = \text{Sigmoid}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) \quad (2.10)$$

$$\mathbf{h}_2 = \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_2) \quad (2.11)$$

In the composition operation, the nodes of  $\mathbf{h}_1$  and  $\mathbf{h}_2$  in Eq (2.9) are replaced by the graphs of Eqs. (2.10-2.11).

The main use of computation graphs is in executing the function. This is exactly the same thing as predicting the output of a neural network. The method is quite simple. First, the nodes of the graph are topologically sorted such that they are placed in an order consistent with the information flow. Then, given the values that are fed into the input nodes, the graph is traversed in a way that we compute the output of each node and flush it to its parent nodes. The final result is got out of the output node. This procedure is typically called a **forward pass**. A forward pass can be efficient, as every node only needs to be visited once and its output can be reused by multiple nodes without the need of recomputing the result. Moreover, a forward pass can be optimized by reconstructing the graph. This can develop the reuse idea a bit more and avoid unnecessary computation and memory consumption.

Another use of computation graphs is to compute gradients automatically. In training neural networks, it is in general required the partial derivatives of the loss function  $L$  with respect to every weight matrix ( $\mathbf{w}$ ) and every bias term ( $\mathbf{b}$ ), say  $\frac{\partial L}{\partial \mathbf{w}}$  and  $\frac{\partial L}{\partial \mathbf{b}}$ . Before seeing how these partial derivatives are used in updating a model (see Section 2.4.1), though, we first give an idea of computing derivatives in a computation graph. For example, consider the function below:

$$\mathbf{y} = \psi(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) \cdot \mathbf{w}_2 \quad (2.12)$$

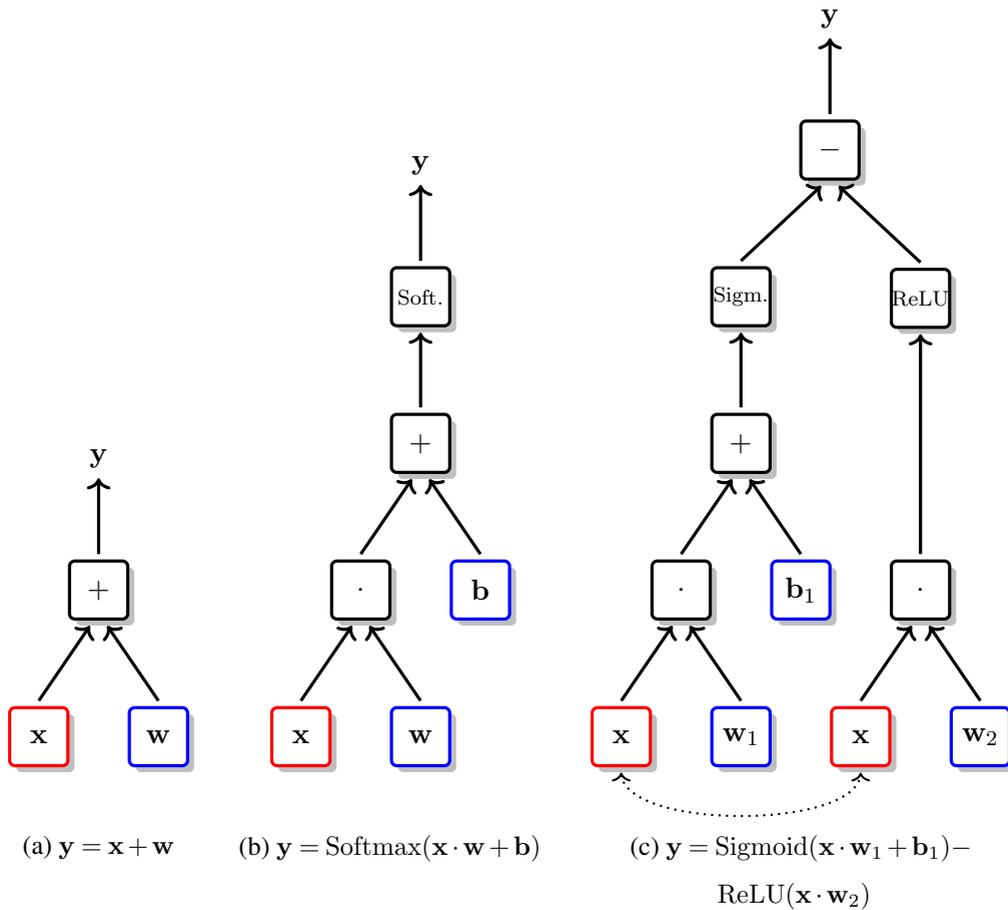


Figure 2.4: Computation graphs of three example neural networks. The black boxes represent the mathematical operations, and the colored boxes represent the variables. A mathematical operation node has incoming edges from other nodes, and each of these nodes can be treated as an argument of the operation. For example, in sub-figure (a), the addition node has two child nodes labeled with  $x$  and  $w$  respectively. This node reads the output of the nodes  $x$  and  $w$ , and generates the output  $y = x + w$ . Things are a bit interesting for larger graphs. In sub-graph (b), the output of the dot node (i.e.,  $x \cdot w$ ) is passed along the edge to the addition node. Then, the addition node computes the sum of  $x \cdot w$  and  $b$  as its output. We can repeat the same process over all the mathematical operation nodes in a bottom-up manner, and get the final result of computing the whole expression out of the top-most node.

To obtain  $\frac{\partial L}{\partial w_1}$ ,  $\frac{\partial L}{\partial b_1}$  and  $\frac{\partial L}{\partial w_2}$ , it is natural to use the **chain rule of differentiation**. For example, for a composite function  $y = p(q(x))$ , the formula of the chain rule is given as:

$$\frac{\partial y}{\partial x} = \frac{\partial p}{\partial q} \cdot \frac{\partial q}{\partial x} \quad (2.13)$$

But the analytic formula of a derivative based on Eq. (2.13) would make a lengthy equation.

Instead, we can decompose a complex function into several functions, each standing for some operation. Then, Eq. (2.12) can be rewritten as:

$$\mathbf{y} = \mathbf{h}_1 \cdot \mathbf{w}_2 \quad (2.14)$$

$$\mathbf{h}_1 = \psi(\mathbf{h}_2) \quad (2.15)$$

$$\mathbf{h}_2 = \mathbf{h}_3 + \mathbf{b}_1 \quad (2.16)$$

$$\mathbf{h}_3 = \mathbf{x} \cdot \mathbf{w}_1 \quad (2.17)$$

All these variables can be understood in a better way from a computation graph: each variable is a node of the graph, and nodes are connected by algebraic operations and function compositions. Taking Eq. (2.13) and some basic knowledge of calculus, we compute the derivatives of the variables, like these:

$$\text{node 1:} \quad \frac{\partial L}{\partial \mathbf{y}} = \delta_{\mathbf{y}} \quad (2.18)$$

$$\text{node 2:} \quad \frac{\partial L}{\partial \mathbf{h}_1} = \frac{\partial L}{\partial \mathbf{y}} \cdot \mathbf{w}_2^T \quad (2.19)$$

$$\text{node 3:} \quad \frac{\partial L}{\partial \mathbf{w}_2} = \mathbf{h}_1^T \cdot \frac{\partial L}{\partial \mathbf{y}} \quad (2.20)$$

$$\text{node 4:} \quad \frac{\partial L}{\partial \mathbf{h}_2} = \frac{\partial L}{\partial \mathbf{h}_1} \odot \psi'(h) \quad (2.21)$$

$$\text{node 5:} \quad \frac{\partial L}{\partial \mathbf{h}_3} = \frac{\partial L}{\partial \mathbf{h}_2} \quad (2.22)$$

$$\text{node 6:} \quad \frac{\partial L}{\partial \mathbf{b}_1} = \frac{\partial L}{\partial \mathbf{h}_2} \quad (2.23)$$

$$\text{node 7:} \quad \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{h}_3} \cdot \mathbf{w}_1^T \quad (2.24)$$

$$\text{node 8:} \quad \frac{\partial L}{\partial \mathbf{w}_1} = \mathbf{x}^T \cdot \frac{\partial L}{\partial \mathbf{h}_3} \quad (2.25)$$

where  $\delta_{\mathbf{y}}$  is the derivative of the loss with respect to the model output.  $\delta_{\mathbf{y}}$  depends on the choice of the loss function, e.g., if we use the squared loss  $L = \frac{1}{2}(\mathbf{y} - \mathbf{y}_{\text{gold}})^2$ , where  $\mathbf{y}_{\text{gold}}$  is the benchmark, then  $\delta_{\mathbf{y}} = \mathbf{y} - \mathbf{y}_{\text{gold}}$ . The above process is essentially a **backward pass**, as the gradients are passed in a top-down fashion. Another name for this is **error-propagation**. It has been the de facto standard for training deep neural networks. For a better understanding of how forward and backward passes work, Figure 2.5 shows two running examples.

## 2.2 Example: Neural Language Modeling

Language modeling is a well-known NLP task that estimates a probability distribution over sequences of words. Given a sequence of  $m$  words  $w_1 \dots w_m$ , the probability  $\Pr(w_1, \dots, w_m)$  is

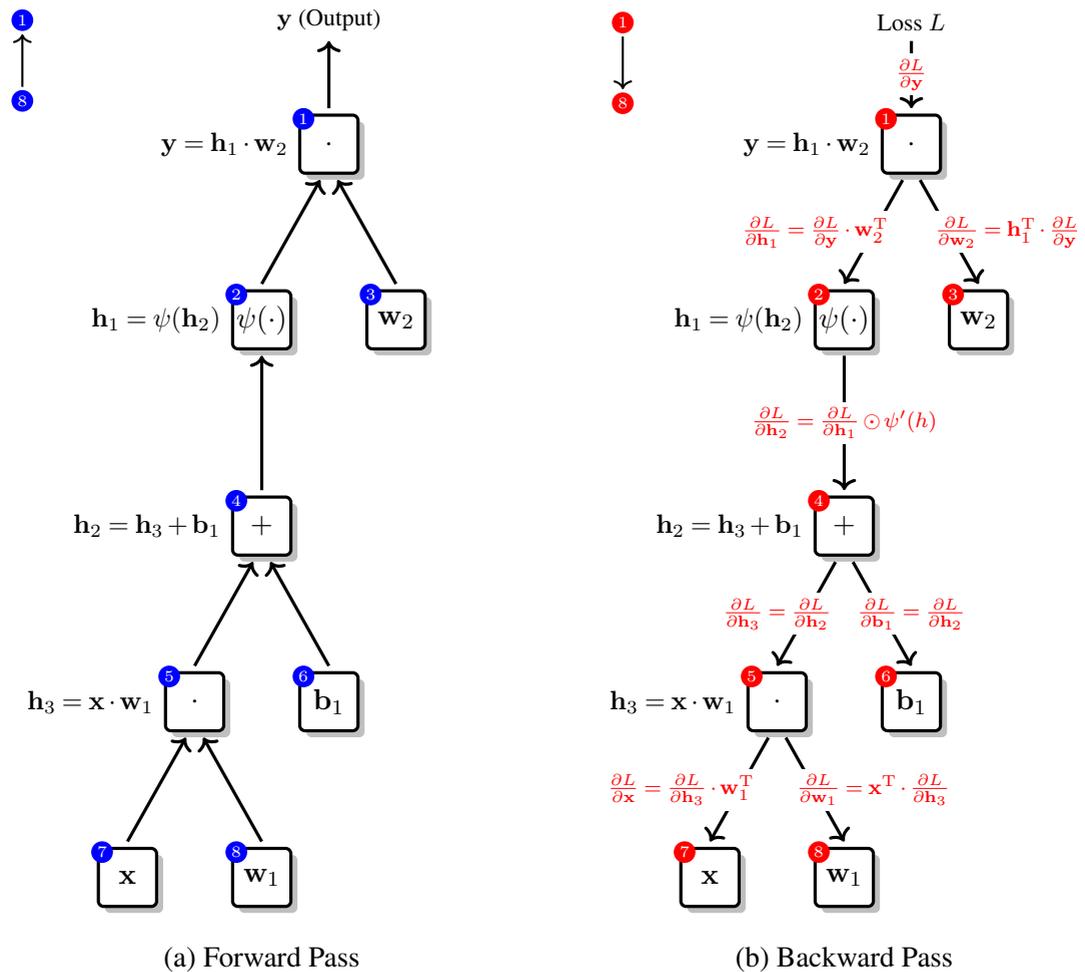


Figure 2.5: The forward pass and backward pass for an example computation graph. In the forward pass (left), the nodes are visited in an order from the input to the output, say, from node 8 to 1. On each node, we execute the corresponding function, such as addition, to generate the output, which is then consumed by the subsequent nodes. In contrast, in the backward pass (right), the nodes are visited in the reverse order, say, from node 1 to 8. During this process, we pass the gradient of the loss (or error) from the output to the input, that is, for each node, we compute the gradient at the input point of the node by using the chain rule, given the gradient at the output point of the node.

given by the equation:

$$\Pr(w_1, \dots, w_m) = \prod_{i=1}^m \Pr(w_i | w_1, \dots, w_{i-1}) \quad (2.26)$$

As such, the language modeling problem is framed as predicting the next word given all previous context words. A simple method of modeling  $\Pr(w_i | w_1, \dots, w_{i-1})$  is to condition the

prediction on a context window that covers at most a certain number of words, like this:

$$\Pr(w_i|w_1, \dots, w_{i-1}) \approx \Pr(w_i|w_{i-n+1}, \dots, w_{i-1}) \quad (2.27)$$

where  $n$  is the window size. One way to estimate the probability is the  **$n$ -gram language modeling** approach: we compute the relative frequency for each  $n$ -gram  $w_{i-n+1} \dots w_i$ , i.e.,  $\Pr(w_i|w_{i-n+1}, \dots, w_{i-1}) = \frac{\text{count}(w_{i-n+1} \dots w_i)}{\text{count}(w_{i-n+1} \dots w_{i-1})}$ . While  $n$ -gram language models have dominated the NLP field for a long time, they usually require huge tables for recording all those  $n$ -gram probabilities. In consequence, the models will be very sparse if more and more texts are used in training such models. This is also known as a kind of the curse of dimensionality.

Here we consider neural networks in addressing the language modeling problem [Bengio et al., 2000; 2003b]. Unlike  $n$ -gram language models, **neural language models** do not generalize in a discrete space that requires an exponentially large number of distinct feature vectors as more words and a large context are involved, but in a continuous space that encodes words via dense, low-dimensional real vectors. In particular, a feed-forward network is utilized here to predict how likely  $w_i$  occurs given  $w_{i-n+1} \dots w_{i-1}$ .

Figure 2.6 presents the architecture of the **feed-forward neural network based language model (FFNNLM)**. The input is the context words  $w_{i-n+1} \dots w_{i-1}$ . Each is a discrete variable choosing values from a vocabulary  $V$ . Since the neural network operates on vectors, all words are vectorized as **one-hot representations**. In this case, the word  $w = V_k$  is a  $|V|$ -dimensional vector in which entry  $k$  is 1 and other entries are all 0. For example, consider a vocabulary  $V = \{\text{“I”}, \text{“you”}, \text{“he”}, \text{“she”}, \text{“they”}\}$ . The one-hot representation of “you” is

$$w(\text{“you”}) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (2.28)$$

While the one-hot vectors make word representations distinguishable, it may not appear that we can gain too much by this because such representations cannot describe the closeness between words, e.g., similar words should tend to be close in the vector space. If we relax the indicator-based representations to real-valued representations, then it turns out that we can obtain some word relationship by computing similarities between these vectors. To this end, an effective technique is to transform one-hot representations to **distributed representations**. More formally, let  $\mathbf{x}$  be a one-hot vector of a word  $w$ . The distributed representation of the word is a real-valued vector, given by:

$$\mathbf{e} = \mathbf{x} \cdot \mathbf{C} \quad (2.29)$$

where the representation  $\mathbf{e}$  is a vector  $\in \mathbb{R}^{d_e}$ , and  $d_e$  is the number of dimensions of the representation. Each dimension of  $\mathbf{e}$  can be viewed as some countable aspect of the word, though it is not required to be interpreted by linguistics.  $\mathbf{C}$  is a  $|V| \times d_e$  matrix, of which the  $k$ -th row corresponds to the vector for  $V_k$ . Hence,  $\mathbf{w} \cdot \mathbf{C}$  is to “select” a row from  $\mathbf{C}$ . For

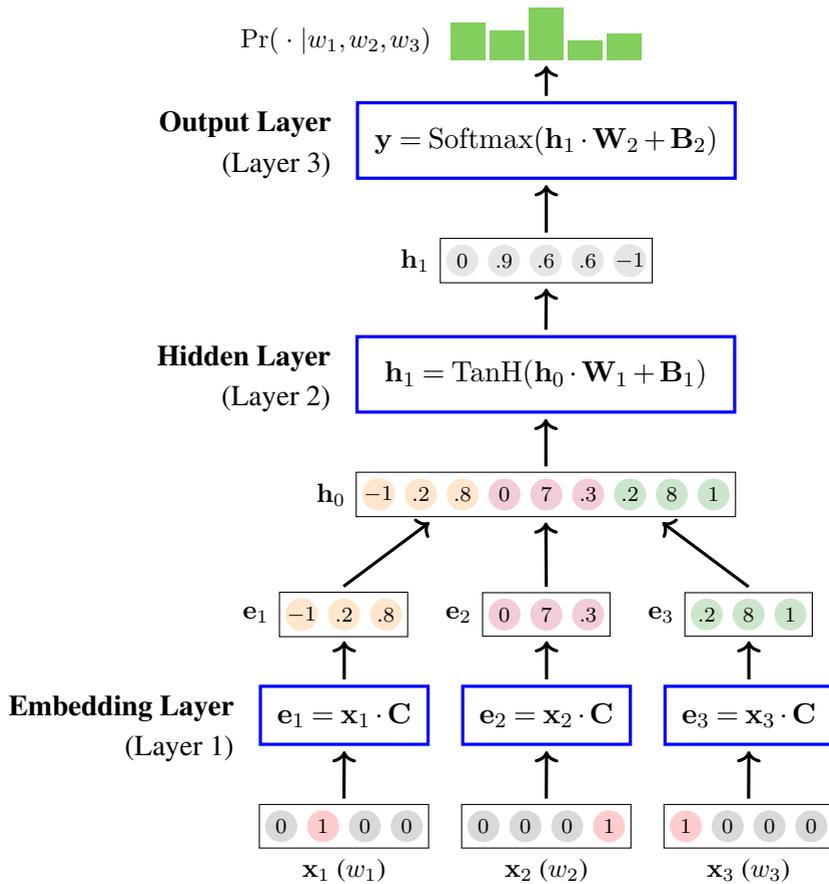


Figure 2.6: A neural language model [Bengio et al., 2003b]. Blue boxes represent the layers of the neural network. The input is three context words in their one-hot representations  $\{x_1, x_2, x_3\}$ , and the output is the probability distribution of the next word  $\Pr(w_4|w_1, w_2, w_3)$ . First, an embedding layer is used to map each word into the distributed representation (i.e., the word embedding). The embeddings of these words are concatenated to form a bigger vector  $h_0$  such that the concatenated vector encodes all input information. Then,  $h_0$  is taken as the input to a normal layer that performs the mapping  $h_1 = \text{TanH}(h_0 \cdot W_1 + B_1)$ . The final layer reads  $h_1$  and produces a distribution over the vocabulary, i.e.,  $y = \text{Softmax}(h_1 \cdot W_2 + B_2)$  where  $y_k = \Pr(V_k|w_1, w_2, w_3)$ .

example, given  $C \in \mathbb{R}^{5 \times 3}$ , the distributed representation of “you” is given by:

$$\begin{aligned}
 e(\text{“you”}) &= w(\text{“you”}) \cdot C \\
 &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 73 & 12 & 0.1 \\ 12 & 0.5 & 18 \\ 37 & 0.7 & 28 \\ 61 & 0.4 & 23 \\ 62 & 11 & 0.4 \end{bmatrix} \\
 &= \begin{bmatrix} 12 & 0.5 & 18 \end{bmatrix}
 \end{aligned} \tag{2.30}$$

Eq. (2.29) implies an idea of learning to represent words, leading to a big development of NLP. Typically, the vector  $\mathbf{e}$  is called the **word embedding**, and the parameter matrix  $\mathbf{C}$  is called the **embedding matrix**. A number of methods may be used for learning word embeddings, though we will tend to not focus on such methods in this chapter. The reader can refer to Chapter 3 for a more detailed discussion on this topic.

To encode the context words  $\{w_{i-n+1}, \dots, w_i\}$  (or  $\{\mathbf{x}_{i-n+1}, \dots, \mathbf{x}_i\}$ ), a simple method is to concatenate the word embeddings  $\{\mathbf{e}_{i-n+1}, \dots, \mathbf{e}_{i-1}\}$  as a new vector  $\mathbf{h}_0$ :

$$\mathbf{h}_0 = [\mathbf{e}_{i-n+1}, \dots, \mathbf{e}_{i-1}]$$

The next part of the model is a 2-layer feed-forward neural network. The first layer, called a **hidden layer**, is a standard layer of neurons, followed by the hyperbolic tangent activation function. The layer produces a  $d_h$ -dimensional vector:

$$\mathbf{h}_1 = \text{TanH}(\mathbf{h}_0 \cdot \mathbf{W}_1 + \mathbf{B}_1) \quad (2.31)$$

The second layer is the output layer. It produces a distribution over  $V$ . This can be formulated as:

$$\Pr(\cdot | w_{i-n+1}, \dots, w_{i-1}) = \text{Softmax}(\mathbf{h}_1 \cdot \mathbf{W}_2 + \mathbf{B}_2) \quad (2.32)$$

The parameters of the model are  $\mathbf{C} \in \mathbb{R}^{|V| \times d_e}$ ,  $\mathbf{W}_1 \in \mathbb{R}^{(n-1)d_e \times d_h}$ ,  $\mathbf{B}_1 \in \mathbb{R}^{d_h}$ ,  $\mathbf{W}_2 \in \mathbb{R}^{d_h \times |V|}$ , and  $\mathbf{B}_2 \in \mathbb{R}^{|V|}$ . A popular way to optimize these parameters is to minimize the cross-entropy loss via gradient descent. Additionally, training can be improved via regularization. These methods will be discussed in Sections 2.4 and 2.5.

A few remarks on the neural language model. First, by using distributed feature vectors, “senses” can be shared in part by different words. This enables learnable word senses by which the similarity between words is implicitly considered. An advantage of such a model is that a small change in word vectors would not lead to a big change in the result. For example, suppose we have seen “grapes are fruits” many times but have never seen “peaches are fruits”. If “grapes” and “peaches” are close in the vector space, then we would say that  $n$ -grams “grapes are fruits” and “peaches are fruits” are something similar. This differentiates neural language models greatly from  $n$ -gram language models in which different surface forms mean different meanings.

Second, the dense representation of words makes a smaller model. For example, a common setting of  $d_e$  and  $d_h$  is less than 1000, making the number of parameters under control. By contrast, the size of an  $n$ -gram language model increases by a factor of  $|V|$  as  $n$  increases. For example, there will be a huge table of probabilities for a common vocabulary if  $n$  is larger than 3.

Third, the neural language model is computationally expensive because of the heavy use of vector and matrix operations, such as matrix multiplication. This is a common problem with most of deep neural network-based systems. A common solution is to break the computation problem into independent sub-problems so that these sub-problems can be handled in parallel.

At a lower level, one can use GPUs or other parallel computing devices to speed up linear algebra operators. At a higher level, one can distribute parts of the model or parts of the data to multiple devices for model-level or task-level speed-ups.

## 2.3 Basic Model Architectures

We now describe, in more detail, several basic building blocks for neural networks. They are widely used in developing state-of-the-art neural models in NLP.

### 2.3.1 Recurrent Units

**Recurrent neural networks (RNNs)** are a class of neural networks that read and/or produce sequential data or time series data. As with a feed-forward neural network, an RNN comprises layers of neurons and connections between neurons [Hopfield, 1984; Rumelhart et al., 1986; Williams and Zipser, 1989; Elman, 1990]. Some of the neurons are used as a “memory” that keeps the state of the problem when the processing moves on along a sequence of signals. As a result, it is straightforward to use RNNs to deal with variable length problems, such as machine translation and speech recognition.

The main idea behind RNNs is to repeatedly utilize a **recurrent unit** (or **recurrent cell**) to compute the output at each position of an input sequence. To be more precise, given a sequence of vectors  $\mathbf{x}_1 \dots \mathbf{x}_m$ , a standard **recurrent unit** can be described as a function  $\text{RNN}(\cdot)$  that consumes an input  $\mathbf{x}_i$  and a state  $\mathbf{s}_{i-1}$  at each time and generates a new state  $\mathbf{s}_i$ , like this:

$$\mathbf{s}_i = \text{RNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) \quad (2.33)$$

The state  $\mathbf{s}_i$  can be viewed as a “memory” that summarizes the past data, and would be updated when the new data comes. See Figure 2.7 (a) for visualization of Eq. (2.33). The circle here indicates the reuse of the recurrent unit. This can be understood by rewriting Eq. (2.33) in a sequence of calls of the function  $\text{RNN}(\cdot)$ :

$$\begin{aligned} \text{RNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) &= \text{RNN}(\text{RNN}(\mathbf{s}_{i-2}, \mathbf{x}_{i-1}), \mathbf{x}_i) \\ &= \text{RNN}(\text{RNN}(\text{RNN}(\mathbf{s}_{i-3}, \mathbf{x}_{i-2}), \mathbf{x}_{i-1}), \mathbf{x}_i) \\ &= \dots \\ &= \text{RNN}(\text{RNN}(\dots (\text{RNN}(\mathbf{s}_0, \mathbf{x}_1), \mathbf{x}_2) \dots \mathbf{x}_{i-1}), \mathbf{x}_i) \end{aligned} \quad (2.34)$$

Figure 2.7 (b) shows the structure of this network. This is sometimes referred to as an **unrolled** (or **unfolded**) structure of RNNs. Basically, Figures 2.7 (a) and (b) are the same thing. While a rolled RNN has a simple and well-explained form, an unrolled RNN is more suitable for visualizing the data flow through the network. So, we will use the unrolled version of RNNs throughout this document. Moreover, it is worth noting that an unrolled RNN is in fact a deep feed-forward neural network. For example, each use of the recurrent unit creates a “layer” that receives information from a previous “layer”. In this sense, an RNN is a stack of

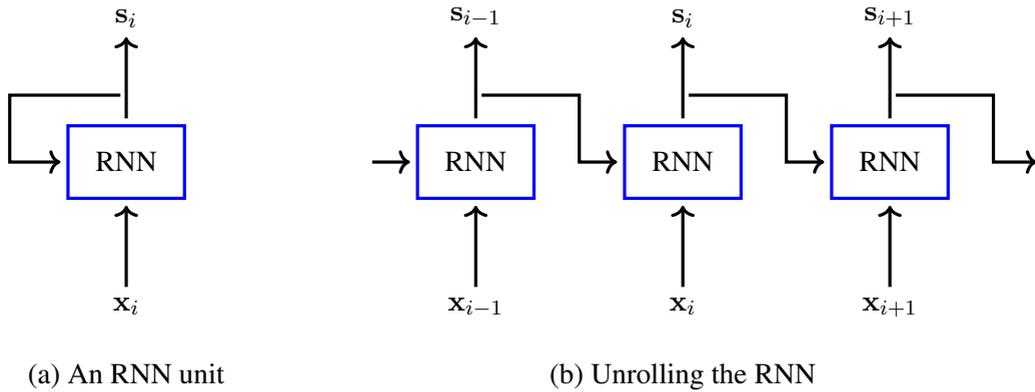


Figure 2.7: Example of RNN (rolled vs unrolled). An RNN unit reads the input at each time step  $i$  and the output at the last time step  $i - 1$ , and produces a new output  $s_i$ . As such we can reuse the same RNN unit to make predictions over a sequence of inputs (see sub-figure (a)): for each  $i$ , the current input  $\mathbf{x}_i$  and last output  $s_{i-1}$  are consumed and mapped to the output that is fed into the same RNN unit for the processing at the next time step. A better way to visualize the RNN is to unroll it into a network with no cycles (see sub-figure (b)). The unrolled RNN can be regarded as a deep feed-forward neural network in that all RNN units share the same set of parameters.

layers, say, stacking layers from left to right. A benefit of treating RNNs as deep feed-forward neural networks is that one can use the same methods to train and deploy the two types of neural networks. An example is that both RNNs and feed-forward neural networks can be trained by the error-propagation tool provided within a common optimizer.

There are a number of RNN variants, differing in ways of defining  $\text{RNN}(\cdot)$ . The simplest of these is to formulate  $\text{RNN}(\cdot)$  as a single-layer neural network. Assume that  $\mathbf{s}_{i-1}$  and  $\mathbf{x}_i$  are in  $\mathbb{R}^{d_h}$ . The form of  $\text{RNN}(\cdot)$  is given by:

$$\text{RNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) = \psi(\mathbf{s}_{i-1} \cdot \mathbf{U} + \mathbf{x}_i \cdot \mathbf{V}) \quad (2.35)$$

where  $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$  and  $\mathbf{V} \in \mathbb{R}^{d_h \times d_h}$  are parameters. The common choices for the activation function  $\psi$  are  $\text{TanH}(\cdot)$ ,  $\text{Sigmoid}(\cdot)$ ,  $\text{ReLU}(\cdot)$ , and among others. Eq. (2.35) is a single-layer neural network because it has the same form as Eqs. (2.3-2.4):

$$\psi(\mathbf{s}_{i-1} \cdot \mathbf{U} + \mathbf{x}_i \cdot \mathbf{V}) = \psi([\mathbf{s}_{i-1}, \mathbf{x}_i] \cdot \mathbf{W}) \quad (2.36)$$

where  $[\mathbf{s}_{i-1}, \mathbf{x}_i]$  is the concatenation of  $\mathbf{s}_{i-1}$  and  $\mathbf{x}_i$ , and  $\mathbf{W} \in \mathbb{R}^{2d_h \times d_h}$  is the parameter matrix that is formed by  $\begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix}$ .

RNNs often work as a part of a model. For example, the input of a recurrent unit could be either a representation of real data or an output of another neural network. Also, we can stack other neural networks on top of a recurrent unit. For example, in many real-world systems, an

additional layer is generally stacked on  $\mathbf{s}_i$  for projecting it to a desirable output.

### 2.3.2 Convolutional Units

**Convolutional neural networks (CNN)** are another well-known class of neural networks [Waibel et al., 1989; LeCun et al., 1989]. In a biological sense, they are inspired by human vision systems: neurons react to the stimulus in a certain vision region or patch (call it the **receptive field**) [Hubel and Wiesel, 1959]. In CNNs, the receptive field describes the region in the input space that is involved in generating the output for a neuron. CNNs are therefore “partially connected” models in which each neuron only considers input features in a restricted region. This differentiates CNNs from fully connected feed-forward neural networks. In general, CNNs can resemble the hierarchical nature of features describing data and scale better in complexity.

While CNNs have many applications in processing 2D data, such as image classification, we discuss them here in a sequential data processing scenario for a consistent treatment of the problem in this chapter. Typically, a CNN consists of a **convolutional layer**, a **pooling layer**, and other layers optionally. It begins with the convolutional layer where the receptive field is defined by a set of **convolution kernels** or **filters**. A filter is a linear mapping function that convolves the input features in the receptive field to form an output feature. For example, consider a sequence of numbers  $x_1 \dots x_m$ . The filter ranging from position  $i$  to position  $i + r - 1$  is defined to be:

$$\begin{aligned} v &= \text{Conv}(\mathbf{x}_{[i, i+r-1]}, \mathbf{W}) \\ &= \mathbf{x}_{[i, i+r-1]} \cdot \mathbf{W} \\ &= \sum_{k=0}^{r-1} x_{i+k} \cdot W_k \end{aligned} \quad (2.37)$$

where  $r$  is the size of the receptive field,  $\mathbf{x}_{[i, i+r-1]}$  is the sub-sequence  $x_i \dots x_{i+r-1}$ , and  $\mathbf{W} \in \mathbb{R}^r$  is the parameters of the filter. Then, a sequence of output features can be generated by moving the filter along the input sequence. Let *stride* be the distance between consecutive moves. The output for move  $i$  is then defined to be:

$$v_i = \text{Conv}(\mathbf{x}_{[\text{stride} \times i, \text{stride} \times i + r - 1]}, \mathbf{W}) \quad (2.38)$$

In this way, the convolutional layer transforms the input sequence  $x_1 \dots x_n$  to the output sequence  $v_1 \dots v_{\lfloor \frac{m}{\text{stride}} \rfloor}$ <sup>4</sup>. A remark here is that the parameters  $\mathbf{W}$  are shared across positions of the input sequence. This method is known as **parameter sharing** or **weight sharing**. Parameter sharing makes a CNN efficient because it requires fewer parameters than a feed-forward neural network given the same number of neurons.

A problem with the above formulation is that the use of the filter may not be tiled to fit the input sequence. For example, when  $\text{stride} \times i + r - 1 > m$ , the input of the filter is incomplete. A commonly used solution is **padding**. It simply sets the features outside the input sequence

<sup>4</sup> $\lfloor \cdot \rfloor$  stands for the floor function.

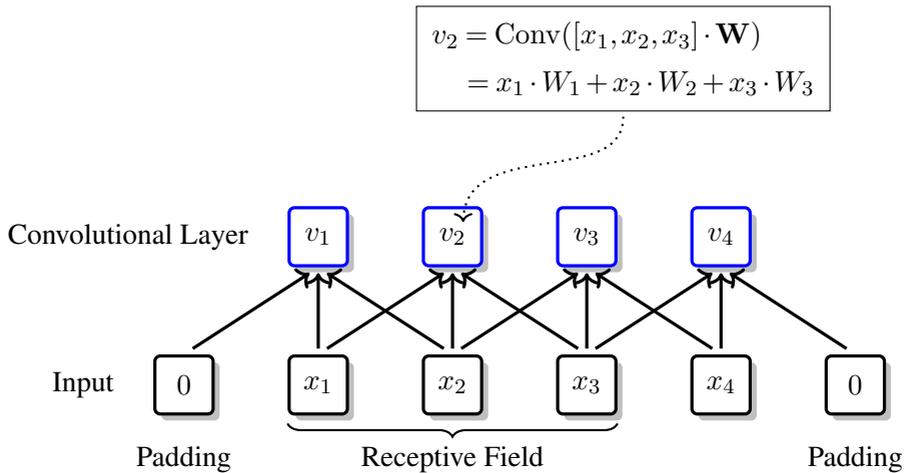


Figure 2.8: Convolution over a sequence of numbers  $\{x_1, x_2, x_3, x_4\}$  ( $r = 3$  and  $stride = 1$ ). The receptive field defines the region in the input that is taken in computing the output. Here the receptive field has a size of 3, that is, the convolutional operation covers three consecutive numbers in the input sequence. The filter (or convolutional kernel) outputs a weighted sum of these numbers. Each time we slide the receptive field over the input, the filter generates a new output. As such, the output of the convolutional layer is a vector of numbers. Also, a padding number (i.e. 0) is added to each end of the sequence so that the convolution is feasible when the receptive field is incomplete.

to a constant. For example, we can attend dummy feature vectors (say 0) to each end of the sequence so that all convolution operations are feasible. See Figure 2.8 for an example filter computed over a sequence of numbers. Note that the receptive fields of different filter applications may overlap. This is beneficial sometimes because it reduces information loss in feature representation when a low-level feature is used in forming multiple high-level features.

In addition, a convolutional layer can involve an activation function  $\psi(\cdot)$  to perform some non-linear mapping on the filter output. Let  $m_k = \lfloor \frac{m}{stride} \rfloor$  be the number of filter applications. The output of a convolutional layer is given by

$$\begin{bmatrix} h_1 & \dots & h_{m_k} \end{bmatrix} = \psi(\begin{bmatrix} v_1 & \dots & v_{m_k} \end{bmatrix}) \quad (2.39)$$

In general, a convolutional layer may not be restricted to a scalar-based input or a single filter. Often, we can take a vector as the representation of a token in the input sequence, and take a set of filters as feature extractors. To this end, we can adopt the same formulation as in Eqs. (2.37-2.39), but replace  $x_i$ ,  $v_i$  and  $h_i$  by the vectorized counterparts.

A convolutional layer is typically followed by a pooling layer. Like convolution, pooling is a function that sweeps a filter on a sequence. But the pooling operation does not have any parameters. It can be instead thought of as an aggregation function that performs down-sampling on the input sequence. There are several ways to design a pooling function. One of the most common methods is **max pooling** which outputs the maximum value in the

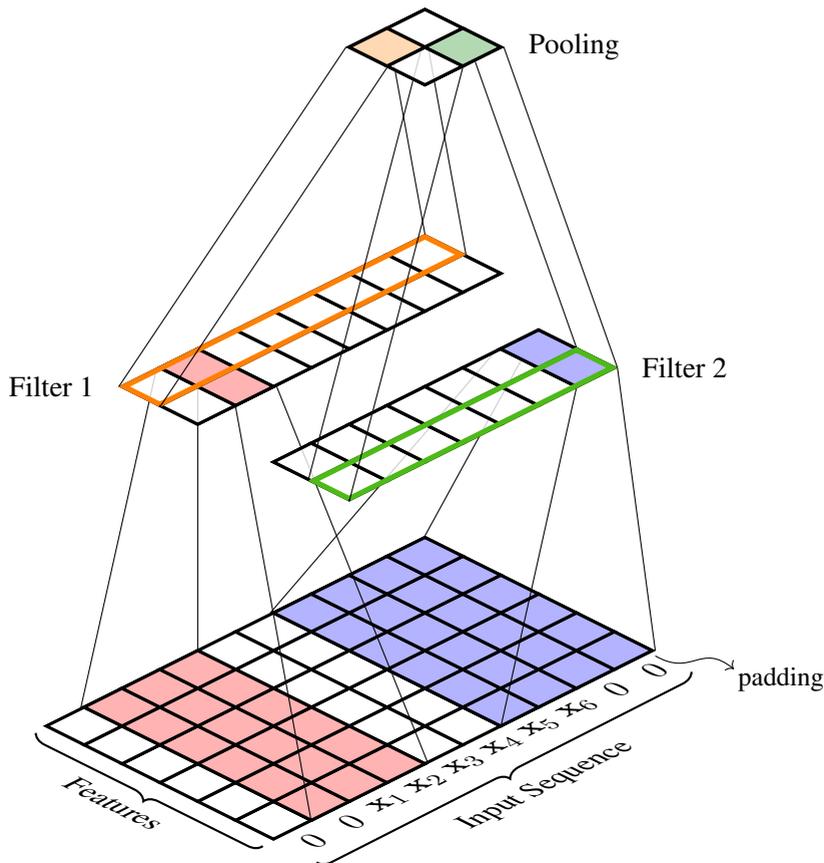


Figure 2.9: Example of CNN. There are two filters for the convolutional layer. The input is a sequence of 6 tokens represented in their feature vectors  $\{x_1, \dots, x_6\}$ . To tile the filters to fit the input sequence, two padding vectors are attached to each end of the sequence. When applying a filter, we map the feature vectors in the receptive field to a new feature vector. For example, for filter 1, the receptive field is a  $6 \times 3$  rectangle in the input, and the output is a 2-dimensional feature vector. By sweeping the filter on the sequence, we obtain a sequence of feature vectors, say, a sequence of 8 feature vectors, each having 2 features. The pooling layer fuzes features along the sequence. For example, performing the pooling on the output of filter 1 results in 2 fuzed features. The final output of the CNN is two 2-dimensional feature vectors.

receptive field. Another method is **averaging pooling** which outputs the averaged value over the receptive field. For a complete picture of how a CNN works, Figure 2.9 depicts a running example where convolutional and pooling operations are performed on a sequence of feature vectors via 2 filters.

### 2.3.3 Gate Units

In neural networks, a **gate** is used to decide how much information is passed along [Hochreiter and Schmidhuber, 1997]. Consider a standard RNN as an example. At each time step  $i$ , instead of directly passing the previous state  $s_{i-1} \in \mathbb{R}^{d_h}$  to the recurrent unit, it might be more

interesting to see how much information in  $s_{i-1}$  is useful for a next-step decision. In this case, we want  $s_{i-1}$  to be more like a real memory: as the time goes by, something should be memorized, and something should be forgotten.

A way to achieve this goal is to introduce a coefficient for controlling the scale of data flow. Here we reuse the notation in RNNs (see Section 2.3.1), but our description is general and could be applied to all the cases that need such a method. Let  $\mathbf{z} \in [0, 1]^{d_h}$  be a coefficient vector, where  $z_i = 0$  means that nothing is memorized for dimension  $i$ , and  $z_i = 1$  means that everything is memorized for dimension  $i$ . We can set  $\mathbf{z}$  as a gate on  $s_{i-1}$ . This can be formulated as:

$$\text{Gate}(\mathbf{z}, \mathbf{s}_{i-1}) = \mathbf{z} \odot \mathbf{s}_{i-1} \quad (2.40)$$

where  $\odot$  is the element-wise product of two vectors or matrices.  $\text{Gate}(\mathbf{z}, \mathbf{s}_{i-1})$  is an update of  $s_{i-1}$ . Thus,  $\mathbf{z}$  can be called an update gate, or a forget gate, or something similar. Alternatively, we can define the gating function in another way:

$$\text{Gate}(\mathbf{z}, \mathbf{s}_{i-1}) = (1 - \mathbf{z}) \odot \mathbf{s}_{i-1} \quad (2.41)$$

Eqs. (2.40) and (2.41) basically tell the same story but have different interpretations for  $\mathbf{z}$  in practice.

The key problem here is how to obtain  $\mathbf{z}$ . A general method is to define  $\mathbf{z}$  as the output of another network. For example, for a recurrent unit,  $\mathbf{z}$  can be defined to be:

$$\mathbf{z} = \text{Sigmoid}(\mathbf{s}_{i-1} \cdot \mathbf{W}_1 + \mathbf{x}_i \cdot \mathbf{W}_2 + \mathbf{B}) \quad (2.42)$$

The use of the Sigmoid activation function guarantees that the output falls into the range of  $[0, 1]$ . Note that Eq. (2.42) describes a learnable gate. This in turn makes the gate a part of the model and can be trained to fit the data. There are a number of methods to design a gate, and we will see a few in Chapter 4.

### 2.3.4 Normalization (Standardization) Units

A neural network works by transforming feature vectors layer by layer. While the multi-layer, multi-dimensional nature of neural networks enables the models to compute complex functions, it might lead to very different distributions of output activations across layers or features. This is a problem with deep neural networks because a model of this kind has to adapt to different distributions over different layers or different features [Ioffe and Szegedy, 2015]. Sometimes, as model parameters are initialized randomly in all layers and in all feature dimensions, it is likely for some features to be large values. In this case, the model would be biased to those large value features.

A way to mitigate this issue is **normalization**, which standardizes an  $n$ -dimensional feature vector  $\mathbf{s}$  as

$$\text{Normalize}(\mathbf{s}) = \alpha \odot \frac{\mathbf{s} - \mu}{\sigma + \epsilon} + \beta \quad (2.43)$$

where  $\mu \in \mathbb{R}^n$  and  $\sigma \in \mathbb{R}^n$  are the mean and standard deviation of  $\mathbf{s}$ , respectively.  $\epsilon$  is a small number used for numerical stability [Chiang and Cholak, 2022].  $\alpha \in \mathbb{R}^n$  and  $\beta \in \mathbb{R}^n$  are the parameters of the normalization unit. A simple choice is  $\alpha = 1$  and  $\beta = 0$ , whereas a more sophisticated method is to learn  $\alpha$  and  $\beta$  together with other parameters.

We may implement Eq. (2.43) in several ways that differ in how to define  $\mu$  and  $\sigma$ . Let us consider this for one dimension in  $\mathbf{s}$ , say  $s$ , in a general setting. Suppose that  $s$  is drawn from a set of feature values  $\Omega_k$ . The mean and the standard deviation on  $\Omega_k$  are then defined to be:

$$\mu_k = \frac{1}{|\Omega_k|} \cdot \sum_{s \in \Omega_k} s \quad (2.44)$$

$$\sigma_k = \sqrt{\frac{1}{|\Omega_k|} \cdot \sum_{s \in \Omega_k} (s - \mu_k)^2} \quad (2.45)$$

Several methods are available to define  $\Omega_k$ . For example, one can define  $\Omega_k$  as features in the same layer [Ba et al., 2016], or features along the same dimension over different samples or input positions [Ioffe and Szegedy, 2015; Ulyanov et al., 2016], or something in between them [Wu and He, 2018].

An advantage of normalization is to put features on the same scale and make them comparable. This has been found to be very helpful for stabilizing the training process and making neural networks better behaved. As we will see in subsequent chapters, normalization plays an important role in many successful systems.

As an aside, while the term *normalization* in deep learning is usually referred to as a process of subtracting the mean and dividing by the standard deviation, it is in fact a **standardization** process. In other areas, by contrast, normalization is more often referred to as a technique that scales all entries of a vector to the interval  $[0, 1]$ . Standardization has no such requirement. It instead tends to have the input centered around 0. In this sense, normalization might be a misnomer in deep learning somehow. Nevertheless, normalization and standardization are used interchangeably in this book when referred to processes like Eq. (2.43).

### 2.3.5 Residual Units

The success of deep neural networks has been mostly accredited to the more and more layers used in forming more complex functions. Although stacking a large number of layers is the simplest way to obtain a deep model, it has been pointed out that such a model is difficult to train. There are several reasons for this, e.g., optimization algorithms, gradient vanishing/exploding in passing through stacked layers, parameter initialization, and so on. Even, a further notable disadvantage comes with regard to feeding a single representation to upper-level feature extractors, as one might want direct access to the intermediate representations several layers ahead.

**Residual neural networks** are one of the most effective approaches to addressing these issues [He et al., 2016a]. They are a special type of neural networks that add **residual connections** (or **skip connections**, or **shortcut connections**) over layers in a layer stack. Let  $F(\mathbf{x})$  be a neural network that maps  $\mathbf{x}$  to some output. A residual neural network build on

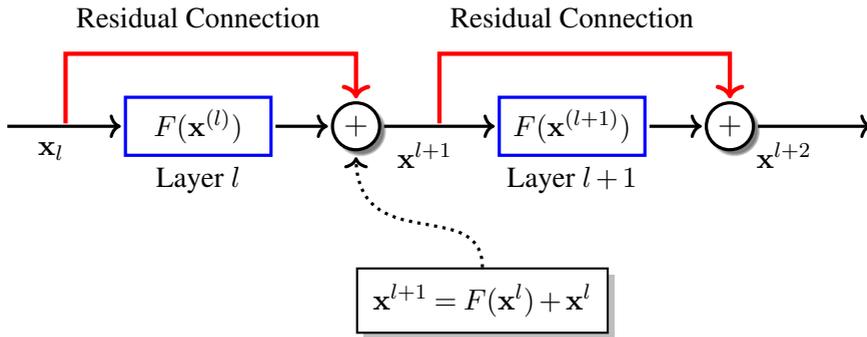


Figure 2.10: A 2-layer residual neural network. For each layer, there is a skip or shortcut connection (in red color) that directly adds the input to the output.

$F(\mathbf{x})$ , given by summing the mapping  $F(\mathbf{x})$  and the identity mapping  $\mathbf{x}$ :

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x} \quad (2.46)$$

A more common use of residual connections is in a neural network consisting of a number of identical layers. Let  $\mathbf{x}^l$  and  $\mathbf{y}^l$  be the input and output of layer  $l$  in a residual multi-layer neural network. The output of layer  $l$  can be defined as:

$$\mathbf{x}^{l+1} = F(\mathbf{x}^l) + \mathbf{x}^l \quad (2.47)$$

or

$$\mathbf{y}^l = F(\mathbf{y}^{l-1}) + \mathbf{y}^{l-1} \quad (2.48)$$

Figure 2.10 shows the architecture of a 2-layer residual neural network. Clearly, the residual connections add the outputs of current layers directly to the outputs of the next layers. The added identity mapping is generally thought of as one of the most effective ways to simplify the network and ease the information flow in a deep model.

## 2.4 Training Neural Networks

In this section, we turn to the training problem, which is fundamental in developing neural network-based systems. Most of the discussion here is focused on methods in a supervised learning setting. We will discuss unsupervised methods in Section 2.6.

### 2.4.1 Gradient Descent

The gradient method has been proven to be one of the most successful methods for training neural networks. The basic idea is to iteratively update parameters so that we can minimize a differentiable loss function. In an update, the values of the parameters are adjusted in a way that the loss degrades the fastest. In a mathematical sense, it requires the update to be in the

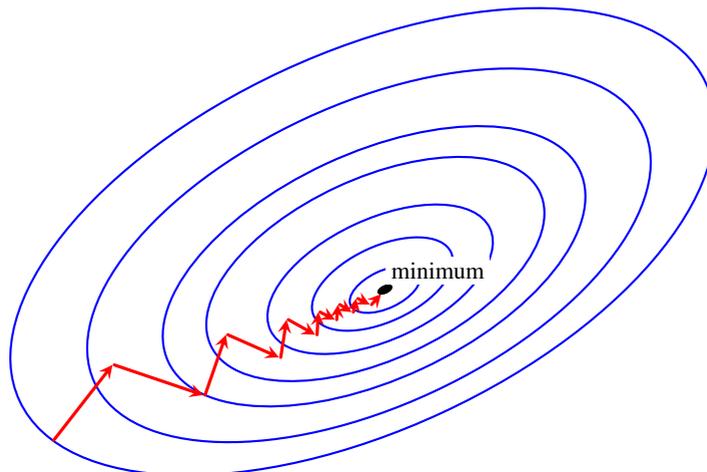


Figure 2.11: Gradient descent in a 2D space (blue lines stand for level sets). The goal is to find the parameters (i.e., values along the two dimensions) that minimize the value of a given loss function. Gradient descent does this by starting at a random point and stepping to the minimum in a number of updates of the parameters. In each update, it adjusts the parameters  $\theta_t$  in the direction that makes the loss lower. The idea here is that the update chooses the direction of the steepest ascent, that is, the model moves a step in the direction of the negative gradient of the loss (i.e.,  $-\frac{\partial L(\theta_t)}{\partial \theta_t}$ ). The size of the move is controlled by a hyper-parameter  $lr$ , called the learning rate. Thus, the amount of the change to the parameters is  $-lr \cdot \frac{\partial L(\theta_t)}{\partial \theta_t}$ . By adding this to  $\theta_t$ , we obtain the new parameters  $\theta_{t+1}$ . This process repeats for a number of updates until the value of the loss function is close to the minimum.

opposite direction of the gradient of the loss. This is known as **gradient descent** or **steepest descent**. Let  $\theta_t$  be the parameters at step  $t$  (call it an **update step** or a **training step**), and  $L(\theta_t)$  be the loss computed by the model parameterized by  $\theta_t$ . The **update rule** (or **delta rule**) of gradient descent is given by the equation:

$$\theta_{t+1} = \theta_t - lr \cdot \frac{\partial L(\theta_t)}{\partial \theta_t} \quad (2.49)$$

where  $\frac{\partial L(\theta_t)}{\partial \theta_t}$  is the gradient of the loss with respect to the parameters at step  $t$ . It can be obtained by running the error-propagation algorithm presented in Section 2.1.3. Since  $\theta_t$  is usually of multiple dimensions,  $\frac{\partial L(\theta_t)}{\partial \theta_t}$  could be a vector or matrix that has the same shape as  $\theta_t$ .  $lr$  is the **learning rate** that controls how big a step we take in the direction of the minimum. While  $lr$  can be simply set to a constant value during training, it is more common to adjust its value as the training proceeds (see Section 2.4.4 for a discussion). See Figure 2.11 for an illustration of gradient descent.

Eq. (2.49) gives a very basic definition of gradient descent. There are a number of improvements to the form of Eq. (2.49). Some of them are:

- **Gradient Descent with Momentum.** In physics, **momentum** is a vector quantity that describes the mass of motion. If we think of updating parameters as moving an object in a space, then we need to consider the momentum of the object at a position to determine the direction of the next move. This idea can be implemented by re-defining the update rule as:

$$\theta_{t+1} = \theta_t + v_t \quad (2.50)$$

where  $v_t$  is the velocity vector of the momentum. In the classic momentum method [Polyak, 1964],  $v_t$  is defined to be:

$$v_t = \lambda \cdot v_{t-1} - lr \cdot \frac{\partial L(\theta_t)}{\partial \theta_t} \quad (2.51)$$

$v_t$  retains some of the previous momentum (i.e.,  $v_{t-1}$ ), followed by a correction based on the gradient (i.e.,  $\frac{\partial L(\theta_t)}{\partial \theta_t}$ ).  $\lambda$  is a scalar for weighting  $v_t$  in an update. A well-known improvement to Eq. (2.51) is to take into account the momentum in the gradient, avoiding a too large velocity when approaching the minimum [Nesterov, 1983], like this

$$v_t = \lambda \cdot v_{t-1} - lr \cdot \frac{\partial [L(\theta_t) + \lambda \cdot v_{t-1}]}{\partial \theta_t} \quad (2.52)$$

A more detailed discussion on the difference between Eq. (2.51) and Eq. (2.52) can be found in Sutskever et al. [2013]’s paper.

- **Adaptive Gradient Descent.** In adaptive methods for gradient descent, the update rule is adapted to every parameter, rather than the whole model. **AdaGrad** is a method of this kind [Duchi et al., 2011]. It scales up the learning rate for parameters that have not been updated too much, and scales down the learning rate for parameters that have been much updated. Assume that  $\theta_t$  and  $\frac{\partial L(\theta_t)}{\partial \theta_t}$  are both  $d$ -dimensional vectors. We can define a new variable  $G \in \mathbb{R}^{d \times d}$  as the sum of the outer product of the gradient over the past  $t$  steps<sup>5</sup>:

$$G_t = \sum_{i=1}^t \left[ \frac{\partial L(\theta_i)}{\partial \theta_i} \right]^T \cdot \frac{\partial L(\theta_i)}{\partial \theta_i} \quad (2.54)$$

---

<sup>5</sup>Given two vectors  $\mathbf{a} = [a_1 \ \cdots \ a_d]$  and  $\mathbf{b} = [b_1 \ \cdots \ b_d]$ , the outer product of  $\mathbf{a}$  and  $\mathbf{b}$  is:

$$\begin{aligned} \mathbf{a} \otimes \mathbf{b} &= \mathbf{a}^T \cdot \mathbf{b} \\ &= \begin{bmatrix} a_1 \\ \vdots \\ a_d \end{bmatrix} \cdot [b_1 \ \cdots \ b_d] \\ &= \begin{bmatrix} a_1 b_1 & \cdots & a_1 b_d \\ \vdots & \ddots & \vdots \\ a_d b_1 & \cdots & a_d b_d \end{bmatrix} \end{aligned} \quad (2.53)$$

In general,  $(G_t)^{\frac{1}{2}} \in \mathbb{R}^d$  can be viewed as an indicator that describes to what extent a parameter has been updated so far. However, computing  $(G_t)^{\frac{1}{2}}$  is extremely expensive. So it is more common to use the diagonal of  $G_t$  instead. Then, the update rule of AdaGrad is given by:

$$\theta_{t+1} = \theta_t - \frac{lr}{\sqrt{\text{diag}(G_t) + \epsilon}} \odot \frac{\partial L(\theta_t)}{\partial \theta_t} \quad (2.55)$$

where  $\text{diag}(G_t)$  is the diagonal of  $G_t$ , i.e.,  $\text{diag}(G_t)(k) = G_t(k, k)$ .  $\epsilon$  is a smoothing factor for numerical stability. Instead of summing over those squared gradients in an unweighted manner, another way is to reduce the impact of “old” gradients and make “recent” gradients more important. **AdaDelta** considers this by accumulating squared gradients with a **decay factor** [Zeiler, 2012]:

$$g_t^2 = \sigma \cdot g_{t-1}^2 + (1 - \sigma) \cdot \left( \frac{\partial L(\theta_t)}{\partial \theta_t} \odot \frac{\partial L(\theta_t)}{\partial \theta_t} \right) \quad (2.56)$$

where  $\sigma$  is the decay factor of a value  $< 1$ . Like Eq. (2.55), the update rule for AdaDelta can be given by replacing  $\text{diag}(G_t)$  with  $g_t^2$ :

$$\theta_{t+1} = \theta_t - \frac{lr}{\sqrt{g_t^2 + \epsilon}} \odot \frac{\partial L(\theta_t)}{\partial \theta_t} \quad (2.57)$$

Since  $\sqrt{g_t^2 + \epsilon}$  can be seen as the root mean square (RMS) of the gradient, Eqs. (2.56-2.57) are also known as the **RMSProp** method [Hinton, 2018].

- **Adam (Adaptive Moment Estimation)**. The Adam optimizer combines the merits of both the adaptive gradient descent and momentum methods [Kingma and Ba, 2014]. It defines an estimate of the mean of the gradient (the first moment) and an estimate of the variance of the gradient (the second moment). Let  $m_t$  and  $v_t$  be the two moment estimates. They are given by the equations:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \frac{\partial L(\theta_t)}{\partial \theta_t} \quad (2.58)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \left( \frac{\partial L(\theta_t)}{\partial \theta_t} \odot \frac{\partial L(\theta_t)}{\partial \theta_t} \right) \quad (2.59)$$

where  $\beta_1, \beta_2 \in [0, 1]$  are hyper-parameters for a trade-off between the previous estimate and the gradient (or squared gradient) at the current step.  $\beta_1$  and  $\beta_2$  are also treated as the decay factors of these averages. For example, common choices for  $\beta_1$  and  $\beta_2$  are 0.9 and 0.999. As the initial moments are set to 0, these estimates are biased to 0 vectors at the very beginning of the training process. To address this issue, bias corrections are

used in Adam, leading to bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (2.60)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (2.61)$$

Since  $\beta_1, \beta_2 < 1$ , the corrections would be sufficiently small if a larger number of updates are performed. The update rule is finally defined to be:

$$\theta_{t+1} = \theta_t - lr \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \quad (2.62)$$

Eq. (2.62) resembles the general form of gradient descent, but makes use of both the momentum method (i.e., the moving average of the past gradients) and the adaptive method (i.e., the moving average of the past squared gradients). In practice, Adam has become a popular optimizer for training neural networks.

Improving gradient descent is an active sub-field of deep learning, but a full discussion of all those techniques is beyond the scope of this document. A few related issues will be discussed in the remainder of this section.

On a last note of this subsection, a practical issue that one should consider in utilizing iterative training methods is when to stop training. **Stopping criterion** is a general topic in optimization. For gradient descent and its variants, it is common practice to set a maximum number of training steps or **training epochs**<sup>6</sup>, say 20,000 steps, or 100 epochs. As an alternative, we can perform training until convergence. For example, we can say that the training coversages if the loss tends to be stable for a number of training steps. When there is some data for validating the model, a better method may be to check the states of the model on validation data. For example, we can stop the training when the prediction error increases on the validation data. This method, known as **early stopping**, is often used as a means of regularization. In Section 2.5.3, we will see more details about how to early stop the training by using a validation dataset. On the algorithmic side, there has been much interest and work in studying the convergence and error bounds for machine learning methods. We refer the interested reader to a few textbooks for further discussions [Mohri et al., 2018; Kochenderfer and Wheeler, 2019].

## 2.4.2 Batching

The loss function is an essential aspect of the training of neural networks. While a number of mathematical forms are available to define the loss function (see Section 1), we still need to decide in what scale of samples we use that loss function. Perhaps the simplest method is **stochastic gradient descent (SGD)**. In each update of parameters, SGD computes the loss function on a single sample that is randomly selected from the training dataset. Let  $D$  be a set of training samples, and  $(\mathbf{x}^{(i)}, \mathbf{y}_{\text{gold}}^{(i)})$  be a randomly selected sample from  $D$ . Given a neural

<sup>6</sup>A training epoch means that the trainer goes over the whole training dataset for one time.

network

$$\mathbf{y}_\theta^{(i)} = F_\theta(\mathbf{x}^{(i)}) \quad (2.63)$$

the loss of SGD is defined to be:

$$L(\theta) = L(\mathbf{y}_\theta^{(i)}, \mathbf{y}_{\text{gold}}^{(i)}) \quad (2.64)$$

where  $L(\mathbf{y}_\theta^{(i)}, \mathbf{y}_{\text{gold}}^{(i)})$  is a sample-level loss function that counts errors in the model output  $\mathbf{y}_\theta^{(i)}$  with respect to the benchmark  $\mathbf{y}_{\text{gold}}^{(i)}$ .

SGD has been one of the most important optimization methods in machine learning due to its simplicity. However, SGD converges slowly because it is just an analog of the actual gradient on the entire training set. To estimate the gradient in a more precise way, we can take into account a set of samples (call it a **batch**) in computing the loss. This method is known as **batching**. Let  $S$  be a set of samples from  $D$ . The loss function is then defined on  $S$ , as follows

$$L(\theta) = \frac{1}{|S|} \cdot \sum_{(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \in S} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \quad (2.65)$$

If  $S = D$ , then we have the **batch gradient descent (BGD)** method, i.e., the gradient is estimated on the entire set of training samples. In general, batch gradient descent is what we would ordinarily call gradient descent. However, calculating the loss on all the training samples simultaneously is time consuming. In practice, it is more common to use a batch much smaller than  $D$ . This is known as **mini-batch gradient descent**. It is adopted in learning real-world systems for its good efficiency and strong performance.

As another “bonus”, batching is generally used as a way to make dense computation on matrices for system speed up. Assume that  $S$  consists of  $m$  samples  $\{(\mathbf{x}^{(i_1)}, \mathbf{y}_{\text{gold}}^{(i_1)}), \dots, (\mathbf{x}^{(i_m)}, \mathbf{y}_{\text{gold}}^{(i_m)})\}$ . We can batch all input vectors and benchmark vectors as matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(i_1)} \\ \vdots \\ \mathbf{x}_{\text{gold}}^{(i_m)} \end{bmatrix} \quad (2.66)$$

$$\mathbf{Y}_{\text{gold}} = \begin{bmatrix} \mathbf{y}_{\text{gold}}^{(i_1)} \\ \vdots \\ \mathbf{y}_{\text{gold}}^{(i_m)} \end{bmatrix} \quad (2.67)$$

Then, we can run the neural network on the batched input and output, like this:

$$\mathbf{Y}_\theta = F_\theta(\mathbf{X}) \quad (2.68)$$

Likewise, we can compute the batched loss

$$L(\theta) = \frac{1}{m} \cdot L(\mathbf{Y}_\theta, \mathbf{Y}_{\text{gold}}) \quad (2.69)$$

where  $L(\mathbf{Y}_\theta, \mathbf{Y}_{\text{gold}})$  vectorizes the computing of  $\sum_{k=1}^m L(\mathbf{y}_\theta^{(i_k)}, \mathbf{y}_{\text{gold}}^{(i_k)})$ . Eqs. (2.68-2.69) prevent the repetitive calls of the forward and backward passes on individual samples. They instead pack everything in a single pass through the network. This makes better use of maximum available compute on modern GPUs which are the majority of the devices for running deep learning systems.

### 2.4.3 Parameter Initialization

Gradient descent requires that the training process starts from some initial parameters. Since the training objective in a practical system is often a non-convex function with many local minima, the performance of the resulting model is highly sensitive to the parameter initialization step. Here we describe some of the most common methods of parameter initialization.

- **Constant Initialization.** The first method could assign the same value to all parameters (or all parameters of a parameter matrix). This method, though quite simple, results in that all output entries of a model make no difference, rendering the model meaningless. It performs poorly in most cases if no randomness is introduced into training.
- **Initialization with Predefined Distributions.** A useful way is to randomly initialize parameters by some distributions. The simplest of this kind is to assign a parameter a value drawn from a uniform or Gaussian distribution, e.g., a random value in the interval  $[-0.1, 0.1]$ . Interestingly, this method is satisfactory in most cases in practice.
- **Layer-sensitive Initialization.** An extension to random initialization is to use tailored distributions for different layers of a neural network. **Xavier initialization** is a well-known method of this kind [Glorot and Bengio, 2010]. Given a layer  $\mathbf{y} = \psi(\mathbf{x} \cdot \mathbf{W} + \mathbf{B})$ , let  $d_{\text{in}}$  and  $d_{\text{out}}$  be the numbers of the input and output dimensions (i.e., the row and column numbers of  $\mathbf{W}$ ). The standard Xavier initialization method, also known as the **LeCun initialization** method [LeCun et al., 2012], gives a random number to every parameter of  $\mathbf{W}$ :

$$\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}} \sim U\left(-\frac{1}{\sqrt{d_{\text{in}}}}, \frac{1}{\sqrt{d_{\text{in}}}}\right) \quad (2.70)$$

where  $U(-a, a)$  means a uniform distribution over the interval  $[-a, a]$ . Likewise, we can initialize the bias term in a similar way. As an improvement, the normalized Xavier initialization method considers both  $d_{\text{in}}$  and  $d_{\text{out}}$  in defining the distribution, like this:

$$\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}} \sim U\left(-\sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}\right) \quad (2.71)$$

More details can be found in the original paper. Note that the uniform distributions can

be replaced by the normal distributions with mean = 0 and variance =  $\frac{1}{d_{\text{in}}}$  or  $\frac{6}{d_{\text{in}} + d_{\text{out}}}$ .

Many parameter initialization methods are designed for certain types of neural networks. For example, Xavier initialization is assumed to work with the Sigmoid and hyperbolic tangent activation functions. For ReLU, one can refer to He et al. [2015]’s work. Another example is initialization for deep neural networks. It has been found that appropriate initialization is critical to the success of extremely deep models in NLP. Considering the model depth as an additional factor in initialization, we can modify Eq. (2.71) to be:

$$\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}} \sim U\left(-\frac{\alpha_s}{l} \cdot \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}, \frac{\alpha_s}{l} \cdot \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}\right) \quad (2.72)$$

where  $l$  is the depth for a layer, and  $\alpha_s$  is a hyper-parameter. Apart from this, several methods are proposed to address the initialization of deep neural networks, including the Lipschitz initialization [Xu et al., 2020], the T-Fixup initialization [Huang et al., 2020a], the Admin initialization [Liu et al., 2020c], and so on.

Note that in practice we do not have to restrict training to a single starting point. It is common to try a few starting points by using different initialization methods or random seeds, and to choose the best performing one from these tries. It generally helps when local minimums abound.

#### 2.4.4 Learning Rate Scheduling

To achieve desirable results, it is essential to carefully configure the learning rate throughout the learning process. While some of the update rules, as noted above, have considered scaling the gradient for different parameters, **learning rate scheduling** is conventionally focused more on designing heuristics to adjust  $lr$  over training steps. In a practical sense, a too large learning rate usually leads to overshooting around the minimum, while a too small learning rate usually leads to slow convergence (see Figure 2.12). A common idea is to learn fast at the beginning (i.e., a large learning rate) and learn slowly when the loss is close to the minimum (i.e., a small learning rate). Here we present some of the popular methods for learning rate scheduling.

- **Fixed Learning Rates.** Fixing the learning rate is generally a bad strategy, but could be used in prototyping systems, e.g., a quick test of a new method by training it for only a few epochs.
- **Learning Rate Decay.** Decay is a commonly-used technique for learning rate scheduling. There are many approaches to this idea. For example, one can halve the learning rate after each training epoch. Here we use  $n_t$  to denote the number of training steps, and  $\tau_{\text{decay}}$  be how often we change the learning rate (e.g., 100 steps). Table 2.2 shows several decay functions for learning rate scheduling.
- **Warmup and Decay.** As noted in Section 2.4.3, it is common to initially set model parameters to random values when a neural network is being trained. However, learning from scratch with a large learning rate is usually not a good choice because the gradient at the early stage of the training is not much precise and the state of the model is unstable.

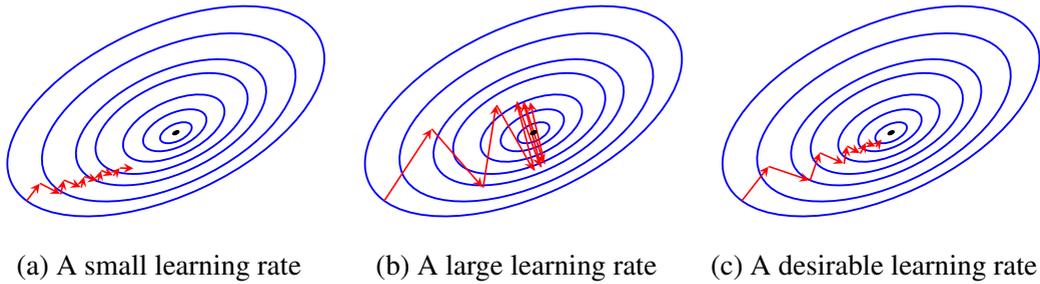


Figure 2.12: Learning with different learning rates. Small learning rates (left) help us step to the minimum in a precise way, but require much additional time for convergence. Large learning rates (middle), on the other hand, lead to fast learning, which is very beneficial when we are far away from the minimum. However, as we get closer to the minimum too large learning rates cause overshooting. A more desirable strategy (right) may be to learn the model in a reasonably fast way when there is a long way to go, and to learn the model slower when we are close to the minimum.

Thus, it is more reasonable to start with a small learning rate and gradually increase it. Then, when the model is trained for some time, the learning rate begins to decay as usual. Such a thought motivates the warmup and decay method for learning rate scheduling. A popular form of this method in recent studies is proposed in [Vaswani et al. \[2017\]](#)'s work, as follows:

$$lr_{n_t} = lr_0 \cdot \min \left( \left( \frac{n_t}{n_{\text{decay}}} \right)^{-0.5}, \frac{n_t}{n_{\text{decay}}} \cdot (n_{\text{warmup}})^{-1.5} \right) \quad (2.73)$$

where  $lr_0$  is the initial learning rate, and  $n_{\text{warmup}}$  is a hyper-parameter that specifies for how many steps we execute the warmup process. Figure 2.13 plots the curve of Eq. (2.73) where  $n_{\text{warmup}}$ ,  $n_{\text{decay}}$ , and  $lr_0$  are set to 4,000, 1 and 1. We see that the learning rate increases linearly in the first  $n_{\text{warmup}}$  steps and then decays as an inverse square root function.

Choosing an appropriate learning rate scheduling strategy is a highly empirical problem, and there are no universally good choices. The problem is even harder if we consider the correlation between the learning rate and other aspects of the training, though learning rate scheduling is typically taken to be an individual task. For example, when a larger batch is used in training, a larger learning rate is desired for a good result [[Ott et al., 2018b](#); [Smith et al., 2018](#)]. So, making good learning rate choices is still difficult and time-consuming in neural network applications. Occasionally one needs a large number of trial-and-test runs to find a desirable learning rate setup for the particular problem at hand.

Entry	Formula	Hyper-parameters
Piecewise Constant Decay	$lr_{n_t} = \beta_i$ if $\gamma_i \leq \frac{n_t}{n_{\text{decay}}} < \gamma_{i+1}$	values $\{\beta_1, \dots, \beta_m\}$ thresholds $\{\gamma_1, \dots, \gamma_m\}$
Exponential Decay	$lr_{n_t} = lr_0 \cdot \lambda^{\frac{n_t}{n_{\text{decay}}}}$	decay rate $\lambda$ , init. lr. $lr_0$
(Drop) Exponential Decay	$lr_{n_t} = lr_0 \cdot \lambda^{\lfloor \frac{n_t}{n_{\text{decay}}} \rfloor}$	decay rate $\lambda$ , init. lr. $lr_0$
Natural Exponential Decay	$lr_{n_t} = lr_0 \cdot \exp(-\lambda \cdot \frac{n_t}{n_{\text{decay}}})$	decay rate $\lambda$ , init. lr. $lr_0$
Inverse Time Decay	$lr_{n_t} = lr_0 \cdot \frac{1}{1 + \lambda \cdot \frac{n_t}{n_{\text{decay}}}}$	decay rate $\lambda$ , init. lr. $lr_0$
(Drop) Inverse Time Decay	$lr_{n_t} = lr_0 \cdot \frac{1}{1 + \lambda \cdot \lfloor \frac{n_t}{n_{\text{decay}}} \rfloor}$	decay rate $\lambda$
Cosine Decay	$lr_{n_t} = lr_0 \cdot ((1 - \alpha) \cdot c_{\text{decay}} + \alpha)$ $c_{\text{decay}} = \frac{1}{2} \cdot (1 + \cos(\pi \cdot \frac{n_t}{n_{\text{decay}}}))$	coefficient $\alpha$ init. lr. $lr_0$

Table 2.2: Decay functions.  $\lambda$  = decay rate,  $lr_0$  = initial learning rate, and  $\{\beta_i\}$ ,  $\{\gamma_i\}$  and  $\alpha$  = other hyper-parameters.

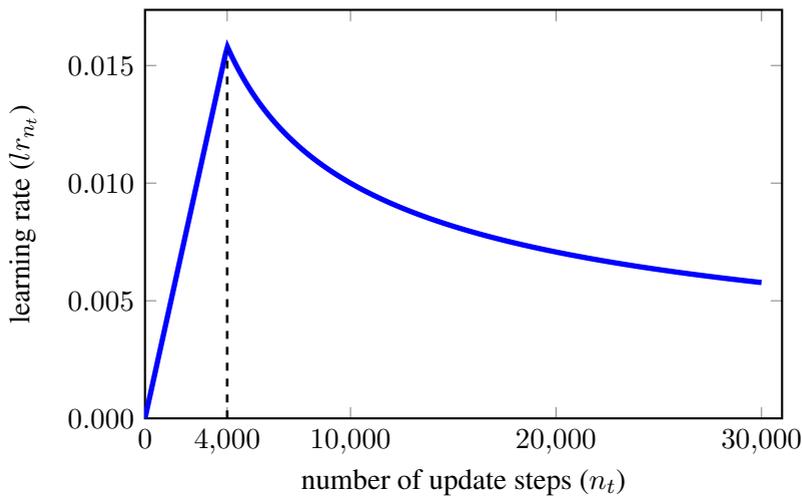


Figure 2.13: Learning rate scheduling: warmup and then decay ( $n_{\text{warmup}} = 4,000$ ,  $n_{\text{decay}} = 1$ , and  $lr_0 = 1$ ). The learning rate increases linearly with  $n_t$  for the first 4,000 steps. Then, the learning rate follows an inverse square root function and decays as the learning continues. The change of the rate learning will be small if  $n_t$  is sufficiently large, indicating the fine adjustment of the parameters when we are approaching the minimum of the loss.

## 2.5 Regularization Methods

We now discuss the regularization methods for preventing overfitting. While regularization is a wide-ranging topic in machine learning, we present some of those that are commonly adopted in training neural networks.

### 2.5.1 Norm-based Penalties

One of the most popular methods involves a regularization term based on the  $l_p$  norm. A general form of the regularized objective can be defined as:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} L(\theta) + \alpha \cdot R(\theta) \quad (2.74)$$

where  $R(\theta)$  is the regularization term weighted by a coefficient  $\alpha$ . In general,  $R(\theta)$  serves as an additional loss that penalizes complex models. This is motivated by the fact that complex models are more likely to overfit the data (see Section 1). To impose a penalty on the model complexity, a simple way is to define  $R(\theta)$  as the  $l_1$  norm on the parameters  $\theta$ . Let us treat  $\theta$  as a vector of parameters. The  $l_1$  norm-based regularization term is given by

$$R(\theta) = \sum_i |\theta_i| \quad (2.75)$$

Eq. (2.75) penalizes models having large value parameters. This can be understood in a way from a polynomial function: large coefficients of variables in a polynomial function lead to a complex curve. Typically, regularization with the  $l_1$  norm is referred to as the  $l_1$  regularization or the **Lasso regularization**. Such a method does not require updates of the trainer, and can be implemented by standard gradient descent. More interestingly, the  $l_1$  regularization typically provides sparse solutions to the original training objective. It biases the model to those having small values (or even zero values) for most of the parameters and large values for only very few parameters. This also implies an inherent ability of feature selection because parameters are forced to be close to zero for not-so-important features.

An alternative to the  $l_1$  regularization is the  $l_2$  **regularization** or the **Ridge regularization**. In the  $l_2$  regularization, the regularization term is given by

$$R(\theta) = \sqrt{\sum_i |\theta_i|^2} \quad (2.76)$$

Like the  $l_1$  norm, the  $l_2$  norm penalizes the cases that deviate the model parameters far away from the origin. However, it slightly differs from the  $l_1$  norm in that the  $l_2$  norm enforces all parameters to have small values (but not necessarily to be zeros) and there are no large value parameters. In this sense, the use of the  $l_2$  norm does not introduce sparsity into the solution but performs “smoothing” on the underlying distributions of features. Note that the  $l_2$  regularization has a relatively bigger effect of regularization. So, it is sometimes called **weight decay** to emphasize its ability to prevent the model from learning parameters of too large values.

In a broader sense of machine learning, Eq. (2.75) offers a general method to introduce prior knowledge into the training of a neural network. There are a number of ways to design the regularization term, and addressing overfitting is just one purpose of these designs. We can see many applications of this approach in NLP, and will see a few examples in the remaining chapters of this document.

## 2.5.2 Dropout

In a real-world neural network, a layer typically involves hundreds or thousands of neurons and produces a feature vector accordingly. While each of these features is computed by a single neuron, they work together to form the input to each neuron of the following layer. As a result, a feature is forced to cooperate with other features. It is like a group of people sitting together and making a collective decision. Although a member could have opinions independently, he or she occasionally tries to correct the error when all other members have had their decisions. In this case, every group member is co-adapted to others in the group [Hinton et al., 2012]. From a feature engineering standpoint, the **co-adaptation** of neurons helps when modeling complex problems, as it implicitly makes some sort of higher order features. Beyond this, the strong supervision information (e.g., propagating errors through layers) could strengthen the co-adaptation in training. This explains more or less why a neural network with a large number of neurons can fit complex curves. At test time, however, the co-adaptation prevents generalization. Since all neurons of a layer are learned to collaborate well on the training data, a small change in the input could affect all these neurons and lead to a big change in the behavior of the neural network.

A way to mitigate or eliminate complex co-adaptations is to learn for each neuron to predict in the absence of other neurons. To this end, one can simply drop some of the neurons in training. This method is known as **dropout** [Srivastava et al., 2014]. Let  $n$  be the number of neurons of a layer. Given a probability  $\rho$  (call  $1 - \rho$  the **dropout rate**), we can generate an  $n$ -dimensional mask vector  $\mathbf{M}_{\text{drop}}$  where every entry is set to 1 with a possibility of  $\rho$ , and set to 0 with a possibility of  $1 - \rho$ . Then, a dropout layer can be defined as

$$\mathbf{y} = \mathbf{M}_{\text{drop}} \odot \psi(\mathbf{x} \cdot \mathbf{W} + \mathbf{B}) \quad (2.77)$$

where  $\psi(\mathbf{x} \cdot \mathbf{W} + \mathbf{B})$  is a usual single-layer neural network. Eq. (2.77) only activates the neurons whose masks are 1. For dropped neurons, all connections from/to these neurons are blocked (see Figure 2.14 (a)). During training,  $\mathbf{M}_{\text{drop}}$  is randomly generated in a call of the forward and backward passes. A neuron therefore can learn to work with different neurons each time and would not adapt to the same group of “co-workers”. Another way to understand dropout is to view it as learning sub-models of a “big” model. The use of Eq. (2.77) is essentially a sampling process that extracts a sub-network from the original network. So, training with dropout is doing something like training an exponentially large number of sub-networks<sup>7</sup>. On the other hand, the training is efficient because these sub-networks share the same parameters for the same neuron and the update of a parameter can benefit exponentially many sub-networks.

At test time, all these sub-networks are combined for prediction. In this case, we do not need to drop any neuron but use the original network as usual. This makes it simple to implement dropout: a neuron is present with some probability on the training data, and all neurons are present and work together on the test data. Since the connections between neurons are involved with a probability of  $\rho$  in training, the learned weights are scaled down with  $\rho$  in

<sup>7</sup>For a single-layer network having  $n$  neurons, there are  $2^n$  possible sub-networks.

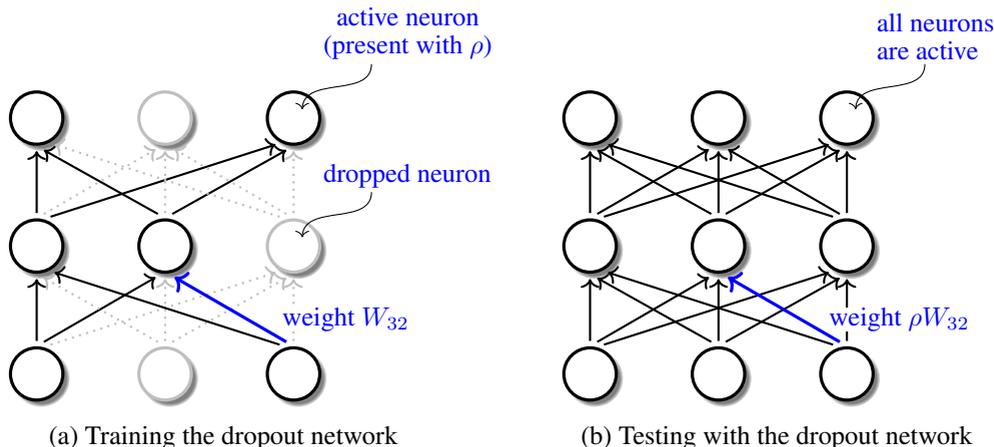


Figure 2.14: Dropout for a multi-layer neural network (training vs test). At training time, every neuron is randomly dropped with a probability of  $1 - \rho$ , resulting in a slimmed network. In this sense, dropout training is essentially a process of learning an exponentially large number of sub-networks. At test time, the full network is used as usual, which is the result of combining all those sub-networks for prediction. Since all connections between neurons are activated with the probability  $\rho$  during training, the weights of the predicting network are scaled down with  $\rho$ .

the predicting network, i.e., a layer has a form:

$$\mathbf{y} = \psi(\mathbf{x} \cdot \rho \mathbf{W} + \rho \mathbf{B}) \quad (2.78)$$

See Figure 2.14 for a comparison of training and applying a dropout network. Eq. (2.78) requires an update of the predicting system. An alternative is to take into account the scaling issue only in the training process and leave the predicting system as it is. For example, we can scale up all the parameters with  $\frac{1}{\rho}$  in dropout training, like this

$$\mathbf{y} = \mathbf{M}_{\text{drop}} \odot \psi\left(\mathbf{x} \cdot \frac{1}{\rho} \mathbf{W} + \frac{1}{\rho} \mathbf{B}\right) \quad (2.79)$$

Since multiplying  $\frac{1}{\rho} \mathbf{W}$  with  $\rho$  yields  $\mathbf{W}$  (this also holds for the bias term  $\mathbf{B}$ ), we can use  $\mathbf{W}$  (and  $\mathbf{B}$ ) as the parameters of the predicting system.

### 2.5.3 Early Stopping

In Chapter 1 we have discussed a bit of how to stop the training by monitoring the performance on the validation data. It can be treated as a way of model selection that seeks an appropriate state between underfitting and overfitting. Note that early stopping is not just an empirical method. It is also well explained from the perspective of statistical learning theory. For example, researchers have found that, under some conditions, early stopping has a similar effect as the  $l_2$  regularization and restricts the learning to the region of small value parameters [Bishop, 1995a; Goodfellow et al., 2016]. Also, other research shows that some early stopping rules

have a tight relationship with the bias-variance trade-off and could guarantee nice properties of convergence [Yao et al., 2007].

On the other hand, early stopping requires several heuristics to make it practical and useful. The first problem is the condition of stopping. Ideally, one might imagine that there is a perfect U-shaped error curve on the validation data, and the training can be halted immediately when the error starts to increase. The truth, however, is that the error curve cannot be simply described as a strictly convex function of the training time. After drops in the error in a certain number of training steps, the performance of the model tends to fluctuate, leading to many local minimums. The problem would be more interesting if one wants to save time and stop the training as early as possible. However, we never know whether the current choice or decision is the best one because we have no idea of what happens next. A commonly-used method is to decide whether the training should stop by checking the model states for a number of past update steps (or epochs) [Prechelt, 1998]. Some early stopping conditions are:

- The change in the performance is below a threshold for a given number of steps (or epochs).
- The change in the model parameters is below a threshold for a given number of steps (or epochs).
- The average performance over a given number of steps (or epochs) starts to decrease.
- The maximum performance over a given number of steps (or epochs) starts to decrease.

However, using the model at the point that we stop the training is not always a good choice. In practice, a model often has a large variance in generation error around that point, making model selection more difficult. Instead of “selecting” a model, an alternative way is to combine multiple models. For example, we can save the model for every run of a given number of training steps (call each copy of the model a checkpoint). The final model is induced by averaging the parameters of the last few checkpoints. For better results, one may use more sophisticated ensemble methods (see Section 1).

## 2.5.4 Smoothing Output Probabilities

In statistics, smoothing refers to the process of reducing the value of noisy data points (probably of high values) and increasing the value of normal data points. It is typically used when a distribution is estimated on small data and the probabilities of rare events are not well estimated. For example, consider the language modeling problem described in Section 2.2. A language model is trained in a way that enforces the model to output a one-hot distribution, that is, the total probability of 1 is occupied by only one word, leaving other words assigned zero probabilities. It may be more desirable to distribute the probability to all words, even though many of them are not observed to be the answer given the previous words. In this way, the model learns to make a soft prediction of word probabilities so that it can generalize better on unseen data.

Given a distribution  $\mathbf{p} = [p_1 \dots p_n]$ , it is the purpose of smoothing that we obtain the new estimate between  $\mathbf{p}$  and a uniform distribution  $\frac{1}{n}$ . A common approach to this idea is to

use a **shrinkage estimator** to improve  $\mathbf{p}$  by making it closer to  $\frac{1}{n}$ . For example, the additive smoothing mentioned in Section 1 is a simple type of shrinkage estimator. Here we consider, for example, smoothing a multinomial distribution. Let  $p_k$  denote the probability of event  $k$  and  $s_k$  denote a quantity that describes some observed “count” of the event. The probability  $p_k$  is given by

$$p_k = \frac{s_k}{\sum_{k=1}^n s_k} \quad (2.80)$$

Then, the smoothed version of  $p_k$  is defined as

$$\hat{p}_k = \frac{s_k + \alpha}{\sum_{k=1}^n (s_k + \alpha)} \quad (2.81)$$

It simply adds a quantity  $\alpha$  to each  $s_k$ . The value of  $\alpha$  controls the smoothness of the resulting estimate. For example,  $\hat{p}_k = p_k$  if  $\alpha = 0$ , and  $\hat{p}_k \approx \frac{1}{n}$  if  $\alpha$  chooses an extremely large value.

Apart from additive smoothing, we can smooth a distribution in a Softmax manner, as follows

$$\hat{p}_k = \frac{\exp(s_k/\beta)}{\sum_{k=1}^n \exp(s_k/\beta)} \quad (2.82)$$

This form is known as an instance of the **Boltzmann distribution** [Uffink, 2017], where  $s_k$  is viewed as the negative energy of a state, and  $\beta$  is viewed as the temperature indicating the degree of smoothing. Note that  $s_k$  can be interpreted in many ways. For example, in a neural network,  $s_k$  is typically defined as the state of a neuron. Sometimes,  $s_k$  can even be a probability. This means that we can directly apply Eqs. (2.81-2.82) to any  $\mathbf{p}$  even if there is no prior knowledge about how  $\mathbf{p}$  is estimated. Then, we can rewrite Eqs. (2.81-2.82) by replacing  $s_k$  with  $p_k$ :

$$\hat{p}_k = \frac{p_k + \alpha}{\sum_{k=1}^n (p_k + \alpha)} \quad (2.83)$$

$$\hat{p}_k = \frac{\exp(p_k/\beta)}{\sum_{k=1}^n \exp(p_k/\beta)} \quad (2.84)$$

Another method of smoothing is to interpolate  $\mathbf{p}$  with the uniform distribution. A form of the interpolation is given by

$$\hat{p}_k = (1 - \epsilon) \cdot p_k + \epsilon \cdot \frac{1}{n} \quad (2.85)$$

where  $\epsilon$  is a hyper-parameter indicating to what extent we rely on the uniform distribution in computing  $\hat{p}_k$ . To illustrate how Eq. (2.85) works, let us suppose that  $\mathbf{p}$  is a one-hot vector, say,  $p_k = 1$  if  $k = z$  and  $p_k = 0$  otherwise. By using Eq. (2.85), we subtract an amount of probability (i.e.  $\epsilon$ ) from  $p_z$ . The subtracted amount of probability is then redistributed to all dimensions evenly, making the resulting distribution more flat-topped and smoother. See Figure 2.15 for an illustration.

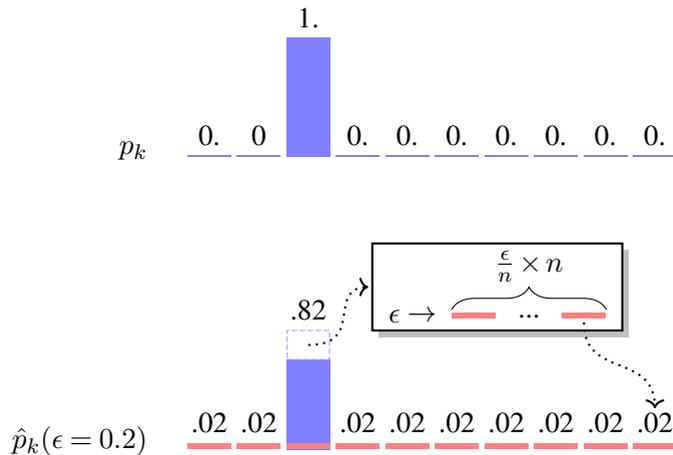


Figure 2.15: Smoothing a distribution by interpolating it with the uniform distribution:  $\hat{p}_k = (1 - \epsilon) \cdot p_k + \epsilon \cdot \frac{1}{n}$ . For each dimension  $k$ , it subtracts an amount of  $\epsilon$  from the probability  $p_k$  and redistributes this amount of probability evenly to all the variables, that is, every variable gets a probability of  $\epsilon \cdot \frac{p_k}{n}$ .

In NLP, since many systems make probability-like predictions, a common application of smoothing is to smooth a system’s output. There are two ways. First, we can smooth the benchmark probability such that the model is guided by the generalized error rather than the error made by hard decisions. For example, the **label smoothing** technique adopts the same form as Eq. (2.85) and improves the benchmark representation on categorical data [Szegedy et al., 2016]. Second, we can reduce the steepness and increase the tailedness of a predicted distribution<sup>8</sup>. This method is often used when the posterior probability of the prediction is required, such as minimum Bayesian risk decoding/training [Bickel and Doksum, 2015; Goodman, 1996a; Kumar and Byrne, 2004a].

### 2.5.5 Training with Noise

Above, we have shown that adding some amount to each observed count of events in predicting a probability can improve generalization. From a **robust statistics** point of view [Olive, 2022], this is equivalent to improving the robustness of an estimator where a skewed distribution often leads to a biased model. The addition of a small perturbation to the estimate can prevent large biases caused by outliers and unexpected observations of rare events. In this sense, smoothing can be regarded as a way of introducing noise into training, that is, we impose a prior of uniform distribution on the estimate though the correct estimate may not be uniform.

Noisy training works with an idea that a model is learned to work in non-ideal conditions and avoid overfitting data points of extreme values. Here the term *noise* has a wide meaning, and there are a few different ways to regularize training with noise. One of the simplest methods is to use noise-sensitive training objectives. For example, smoothing the benchmark

<sup>8</sup>In general one may want a distribution to be a Mesokurtic curve.

distribution (e.g., the one-hot representation of the correct prediction) can be seen as a way of making noisy annotations. Alternatively, we can add random noise to the input, output, and intermediate state of a neural network. A common choice is the **Gaussian noise**. Suppose we have a vector  $\mathbf{x} \in \mathbb{R}^n$ . The addition of the Gaussian noise defines a new vector, as follows

$$\mathbf{x}_{\text{noise}} = \mathbf{x} + \mathbf{g} \quad (2.86)$$

where  $\mathbf{g} \in \mathbb{R}^n$  is a vector of noise. It follows a Gaussian distribution:

$$\mathbf{g} \sim \text{Gaussian}(\mu, \sigma^2) \quad (2.87)$$

For entry  $k$  of  $\mathbf{g}$ , it defines the probability  $\Pr(g_k)$  to be:

$$\Pr(g_k) = \frac{1}{\sigma_k \sqrt{2\pi}} \cdot \exp\left(-\frac{(g_k - \mu_k)^2}{2\sigma_k^2}\right) \quad (2.88)$$

where  $\mu_k$  is the mean of the distribution, and  $\sigma_k$  is its standard deviation. Often,  $\mu_k$  is set to 0.  $\sigma_k$  is a hyper-parameter that is used to control the amount of noise we want to add. For example, a large  $\sigma_k$  means that the random noise spreads out in a large region centered around  $\mu_k$ , and it is more likely to generate large noise.

Eq. (2.86) is generic and can be applied to almost everywhere in a neural network. Given a layer  $\mathbf{y} = \psi(\mathbf{x} \cdot \mathbf{W} + \mathbf{B})$ , the noise (say  $\mathbf{g}_{\text{input}}$ ) can be added to the input, like this

$$\mathbf{y} = \psi((\mathbf{x} + \mathbf{g}_{\text{input}}) \cdot \mathbf{W} + \mathbf{B}) \quad (2.89)$$

Likewise, the noise (say  $\mathbf{g}_{\text{output}}$ ) can be added to the activation (or output):

$$\mathbf{y} = \psi(\mathbf{x} \cdot \mathbf{W} + \mathbf{B}) + \mathbf{g}_{\text{output}} \quad (2.90)$$

For example, one can simply make noisy inputs (or outputs) for a model and run all hidden layers as usual, or can add random noise to all activations throughout the neural network. While it is common to add random noise to the layer inputs and/or activations in a neural network [Plaut et al., 1986; Holmström and Koistinen, 1992; Bishop, 1995b], another approach to noisy training is to add random noise directly to model parameters or gradients [Graves et al., 2013b; Neelakantan et al., 2015]. For example, the addition of noise to the transformation matrix has the following form:

$$\mathbf{y} = \psi(\mathbf{x} \cdot (\mathbf{W} + \mathbf{g}_w) + \mathbf{B}) \quad (2.91)$$

where  $\mathbf{g}_w$  is the matrix of noise and has the same shape as  $\mathbf{W}$ . Also, we can add noise (say  $\mathbf{g}_{\text{gradient}}$ ) to the gradient of loss for  $\mathbf{W}$ . Let  $\mathbf{s}$  denote  $\mathbf{x} \cdot \mathbf{W} + \mathbf{B}$ . The noisy gradient can be

written as:

$$\begin{aligned}
 \frac{\partial L}{\partial \mathbf{W}} &= \mathbf{x}^T \cdot \frac{\partial L}{\partial \mathbf{s}} + \mathbf{g}_{\text{gradient}} \\
 &= \mathbf{x}^T \cdot \left( \frac{\partial L}{\partial \mathbf{y}} \odot \frac{\partial \mathbf{y}}{\partial \mathbf{s}} \right) + \mathbf{g}_{\text{gradient}} \\
 &= \mathbf{x}^T \cdot \left( \frac{\partial L}{\partial \mathbf{y}} \odot \psi'(\mathbf{s}) \right) + \mathbf{g}_{\text{gradient}} \tag{2.92}
 \end{aligned}$$

The use of noisy gradients has been found to not only be helpful for robust training but also to ease the gradient flow in the network [Gulcehre et al., 2016].

It should be noted that noise is only present during training and the model works without the addition of noise when making predictions on new data. In this sense, many of the regularization methods could fall under the noisy training framework that is used to prevent fitting the training data precisely and enable the predicting system to generalize well on the test data. For example, dropout randomly inactivates some of the activations of a layer so that every neuron is learned to work in a noisy environment. When running on the test data, all the neurons work together as in a usual neural network.

There is an additional advantage with noisy training in that the use of random noise makes “new” training samples. Even for the same sample, different noise could lead to different training results. In other words, we essentially train the model on an infinite number of samples. This idea is also linked to another line of research on training with synthetic data, called **data augmentation**. In simple terms, data augmentation is a set of methods to generate new samples from existing samples. An example is **back-translation** [Senrich et al., 2016a]. When developing a machine translation system from language A to language B, we can first train a reverse translation system (say the B→A system) on the bilingual data. Then, we use the B→A system to translate some additional target-language data to source-language data. This results in new bilingual data where the target-language data is real and the source-language data is synthetic. This new data can be used together with other bilingual data to train the A→B system. In addition to back-translation, there are many data augmentation methods in NLP, including replacing words with synonyms, swapping two words, deleting/inserting words, and so on. Moreover, we can do similar things on feature vectors, such as replacing a word embedding with a similar embedding. Since data augmentation covers a wide variety of topics, we refer the reader to a few survey papers for more information [Feng et al., 2021; Shorten and Khoshgoftaar, 2019].

One last note on data augmentation. Synthetic data can be made for some purpose. A popular idea is **adversarial machine learning**. It generates **adversarial samples** on that a model would make mistakes (call such processes **attacks**) [Szegedy et al., 2014a; Goodfellow et al., 2015]. The model is learned to make correct predictions on these samples, i.e., it defends the attacks. For example, in some cases, the output of a machine translation system would be completely wrong if we change the gender of the subject of the input sentence. For a more robust system, one may train the translation model by using more gender-balanced data, gathered either manually or automatically. But it is not easy to craft samples that look like

normal sentences but can fool the model [Zhang et al., 2020b]. This in turn makes it interesting yet challenging to generate adversarial samples in NLP, since a small change in a sentence (such as word replacement) could lead to something with a very different meaning<sup>9</sup>. The challenge also motivates a thread of research on investigating adversarial samples in NLP [Jia and Liang, 2017; Belinkov and Bisk, 2018; Ebrahimi et al., 2018; Alzantot et al., 2018].

## 2.6 Unsupervised Methods and Auto-encoders

Unsupervised learning is concerned with discovering the underlying patterns in a set of unlabeled data points. A number of problems can be viewed as classical unsupervised learning problems, though we will not discuss them in detail throughout this chapter. For example, data clustering is to find groupings in a collection of data objects, given no supervised signals on what the correct grouping is. Another well-known example is association rule mining. It is often framed as a process of establishing the relationship among sets of data objects. While these problems are indeed covered by unsupervised learning, we will focus on problems of unsupervised representation learning or **feature learning**, that is, a model is learned to map an object from an input space to a low-dimensional feature vector space<sup>10</sup>.

Learning low-dimensional representations has been extensively studied in the context of finding a linear transformation from the original space to the new space. For example, **principal components analysis (PCA)** and its variants try to find a linear mapping function so that a (high-dimensional) data object can be represented as its coordinates along the directions of the greatest variance [Pearson, 1901; Wold et al., 1987]. Here we extend the mapping function to its natural non-linear generation and use neural networks as a solution to the mapping problem.

As with other machine learning models, a neural network is typically learned by optimizing model parameters with respect to some loss function. A considerable challenge with unsupervised learning is that there is no benchmark to signal the learning. A solution to this issue is to resort to non-parametric methods or heuristics (see Chapter 1). However, such methods themselves are not designed to address the learning issue of large-scale neural networks, particularly when a neural network is built up of a huge number of parameters. In unsupervised learning of a neural network, therefore, it is more common to use the “supervision” information from the input data itself. While there are several ways to do this [Hopfield, 1982; Ackley et al., 1985; Dayan et al., 1995; Hinton and Salakhutdinov, 2006], we focus on **auto-encoders** in this section. We choose auto-encoders for discussion because they resemble the general form of supervised models and can be trained via back-propagation.

An auto-encoder is a type of neural networks that tries to reconstruct the input data from its representation. It is inspired by the idea of dimensionality reduction:

---

<sup>9</sup>By contrast, in computer vision, it is much easier to create adversarial samples by making a small change in the input (e.g., pixels), since the input space is continuous and a small input perturbation has very little effect on the whole image.

<sup>10</sup>In addition to learning to represent data objects, this section also covers some topics on the generation of data objects. We will see them in Section 2.6.3.

*High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors.*

– Hinton and Salakhutdinov [2006]

This also develops the idea of representation learning in that the information of an object can be sufficiently represented by a low-dimensional real-valued vector. Typically, an auto-encoder involves a (probably non-linear) dimensionality reduction function (call it an **encoder**) to map the input object to its low-dimensional feature vector representation (call it a **code**). Also, it involves a reverse function (call it a **decoder**) that maps the code back to the object. So, although an auto-encoder is called an “encoder”, it is not just an encoder but a combination of an encoder and a decoder. More formally, let  $\mathbf{x}$  be the input vector of the model, such as a high-dimensional representation of a word. The encoder spits out a vector describing the code or low-dimensional representation of  $\mathbf{x}$ , as follows

$$\mathbf{h} = \text{Enc}(\mathbf{x}) \quad (2.93)$$

where  $\text{Enc}(\cdot)$  is the encoding network.  $\text{Enc}(\cdot)$  is typically a multi-layer neural network and works as a plugged-in for other systems. Thus,  $\text{Enc}(\cdot)$  is a general-purpose model. In subsequent chapters, we will see many examples where encoders are trained and applied as parts of “bigger” systems.

Once we obtain the code, we use the decoder to map it back to the input:

$$\tilde{\mathbf{x}} = \text{Dec}(\mathbf{h}) \quad (2.94)$$

where  $\tilde{\mathbf{x}}$  is the reconstruction of the input, and  $\text{Dec}(\cdot)$  is the decoding network. Given the original input  $\mathbf{x}$  and the reconstructed input  $\tilde{\mathbf{x}}$ , the objective of the auto-encoder is to minimize the discrepancy between  $\mathbf{x}$  and  $\tilde{\mathbf{x}}$ . Suppose that the encoder and the decoder are parameterized by  $\theta$  and  $\omega$ , denoted as  $\text{Enc}_\theta(\cdot)$  and  $\text{Dec}_\omega(\cdot)$ . The training objective over a set of samples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  is defined as

$$\begin{aligned} (\hat{\theta}, \hat{\omega}) &= \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \tilde{\mathbf{x}}^{(i)}) \\ &= \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \text{Dec}_\omega(\mathbf{h}^{(i)})) \\ &= \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \text{Dec}_\omega(\text{Enc}_\theta(\mathbf{x}^{(i)}))) \end{aligned} \quad (2.95)$$

where  $L(\cdot)$  is the loss function that computes the discrepancy between  $\mathbf{x}$  and  $\tilde{\mathbf{x}}$ . It is sometimes called the **reconstruction loss**. Popular loss functions for reconstruction include mean squared

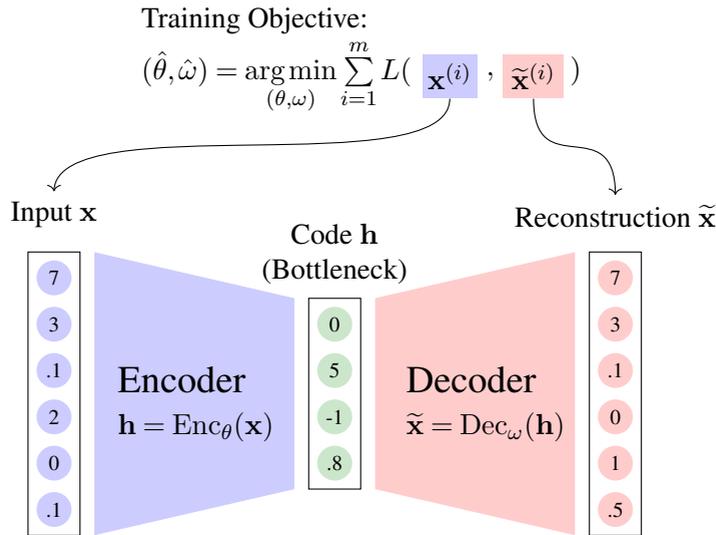


Figure 2.16: An undercomplete auto-encoder with an encoder, a decoder and a bottleneck layer sandwiched between them. An input  $\mathbf{x}$  (left) is transformed into a code  $\mathbf{h}$  (middle) and then a reconstruction  $\tilde{\mathbf{x}}$  (right). The parameters of both the encoder and decoder are optimized by minimizing the discrepancy between the input  $\mathbf{x}$  and the reconstruction  $\tilde{\mathbf{x}}$  on a number of unlabeled samples  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ . On new samples, we throw away the decoder, and use the encoder to generate new codes or representations.

error loss, crossentropy loss, etc.

Putting together the encoder and the decoder, it is tempting to think of a network in which we feed something into the input layer and get back the same thing out of the output layer. The challenge here is that the low-dimensional vector  $\mathbf{h}$  serves as a bottleneck in information flow. There is a risk of information loss in transformation either from  $\mathbf{x}$  to  $\mathbf{h}$  or from  $\mathbf{h}$  to  $\tilde{\mathbf{x}}$ , making it difficult to “copy” the input to the output. Rather, we need to “squeeze” an object from a high-dimensional space to a dense, low-dimensional space, and then “unsqueeze” it from the new space to the original high-dimensional space. A consequence of this squeeze-and-unsqueeze process is that the encoder is forced to compress the data but retain the information as much as possible. So, the auto-encoder discussed here is also called the **undercomplete auto-encoder**, because  $\mathbf{h}$  has a smaller size than  $\mathbf{x}$  and  $\tilde{\mathbf{x}}$ . Figure 2.16 shows an illustration of the undercomplete auto-encoder structure.

Given the loss function  $L(\cdot)$ , the encoder  $\text{Enc}_{\theta}(\cdot)$  and the decoder  $\text{Dec}_{\omega}(\cdot)$ , the parameters  $\hat{\theta}$  and  $\hat{\omega}$  can be optimized by using the gradient descent method as in supervised learning (see Section 2.4.1). When applying the auto-encoder, one can simply drop the decoder and use the encoder as a feature extractor, that is, given a new input  $\mathbf{x}_{\text{new}}$ , we generate a new representation

$$\hat{\mathbf{h}}_{\text{new}} = \text{Enc}_{\hat{\theta}}(\mathbf{x}_{\text{new}}) \quad (2.96)$$

Note that the encoder is not a standalone system but typically works with other models for

a complete working system. For example, we can train an auto-encoder on some sentences and place a Softmax layer on the output of the encoder to build a sentence classifier. The classifier can be further trained on some task-specific data to solve a new problem, such as tagging a sentence with its sentiment polarity. This also makes the application of auto-encoder fall under the general paradigm of pre-training: a sub-model (i.e., an encoder) is first trained on large-scale, task-irrelevant data, and then used as a component of a bigger model on a downstream task.

### 2.6.1 Auto-encoders with Explicit Regularizers

As more complex neural networks are involved, an auto-encoder tends to learn an identity transformation although the bottleneck makes it a bit harder to pass through without information loss. This is what we would ordinarily expect: we could make  $\mathbf{h}$  a surrogate of  $\mathbf{x}$  and decode  $\mathbf{h}$  to something very similar to  $\mathbf{x}$ . On the other hand, learning an exact identity transformation requires a highly complicated model and is prone to overfitting. Fortunately, as with other machine learning models, we can regularize the training by using the methods presented in Section 2.5. One of the most popular methods is adding an explicit regularization term to the loss function. Taking together Eq. (2.74) and Eq. (2.95), we can define the training objective to be

$$(\hat{\theta}, \hat{\omega}) = \underset{(\theta, \omega)}{\operatorname{argmin}} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \operatorname{Dec}_{\omega}(\operatorname{Enc}_{\theta}(\mathbf{x}^{(i)}))) + \alpha \cdot R \quad (2.97)$$

where  $R$  is the regularization term accounting for some prior knowledge we want to impose on training, and  $\alpha$  is its coefficient. A common choice for  $R$  in auto-encoders is the **sparsity penalty** (also known as **sparse auto-encoders**). The simplest way to implement such a penalty is to apply the  $l_1$  or  $l_2$  norm on the code, as follows

$$R_{l_1} = \sum_{i=1}^m \sum_k |h_k^{(i)}| \quad (2.98)$$

$$R_{l_2} = \sum_{i=1}^m \sqrt{\sum_k (h_k^{(i)})^2} \quad (2.99)$$

It is worth noting that, unlike those penalties on model parameters (see Section 2.5.1), the sparsity penalty regularizes the code  $\mathbf{h}$  (or the output of the encoder) to be sparse. The idea of encouraging sparseness in representations stems from **sparse coding** [Olshausen and Field, 1997]. It states that the information of an object is embedded in complex dependencies among the original attributes (or features) of the object. A desirable representation learning system should extract such dependencies and reform them to be a set of independent features. And there should be a small number of these independent features that are active, while the active features vary when we switch to a new object. Note also that, from a Bayesian learning point of view, other penalties in regularized training could be interpreted as priors over models. The sparsity penalty, however, is not a prior because it does not depend on models (or model

parameters) but on the training data [Goodfellow et al., 2016]. In this view, the sparsity penalty should not be treated as a “regularization” term, but simply some distribution over the model’s intermediate states. On the other hand, the sparseness of the code, though not well explained by conventional use of regularization terms, is indeed helpful in many applications of auto-encoders, because it directly models the way of representing the input and imposing “priors” on outcomes of encoders. When considered from an empirical point of view, the sparsity penalty is still thought of as a regularizer that biases the training to certain models.

There are other choices for defining the regularization term  $R$  in addition to Eqs. (2.98–2.99). For example, a way of forcing sparsity is to penalize the cases where the average value of each entry  $h_k$  is far away from a predefined value [Nair and Hinton, 2009]. In case that  $h_k$  chooses values from  $[0, 1]$ , the regularization can be implemented by defining  $R$  as the KL divergence between the average code over a number of samples and the expected code<sup>11</sup>. Let  $\bar{\mathbf{h}}$  denote the average code over  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ , where the value of  $\bar{h}_k$  is the mean of the  $k$ -th variable of the code:

$$\bar{h}_k = \frac{1}{m} \cdot \sum_{i=1}^m \text{Enc}(\mathbf{x}^{(i)})(k) \quad (2.100)$$

Also, let  $\mathbf{q}$  be the expected code, where  $q_k = \tau$  for any  $k$ . If each entry of the average code is viewed as a Bernoulli random variable with mean  $\bar{h}_k$ , and each entry of the expected code is viewed as another Bernoulli random variable with mean  $\tau$ , then the regularization term can be defined as the sum of the KL divergence between  $\bar{h}_k$  and  $q_k$  over all entries:

$$\begin{aligned} R &= \sum_k \text{KL}(\bar{h}_k, q_k) \\ &= \sum_k \tau \cdot \log \frac{\tau}{\bar{h}_k} + (1 - \tau) \cdot \log \frac{1 - \tau}{1 - \bar{h}_k} \end{aligned} \quad (2.101)$$

In this form,  $R$  penalizes the model when  $\bar{\mathbf{h}}$  deviates from  $\mathbf{q}$ .

As another auto-encoder variant, the **contractive auto-encoder (CAE)** tries to improve the robustness of representation by introducing a new regularization term into training [Rifai et al., 2011]:

$$\begin{aligned} R &= \sum_{i=1}^m \left\| \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{x}^{(i)}} \right\|_F^2 \\ &= \sum_{i=1}^m \left\| \frac{\partial \text{Enc}(\mathbf{x}^{(i)})}{\partial \mathbf{x}^{(i)}} \right\|_F^2 \end{aligned} \quad (2.102)$$

where  $\frac{\partial \text{Enc}(\mathbf{x}^{(i)})}{\partial \mathbf{x}^{(i)}}$  (or  $\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{x}^{(i)}}$ ) is the **Jacobian matrix** of the representation<sup>12</sup>, and  $\|\cdot\|_F$  is

<sup>11</sup>In general, we can set all entries of the expected code to  $\tau \in [0, 1]$ . Sparse codes will be preferred if  $\tau$  is close to 0, as features are “inactive” in most cases. By contrast, dense codes will be preferred if  $\tau$  is close to 1.

<sup>12</sup>Suppose that the encoder is a function  $\text{Enc}(\cdot): \mathbf{x} \in \mathbb{R}^{d_x} \rightarrow \mathbf{h} \in \mathbb{R}^{d_h}$ . The Jacobian matrix of  $\mathbf{h} = \text{Enc}(\mathbf{x})$  is

the **Frobenius norm** of a matrix<sup>13</sup>. The contractive penalty helps resist the influence of small perturbations to the input. In the geometric sense, it encourages that the neighborhood relationship holds for output data points if the input data points are neighborhoods, in other words, it forces  $\text{Enc}(\cdot)$  to behave more like a **contraction mapping**<sup>14</sup>, hence the name of contractive auto-encoder.

## 2.6.2 Denoising Auto-encoders

Another source of inspiration for improving the robustness of a model arises from the **denoising** idea: we add noise to the input and then remove it to recover the original input. **Denoising auto-encoders (DAEs)** are such a type of neural networks that marries the idea of auto-encoding with the idea of denoising. First, noise is added to the input vector in a stochastic manner. This can be described as a process of generating a noisy input  $\mathbf{x}_{\text{noise}}$  given the original input  $\mathbf{x}$ :

$$\mathbf{x}_{\text{noise}} \sim \text{Pr}_{\text{noise}}(\mathbf{x}_{\text{noise}}|\mathbf{x}) \quad (2.106)$$

where  $\text{Pr}_{\text{noise}}(\cdot)$  is a distribution for sampling  $\mathbf{x}_{\text{noise}}$ . For example, we can follow the method presented in Section 2.5.5 and take the noisy input as a multivariate Gaussian variable:

$$\mathbf{x}_{\text{noise}} \sim \text{Gaussian}(\mathbf{x}, \sigma^2) \quad (2.107)$$

where  $\text{Gaussian}(\mu, \sigma^2)$  generates  $\mathbf{x}_{\text{noise}}$  via a Gaussian distribution with the mean  $\mu$  and the variance  $\sigma^2$ . Eq. (2.109) introduces an additive noise to the input. Subtracting  $\mathbf{x}$  from the mean, we have

$$\mathbf{x}_{\text{noise}} = \mathbf{x} + \mathbf{g} \quad (2.108)$$

$$\mathbf{g} \sim \text{Gaussian}(0, \sigma^2) \quad (2.109)$$

---

a  $d_h \times d_x$  matrix:

$$\begin{aligned} \text{Jacobian} &= \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \\ &= \begin{bmatrix} \frac{\partial \mathbf{h}}{\partial x_1} & \cdots & \frac{\partial \mathbf{h}}{\partial x_{d_x}} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_{d_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{d_h}}{\partial x_1} & \cdots & \frac{\partial h_{d_h}}{\partial x_{d_x}} \end{bmatrix} \end{aligned} \quad (2.103)$$

<sup>13</sup>For a matrix  $\mathbf{A} \in \mathbb{R}^{d_h \times d_x}$ , the Frobenius norm is given by the equation:

$$\|\mathbf{A}\| = \sqrt{\sum_{i,j} A_{i,j}^2} \quad (2.104)$$

<sup>14</sup>Let  $X$  be a metric space with a metric  $d$ . Given a function  $f(\cdot)$  from  $X$  to  $X$ ,  $f(\cdot)$  is a *contraction mapping* if and only if there is a number  $\epsilon$  such that for any  $x_1, x_2 \in X$ :

$$d(f(x_1), f(x_2)) \leq \epsilon \cdot d(x_1, x_2) \quad (2.105)$$

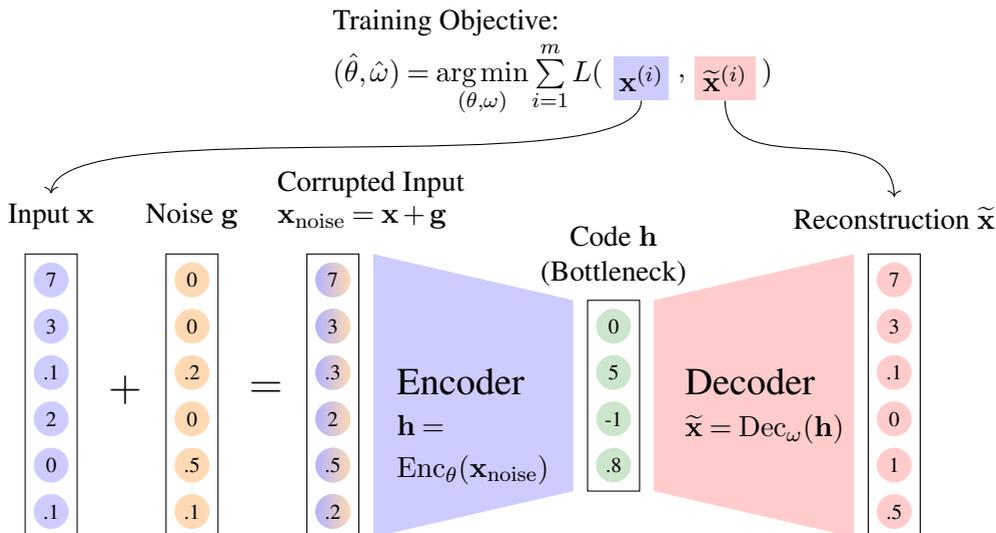


Figure 2.17: The structure of a denoising auto-encoder. An input  $\mathbf{x}$  is first corrupted into a noisy or corrupted input  $\mathbf{x}_{\text{noise}}$ . Then, it is passed through an encoder to form a code  $\mathbf{h}$ . Then, the code is passed through a decoder to form a reconstructed input  $\tilde{\mathbf{x}}$ . The training is performed by minimizing the loss between  $\mathbf{x}$  and  $\tilde{\mathbf{x}}$ . This process is termed “denoising” because it tries to remove the noise from  $\mathbf{x}_{\text{noise}}$  and recover the original input  $\mathbf{x}$ .

Sometimes, this process is called the **corruption** of the input, and  $\mathbf{x}_{\text{noise}}$  is called the **corrupted input**. Aside from additive Gaussian noise, there are a few different ways to corrupt the input [Vincent et al., 2010]. One of the popular methods is to zero some of the entries of  $\mathbf{x}$ . For example, we can set each entry to 0 with a pre-defined probability. This is also called **masking noise**. Another method is to use **salt-and-pepper noise** or **impulse noise** for corruption. It randomly chooses some of the entries, and sets each of them to a minimum or maximum value with a pre-defined probability. Different types of noise are applied to different applications of auto-encoders. For example, the masking noise is popular in training language models, and the salt-and-pepper noise is more commonly used in image processing.

Then, the corrupted input  $\mathbf{x}_{\text{noise}}$  is fed into an encoder-decoder network, and the network produces a reconstructed input  $\tilde{\mathbf{x}} = \text{Dec}(\text{Enc}(\mathbf{x}_{\text{noise}}))$ . The training process is regular. We reuse Eq. (2.95) to minimize the loss of replacing  $\mathbf{x}$  with  $\tilde{\mathbf{x}}$ . Thus, we can rewrite Eq. (2.95) to adapt the objective to the denoising case:

$$(\hat{\theta}, \hat{\omega}) = \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \text{Dec}_{\omega}(\text{Enc}_{\theta}(\mathbf{x}_{\text{noise}}^{(i)}))) \quad (2.110)$$

Eq. (2.110) differs from Eq. (2.95) only in that the input of the auto-encoder is  $\mathbf{x}_{\text{noise}}$  instead of  $\mathbf{x}$ . In other words, we denoise the corrupted input to recover the original input. See Figure 2.17 for the structure of denoising auto-encoders.

Note that both contractive/sparse auto-encoders and denoising auto-encoders can be thought

of as ways to improve the robustness of auto-encoders. Their difference lies in that they regularize the training at different points of the model. Contractive auto-encoders aim at improving the robustness of encoding, that is, the representation is learned to be not so sensitive to small perturbations to the input. Denoising auto-encoders, on the other hand, aim at improving the robustness of reconstruction. It affects both encoders and decoders simultaneously. In some sense, denoising auto-encoders are direct applications of noisy training to auto-encoders (see Section 2.5.5). It is of course difficult to say which models are better. For example, contractive auto-encoders have more direct guidance on learning the representation, which is what we are concerned the most about. The training of denoising auto-encoders, though has an indirect effect on encoding, receives additional denoising signals from the decoder. This offers a new view of robust training: a robust representation can be learned in both where it is generated (the denoising encoder) and where it is applied (the denoising decoder).

### 2.6.3 Variational Auto-encoders

**variational auto-encoders (VAEs)** were not initially proposed to model the encoding problem, although it is termed an “auto-encoder”. They are typically used to generate new data similar to observed data, hence having very different formulations from the classical auto-encoders we mentioned above. In statistics and machine learning, variational auto-encoders are more often viewed as instances of variational Bayesian methods and used to perform efficient statistical inference over latent variables when the posterior probabilities of these variables are intractable [Kingma and Welling, 2014; 2019]. On the other side, variational auto-encoders, implicitly or explicitly, deal with what we do in inducing the underlying representation of an observed object. We therefore involve it in this section for a relatively complete discussion.

We begin with a generative story describing how each data point is generated. Suppose that, for an observed sample  $\mathbf{x}$  in our dataset, there is an unobserved latent variable  $\mathbf{h}$  that describes  $\mathbf{x}$ . Now we intend to develop a probabilistic model to model the generation process of  $\mathbf{x}$ , say, estimating the probability  $\Pr(\mathbf{x})$ . This can be obtained by computing the marginal distribution:

$$\Pr(\mathbf{x}) = \int \Pr(\mathbf{x}, \mathbf{h}) d\mathbf{h} \quad (2.111)$$

where we explicitly introduce the latent variable  $\mathbf{h}$  into the inference of  $\mathbf{x}$ . To solve Eq. (2.111), we use a model  $p_\omega(\mathbf{x}, \mathbf{h})$  to approximate  $\Pr(\mathbf{x}, \mathbf{h})$  (i.e.,  $p_\omega(\mathbf{x}, \mathbf{h}) \approx \Pr(\mathbf{x}, \mathbf{h})$ ), and we have

$$p_\omega(\mathbf{x}) = \int p_\omega(\mathbf{x}, \mathbf{h}) d\mathbf{h} \quad (2.112)$$

where  $p_\omega(\mathbf{x}, \mathbf{h})$  is a probability density function parameterized by  $\omega$ . We replace the left-hand side of Eq. (2.113) with  $p_\omega(\mathbf{x})$  to emphasize that the probability is determined by the model  $p_\omega(\cdot)$ . There are generally many ways to define  $p_\omega(\mathbf{x}, \mathbf{h})$ . Here we can simply think of it as a neural network.

Then, we can rewrite Eq. (2.113) by using the chain rule:

$$p_{\omega}(\mathbf{x}) = \int p_{\omega}(\mathbf{h}) \cdot p_{\omega}(\mathbf{x}|\mathbf{h}) d\mathbf{h} \quad (2.113)$$

where  $p_{\omega}(\mathbf{h})$  is the prior over  $\mathbf{h}$ , e.g., a Gaussian prior. The conditional probability  $p_{\omega}(\mathbf{x}|\mathbf{h})$  describes how likely  $\mathbf{x}$  is observed given the latent variable  $\mathbf{h}$ . To model this generation process,  $p_{\omega}(\mathbf{x}|\mathbf{h})$  is often assumed to be a Gaussian distribution that is parameterized with its mean  $\mu_p$  and variance  $\sigma_p$ :

$$p_{\omega}(\mathbf{x}|\mathbf{h}) = \text{Gaussian}(\mu_p, \sigma_p) \quad (2.114)$$

where  $\mu_p$  and  $\sigma_p$  are determined by a decoding network  $\text{Dec}_{\omega}(\cdot)$  (we will explain later on why it is called “decoding”):

$$(\mu_p, \sigma_p) = \text{Dec}_{\omega}(\mathbf{h}) \quad (2.115)$$

However, Eq. (2.113) is still intractable even though  $p_{\omega}(\mathbf{h})$  and  $p_{\omega}(\mathbf{x}|\mathbf{h})$  are both tractable, because it is impossible to summing over all possible  $\mathbf{h}$ 's. This also leads to an intractable posterior:

$$p_{\omega}(\mathbf{h}|\mathbf{x}) = \frac{p_{\omega}(\mathbf{h}) \cdot p_{\omega}(\mathbf{x}|\mathbf{h})}{p_{\omega}(\mathbf{x})} \quad (2.116)$$

It looks like we are stuck with  $p_{\omega}(\mathbf{x})$  and  $p_{\omega}(\mathbf{h}|\mathbf{x})$ ! Variational auto-encoders address this issue by approximating  $p_{\omega}(\mathbf{h}|\mathbf{x})$  with a tractable posterior  $q_{\theta}(\mathbf{h}|\mathbf{x})$ :

$$q_{\theta}(\mathbf{h}|\mathbf{x}) \approx p_{\omega}(\mathbf{h}|\mathbf{x}) \quad (2.117)$$

where  $\theta$  is the parameter of the new model. Like Eqs. (2.114-2.115),  $q_{\theta}(\mathbf{h}|\mathbf{x})$  is defined as another Gaussian distribution:

$$q_{\theta}(\mathbf{h}|\mathbf{x}) = \text{Gaussian}(\mu_q, \sigma_q) \quad (2.118)$$

$$(\mu_q, \sigma_q) = \text{Enc}_{\theta}(\mathbf{x}) \quad (2.119)$$

where  $\text{Enc}_{\theta}(\cdot)$  is the encoding network that reads  $\mathbf{x}$  and generates the mean and variance of the distribution  $q_{\theta}(\mathbf{h}|\mathbf{x})$ . This is interesting! We now have a feasible path to compute  $\text{Pr}(\mathbf{x})$ : we first sample a latent variable  $\mathbf{h}$  via  $q_{\theta}(\mathbf{h}|\mathbf{x})$ , and then compute  $p_{\omega}(\mathbf{x})$  via the product of

$p_\omega(\mathbf{h})$  and  $p_\omega(\mathbf{x}|\mathbf{h})$ . In this case, the log-scale probability of the observation is defined to be

$$\begin{aligned}
 \log \Pr(\mathbf{x}) &\equiv \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log p_\omega(\mathbf{x}) \\
 &= \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{p_\omega(\mathbf{h}|\mathbf{x})} \\
 &= \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{p_\omega(\mathbf{h}|\mathbf{x})} \cdot \frac{q_\theta(\mathbf{h}|\mathbf{x})}{q_\theta(\mathbf{h}|\mathbf{x})} \\
 &= \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{q_\theta(\mathbf{h}|\mathbf{x})}{p_\omega(\mathbf{h}|\mathbf{x})} + \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})} \quad (2.120)
 \end{aligned}$$

The first term of the right-hand side of Eq. (2.120) is the KL divergence (relative entropy) between  $q_\theta(\mathbf{h}|\mathbf{x})$  and  $p_\omega(\mathbf{h}|\mathbf{x})$ , i.e.,

$$D(q_\theta(\mathbf{h}|\mathbf{x})||p_\omega(\mathbf{h}|\mathbf{x})) = \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{q_\theta(\mathbf{h}|\mathbf{x})}{p_\omega(\mathbf{h}|\mathbf{x})} \quad (2.121)$$

Thus, given  $D(q_\theta(\mathbf{h}|\mathbf{x})||p_\omega(\mathbf{h}|\mathbf{x})) \geq 0$ , we have<sup>15</sup>

$$\begin{aligned}
 \log \Pr(\mathbf{x}) &\geq \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})} \\
 &= \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \left[ \log p_\omega(\mathbf{x}|\mathbf{h}) + \log \frac{p_\omega(\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})} \right] \\
 &= \mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log p_\omega(\mathbf{x}|\mathbf{h}) + D(p_\omega(\mathbf{h})||q_\theta(\mathbf{h}|\mathbf{x})) \quad (2.122)
 \end{aligned}$$

The right-hand side of Eq. (2.122) is a lower bound of the likelihood  $\log \Pr(\mathbf{x})$ . It is also known as the **evidence lower bound (ELBO)**. The first term of the ELBO can be approximately computed by sampling different  $\mathbf{h}$ 's. Also, computing the second term is not difficult because there is an analytical form for  $D(p_\omega(\mathbf{h})||q_\theta(\mathbf{h}|\mathbf{x}))$  if the forms of  $p_\omega(\mathbf{h})$  and  $q_\theta(\mathbf{h}|\mathbf{x})$  are given. Let  $L(\mathbf{x}, \theta, \omega)$  denote the negative ELBO. Then, the training process of a variational auto-encoder can be framed as minimizing  $L(\cdot)$  over a number of observed samples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ :

$$(\hat{\theta}, \hat{\omega}) = \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \theta, \omega) \quad (2.123)$$

Note that, because sampling  $\mathbf{h}$  from  $q_\theta(\mathbf{h}|\mathbf{x})$  is a non-continuous operation,  $\mathbb{E}_{\mathbf{h} \sim q_\theta(\mathbf{h}|\mathbf{x})} \log p_\omega(\mathbf{x}|\mathbf{h})$  is not straightforwardly differentiated. To fit the training of auto-encoders in standard back-prorogation, a common way is to use the so-called **reparameterization trick**. Here we skip the details and refer the reader to a few papers for more information [Kingma and Welling, 2014; Doersch, 2016].

Figures 2.18 illustrates how a variational auto-encoder works. It presents us with a two-step generation process:

<sup>15</sup>The KL divergence between  $p$  and  $q$  is zero only if  $p = q$ , and is positive otherwise.

$$\text{Training Objective: } (\hat{\theta}, \hat{\omega}) = \arg \min_{(\theta, \omega)} \sum_{i=1}^m L(\mathbf{x}^{(i)}, \theta, \omega)$$

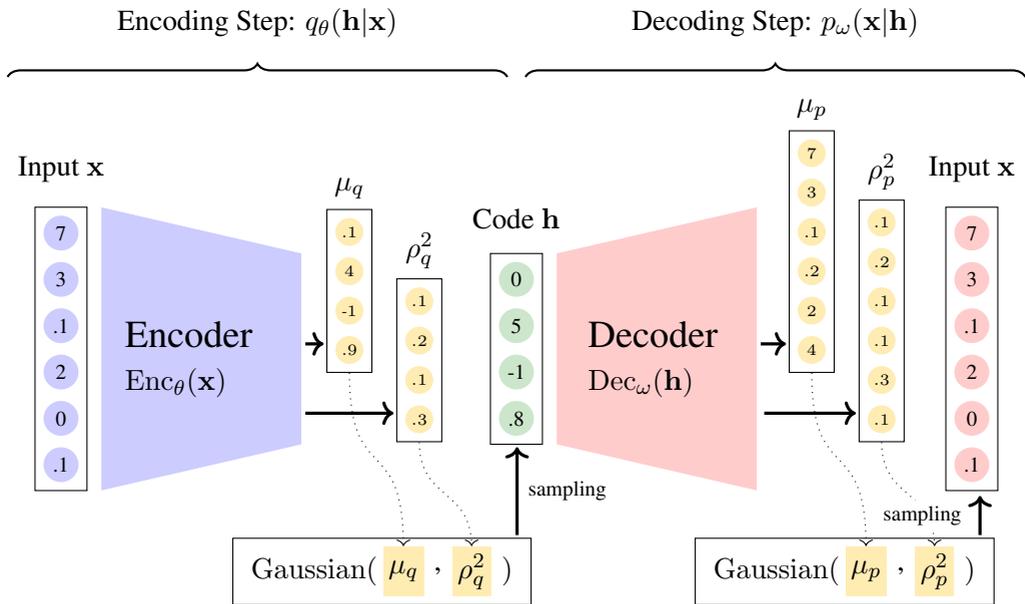


Figure 2.18: The generative story of a variational auto-encoder. For an input sample  $\mathbf{x}$ , we generate a latent variable  $\mathbf{h}$  by using an encoder  $q_{\theta}(\mathbf{h}|\mathbf{x})$ . In the encoding step, a neural network  $\text{Enc}_{\theta}(\cdot)$  is first used to produce the mean and variance of a Gaussian distribution, say,  $\mu_q$  and  $\sigma_q^2$ . The latent variable  $\mathbf{h}$  is then drawn according to  $\text{Gaussian}(\mu_q, \sigma_q^2)$ . After that, we regenerate the original sample  $\mathbf{x}$  by using a decoder  $p_{\omega}(\mathbf{x}|\mathbf{h})$ . In the decoding step, like the generation process in the encoder, a neural network  $\text{Dec}_{\omega}(\cdot)$  is used to generate the mean  $\mu_p$  and variance  $\sigma_p^2$  of  $\text{Gaussian}(\mu_p, \sigma_p^2)$ . The same input  $\mathbf{x}$  is spitted out by sampling from  $\text{Gaussian}(\mu_p, \sigma_p^2)$ .

- **Encoding.** For an input sample  $\mathbf{x}$ , we sample a latent variable  $\mathbf{h}$  from  $q_{\theta}(\mathbf{h}|\mathbf{x})$ . This involves an encoding network  $\text{Enc}_{\theta}(\mathbf{x})$  that generates the mean  $\mu_q$  and variance  $\sigma_q^2$  of the Gaussian distribution  $q_{\theta}(\mathbf{h}|\mathbf{x})$ . The latent variable is then generated by sampling from  $\text{Gaussian}(\mu_q, \sigma_q^2)$ .
- **Decoding.** For the latent variable  $\mathbf{h}$ , we sample the original input  $\mathbf{x}$  from  $p_{\omega}(\mathbf{x}|\mathbf{h})$ . It follows again a Gaussian sampling process: a decoding network  $\text{Dec}_{\omega}(\mathbf{h})$  is used to determine the mean  $\mu_p$  and variance  $\sigma_p^2$  of the distribution.  $\mathbf{x}$  is generated by following  $\text{Gaussian}(\mu_p, \sigma_p^2)$ .

Sometimes,  $q_{\theta}(\mathbf{h}|\mathbf{x})$  and  $p_{\omega}(\mathbf{x}|\mathbf{h})$  are called an “encoder” and a “decoder”, as they try to “map” an input to a representation and then “map” it back to the input. However,  $q_{\theta}(\mathbf{h}|\mathbf{x})$  and  $p_{\omega}(\mathbf{x}|\mathbf{h})$  themselves imply some non-deterministic models, that is, they output the probability density functions of the variables rather than point estimates. An important consequence of

this result is that variational auto-encoders do not tend to find the “best” representation for the input. At first glance it sounds weird as every model we have talked about so far can give a fixed value output. This, however, is the case of the Bayesian inference — we only learn a distribution over possible values of a latent variable. On the empirical side, if you want to obtain something like a good representation, it is fine to just sample a value from that distribution you developed. It would be a high probability that you get a not-so-bad outcome if your model works well [Knight, 2009].

In practice, the main use of variational auto-encoders is in generation but not representation. At test time, provided the optimized parameters  $\hat{\theta}$  and  $\hat{\omega}$ , the encoder (i.e.,  $q_{\hat{\theta}}(\mathbf{h}|\mathbf{x})$ ) is removed, and the decoder (i.e.,  $p_{\hat{\omega}}(\mathbf{x}|\mathbf{h})$ ) works with randomly generated  $\mathbf{h}$ 's. More precisely, we sample a latent variable  $\mathbf{h}_{\text{new}}$  from a Gaussian distribution, and infer a sample  $\mathbf{x}_{\text{new}}$  by  $p_{\hat{\omega}}(\mathbf{x}|\mathbf{h}_{\text{new}})$  as usual. We will see in the subsequent chapters that many NLP problems can be categorized as generation problems where sequential or hierarchical data objects are generated on the condition of some given data objects or latent variables.

## 2.7 Summary

In this chapter we have talked about what a neural network is, as well as a few basic architectures, which are commonly used as building blocks in constructing powerful deep learning systems. Also, we have talked about how to train neural networks, how to regularize the training process, and how to apply neural networks to feature learning in an unsupervised manner.

But neural networks and deep learning are wide-ranging topics and all of our discussions are a little “peek” into them. For a more comprehensive introduction to these topics, Goodfellow et al. [2016]’s book may be a good choice. It also covers several advanced techniques, such as deep structured models and randomized methods, for developing state-of-the-art systems. However, as always, there is a big difference between knowing what a technique is and being fluent with using it in solving real-world problems. So, for practitioners who want to apply neural networks and deep learning in even simple situations, there are a number of books on implementation details of deep learning systems [Géron, 2019; Zhang et al., 2021; Chollet, 2021], and open-source projects that provide code-bases for reference<sup>16</sup>.

In the following chapters, we will dig into how to use neural models to address NLP problems. Along the way, we will see how to learn the representation of words and sentences using the methods we have discussed so far (Chapters 3-4), and how to model different NLP problems by using several interesting neural network-based methods, including the attention mechanism and Transformers (Chapters 5-6), pre-training (Chapter 7), large language models (Chapters 8-10), and so on.

---

<sup>16</sup>URLs to a few popular online tutorials: <https://pytorch.org/tutorials>, <https://keras.io/examples/nlp>, and <https://www.tensorflow.org/tutorials>





# Basic Models

<b>3</b>	<b>Words and Word Vectors</b> .....	<b>123</b>
3.1	Tokenization	
3.2	Vector Representation for Words	
3.3	Count-based Models	
3.4	Inducing Word Embeddings from NLMs	
3.5	Word Embedding Models	
3.6	Evaluating Word Embeddings	
3.7	Summary	
<b>4</b>	<b>Recurrent and Convolutional Sequence Models</b>	
	<b>171</b>	
4.1	Problem Statement	
4.2	Recurrent Models	
4.3	Memory	
4.4	Convolutional Models	
4.5	Examples	
4.6	Summary	
<b>5</b>	<b>Sequence-to-Sequence Models</b> .....	<b>211</b>
5.1	Sequence-to-Sequence Problems	
5.2	The Encoder-Decoder Architecture	
5.3	The Attention Mechanism	
5.4	Search	
5.5	Summary	
<b>6</b>	<b>Transformers</b> .....	<b>269</b>
6.1	The Basic Model	
6.2	Syntax-aware Models	
6.3	Improved Architectures	
6.4	Efficient Models	
6.5	Applications	
6.6	Summary	



# Chapter 3

## Words and Word Vectors

Words are basic units of language [Jackendoff, 1992]. Most language systems that people use to express their feelings and communicate with others involve creating, mixing, and combining words in some way. Before understanding how a word is used in forming larger language units, it is worth first understanding what a word is. This involves two fundamental questions:

- What is the surface form of a word?
- What is the meaning of a word?

But these questions are difficult, of course, because there are no simple rules to describe how a word is formed and how its meaning is defined or induced. While there are a variety of theories to answer these questions in linguistics, NLP researchers are concerned more with two practical issues:

- **Tokenization:** given a string, how to segment it into a sequence of words (also called **tokens**) such that these words can be used as basic units in downstream NLP tasks?
- **Word Representation Learning:** given a corpus, how to learn to represent each word in some countable form, and how to enable NLP models to “compute” on top of this representation?

One goal of this chapter is to show how a sentence is segmented in either a linguistic or statistical manner. Specifically, we describe several approaches to tokenizing a string of characters into words or subwords by heuristic rules or statistical models learned from data. The other goal here is to show how words can be represented as real-valued vectors. In particular, we present modern approaches to learning and evaluating these word vectors. The value of this part is not on drilling on those formulas and models but on showing the core idea of word vector representation which is the basis of many NLP systems. In the next few chapters, we will see a natural generation of this idea to modeling more complicated problems, such as representing sequential and tree-like data.

Chinese	Input: 一直以来，完美的机器翻译是人类的梦想之一。 Output: 一直/以来/， /完美/的/机器翻译/是/人类/的/梦想/之一/。
Japanese	Input: 西日本や海はく晴れて、汗ばむ暑さとなる。 Output: 西日本/や/海/は/く/晴れて/、 /汗ばむ/暑さ/と/なる/。
English	Input: She said, “Deep learning is not the solution to all world’s problems”. Output: She/said/,/“/Deep/learning/is/not/the/solution/to/all/world/’s/problems/”/.

Figure 3.1: Tokenization for different languages (slash = word boundary). For Chinese and Japanese where there are no delimiters between words, tokenization is often called **word segmentation**.

## 3.1 Tokenization

In computer science and related fields, the term *token* can be used in many different ways. Here we simply think of a token as a word in linguistics, although it can be something different (see Section 3.1.4). In NLP, tokenization or segmentation is a task related to **morphological analysis** [Aronoff and Fudeman, 2011]. While morphological analyzers or parsers are generally used to study the internal structure of words, tokenization is concerned with how sentences are broken down into words. It appears that we need to know how words are composed if we want to know how sentences are formed by words. Things are even more interesting because the variety of languages makes it difficult to find a general system to describe the morphology of every language. For example, analytic languages (such as Chinese) have little inflection, and rely on word order to convey meaning. By contrast, synthetic languages (such as French) may have rich inflection and the meaning of a word is highly influenced by morphology.

On the other hand, dividing sentences into smaller linguistic pieces is important in many NLP tasks, even though many of the world’s languages have little morphology. For example, Chinese is a morphologically simple language that has no explicit word boundaries. While it also makes sense to take characters as units in understanding what a Chinese text is talking about, it is more desirable and reasonable to consider larger units in processing the text. Note that, even for languages having delimiters between words, such as English, we still have to tokenize sentences such that they are standardized when serving as the input and/or output of an NLP system.

In this section, we skip the discussion on what exactly a word is in morphology and syntax, but simply view tokenization as a task of adding word or token boundaries to a given string (see Figure 3.1). We will show that a sentence can be broken down into words or tokens in either a heuristic or statistical manner. Note that this process is designed to produce some units that can ease the processing of languages in NLP systems, not necessarily to make strictly linguistic sense.

### 3.1.1 Tokenization via Rules and Heuristics

A common and simple approach to tokenization is to identify every word in a sentence by applying a set of pre-defined rules. In general, these rules are linguistically motivated and reflect our prior knowledge of what the form of a word should be. For example, consider the English example in Figure 3.1. We can define the following rules for tokenizing the sentence:

- Words do not contain spaces. In this sense, we can split the sentence into “word candidates” with space.
- Every word candidate that is made up of English letters only (i.e., *a-z* and *A-Z*) is a word.
- Every punctuation mark (i.e., quote, comma, period, etc.) should be isolated to form a word.
- *'s* is a word, indicating noun possessive.

This might be one of the smallest rule sets we can use in English tokenization. Surely, more rules can be added to cover more linguistic phenomena, e.g., words with dashes, words containing non-English letters, and so on. However, there are no standards to define such a set of rules. In practice, and particularly in NLP applications, we want a minimal set of rules to deal with most problems, and the tokenization is usually implemented by a number of **regular expressions**. Here we will not discuss these rules and regular expressions in detail, but refer the reader to a few textbooks for more details [Lawson, 2003; Friedl, 2006; Jurafsky and Martin, 2008]<sup>1</sup>.

Also, it is common to normalize the text before tokenization so that the input of the tokenizer is canonical. For example, for English and other alphabetic languages, **normalization** or **canonicalization** refers to a process of lowercasing words, normalizing character representation (e.g., Unicode characters), and so on. In addition, we can map different forms of a word to the same form for further generalization of the tokenization. A simple way to do this is to conflate all inflected forms of a word into its base form. In linguistics, the base form of a word is called **lemma**, and the process of mapping words to lemmas is called **lemmatization**. Here are some examples of lemmatization.

*learn* → *learn*  
*learning* → *learn*  
*learns* → *learn*  
*best* → *good*

There are words that correspond to two or more different lemmas (often with different part-of-speeches). In this case, we should select the correct lemma according to the context. In other words, lemmatization is context-dependent.

---

<sup>1</sup>Tokenization scripts can be found in many open-source projects, such as Moses [Koehn et al., 2007] (<https://github.com/moses-smt/mosesdecoder/blob/master/scripts/tokenizer/tokenizer.perl>) and the tokenizers in SacreBLEU (<https://github.com/mjpost/sacrebleu/tree/master/sacrebleu/tokenizers>).

Original	She said, “Deep learning is not the solution to all world’s problems”.
Normalization	she said, “deep learning is not the solution to all world’s problems”.
Tokenization	she/said/,/“/deep/learning/is/not/the/solution/to/all/world/’s/problems/”/.
Lemmatization	she/say/,/“/deep/learning/be/not/the/solution/to/all/world/’s/problem/”/.
Stemming	she/said/,/“/deep/learn/is/not/the/solut/to/all/world/’s/problem/”/.

Figure 3.2: Normalization, lemmatization, and stemming of an English sentence. In normalization, the whole sentence is lowercased. In lemmatization, every word is lemmatized and rewritten as its lemma. In stemming, the suffixes of some words are removed.

Closely related to lemmatization is **stemming**, which represents a word as its **stem**. Like lemmas, a stem is some base form of a word. However, unlike lemmas, a stem is not necessarily a valid word, although there are many words whose lemmas and stems are identical. Another difference from lemmatization is that stemming is performed on individual words, without the need of context for disambiguation. So, stemming is context-independent. There are several efficient algorithms for stemming. A popular one is **suffix stripping** [Porter, 1980]. It simply removes the suffixes *ing*, *ed*, *ion*, etc., like these

*remove* → *remov*  
*removing* → *remov*  
*removal* → *remov*  
*best* → *best*

For more examples, Figure 3.2 shows normalization, lemmatization, and stemming results for an English sentence.

It is worth noting that the above methods are typically implemented using regular expressions, dictionary lookups, and additional heuristics. While in our little exploration here it seems that tokenization is not so difficult, much more work is needed to make it practical. In particular, if we deal with languages with a non-alphabetic writing system, or languages without explicit spacing between words, then tokenization would be a hard problem, and in that case, using simple rules would not be a good strategy. In the following subsections, we will reframe tokenization as a machine learning problem where the way to tokenize or segment sentences is learned from data. These methods are language-independent and can be applied to a wide range of tokenization or segmentation-like problems.

### 3.1.2 Tokenization as Language Modeling

Now let us move to statistical modeling of the tokenization problem. For ease of discussion, in this subsection only languages (or more precisely writing systems) without word boundaries are considered, but the method should be understood to cover other problems where delimiters are used to indicate the end or beginning of a word. Let  $\mathbf{x} = x_1 \dots x_l$  be a string of characters,

and  $\mathbf{y} = y_1 \dots y_m$  be a sequence of words or tokens. We would say that  $\mathbf{y}$  is a tokenization result of  $\mathbf{x}$  if  $\mathbf{y}$  defines a segmentation on  $\mathbf{x}$ . Consider the following Chinese sentence:

$\mathbf{x} =$  机器翻译是人类的梦想之一。

We can define a segmentation on the sentence, for example<sup>2</sup>,

$$\begin{aligned} \mathbf{y} &= \text{机器翻译/是/人类/的/梦想/之一/。} \\ &= \left[ \text{“机器翻译” “是” “人类” “的” “梦想” “之一” “。”} \right] \end{aligned} \quad (3.1)$$

In this way, tokenization can be framed as a problem of mapping  $\mathbf{x}$  to  $\mathbf{y}$ . Given an input string, the output is the most likely segmentation:

$$\begin{aligned} \hat{\mathbf{y}} &= \arg \max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{x}) \\ &= \arg \max_{\mathbf{y}} \log \Pr(\mathbf{y}|\mathbf{x}) \end{aligned} \quad (3.2)$$

Eq. (3.2) describes a prediction model we have been referencing several times in this book. However, the problem we are dealing with is easier because  $\mathbf{y}$  contains the information of  $\mathbf{x}$ , and we can remove the condition from  $\Pr(\mathbf{y}|\mathbf{x})$  in the  $\arg \max$  operation:

$$\begin{aligned} \hat{\mathbf{y}} &= \arg \max_{\mathbf{y}} \log \Pr(\mathbf{y}) \\ &= \arg \max_{\mathbf{y}} \log \Pr(y_1, \dots, y_m) \end{aligned} \quad (3.3)$$

It is easy to check that Eq. (3.3) in fact describes a language modeling problem. There are a few different ways to estimate the joint probability  $\Pr(y_1, \dots, y_m)$ . A simple method is to rewrite  $\log \Pr(y_1, \dots, y_m)$  into a sum of log-scale conditional probabilities:

$$\log \Pr(y_1, \dots, y_m) = \log \Pr(y_1) + \log \Pr(y_2|y_1) + \dots + \log \Pr(y_m|y_1, \dots, y_{m-1}) \quad (3.4)$$

Each conditional probability  $\Pr(y_i|y_1, \dots, y_{i-1})$  can be approximated by

$$\Pr(y_i|y_1, \dots, y_{i-1}) = \Pr(y_i|y_{i-n+1}, \dots, y_{i-1}) \quad (3.5)$$

that is, the generation of  $y_i$  only depends on the  $n - 1$  previous context words. To compute  $\Pr(y_i|y_{i-n+1}, \dots, y_{i-1})$ , we can either use the relative frequency methods or neural networks (see Chapter 2).

Now we can think of tokenization as a supervised learning problem. The process is outlined here:

- Prepare some sentences that are correctly segmented.

---

<sup>2</sup>Following the notation used previously, we use both  $\mathbf{y} = y_1 \dots y_m$  and  $\mathbf{y} = [y_1 \ \dots \ y_m]$  to denote a sequence of variables.

- Learn a language model  $\Pr(\mathbf{y})$  on these labeled sentences.
- For a new sentence, find the “best” tokenization  $\hat{\mathbf{y}}$  that maximizes  $\Pr(\hat{\mathbf{y}})$ , as in Eq. (3.3).

While this procedure follows a standard pipeline of supervised learning, there are several practical issues we have to iron out. First, the language model requires a vocabulary from which  $y_i$  can choose a value, but new words are always around. To handle them, one way is to segment an unknown substring into characters, that is, we treat characters as words if the substring yielding these characters is not contained in the vocabulary. An alternative is to take into account all substrings that are not covered by the vocabulary, and replace them with the <unk> tag. The <unk> trick is widely adopted in state-of-the-art language models and is usually helpful.

Second, the language model described above has a bias towards short sequences because  $\Pr(y_1, \dots, y_m)$  would be large if  $m$  is a small number. A general way to mitigate this bias is to introduce a length reward (or length bonus) to the model, for example,

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \log \Pr(\mathbf{y}) + \lambda \cdot m \quad (3.6)$$

or

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \frac{\log \Pr(\mathbf{y})}{m^\lambda} \quad (3.7)$$

where  $\lambda \cdot m$  and  $m^\lambda$  reward long sequences and  $\lambda > 0$  is a hyperparameter controlling how much we rely on the reward in assessing the goodness of  $\mathbf{y}$ . Interestingly, it is found that the length bias is not a big problem with tokenization in practice because the variance in length is small for those “good” tokenization results. For example, using a unigram language model (i.e.,  $n = 1$ ) without any length reward works well in many real-world applications. We will see a few examples in Section 3.1.4.

Third, performing  $\arg \max$  is difficult because there are exponentially many tokenization candidates. However, the use of language models here enables efficient search algorithms. Consider, for example, applying a unigram language model to tokenization. For the input string  $\mathbf{x}$ , we keep, at each position  $j$  of  $\mathbf{x}$ , a state that describes the probability of the best tokenization on  $x_1 \dots x_j$  (denoted as  $p(j)$ ) as well as the last word of this tokenization. At position  $j + 1$ , we create a new state and compute the probability of the best tokenization on  $x_1 \dots x_{j+1}$  by

$$\begin{aligned} p(j+1) &= \max_{1 \leq i \leq j} p(i) \cdot \Pr(x_{i+1} \dots x_j) \\ &= \max_{1 \leq i \leq j} p(i) \cdot \Pr(w_{[i+1, j]}) \end{aligned} \quad (3.8)$$

where  $\Pr(w_{[i+1, j]})$  is the probability of the word spanning  $x_{i+1} \dots x_j$ . On the algorithmic side, Eq. (3.8) describes a dynamic programming method that has a time complexity of  $O(l^2)$  for an input of length  $l$ . For the final output, we can trace back from the final state and dump the word sequence along the path of the optimal tokenization.

Note that the methods here are generic and can be applied to tokenization for other languages. For example, when applying it to English, we only need a slight update on the format of the input: the input is not a character sequence but a sequence of the smallest possible pieces separated out by punctuation and spaces. For example, for the sentence *Is this Tom's laptop?*, we have

$$\mathbf{x} = [\text{Is this Tom ' s laptop ?}] \quad (3.9)$$

Then, the tokenization process can proceed as in Eqs. (3.2-3.7).

### 3.1.3 Tokenization as Sequence Labeling

One of the major ways by which NLP researchers group together consecutive linguistic pieces is through tagging the sequence with a grouping-inspired label set, often known as **sequence labeling**. Although we limit ourselves here to the problem of grouping characters to words, as we will see in the following chapters, such a method is a good solution to many NLP problems, such as part-of-speech tagging, named entity recognition, and so on. Since the idea of sequence labeling has been discussed in Chapter 1, we present here how it is adapted to the tokenization task.

The label sets used in tokenization are regular. The simplest of these is the “IB” set. The “I” label indicates a linguistic piece inside a word, and the “B” label indicates the beginning of a word. The label set can be enriched by adding the “E” label (i.e., the ending of a word) and/or splitting the “B” label into sub-labels (e.g.,  $B_1$  and  $B_2$  indicate the first and the second linguistic pieces of a word) [Zhao et al., 2006]. Given an input sequence  $\mathbf{x}$  and a tokenization result  $\mathbf{y}$ , transforming  $\mathbf{y}$  to the label sequence is fairly simple. Consider again the example used in the previous subsection. We can label the sequence in different formats:

x:	机	器	翻	译	是	人	类	的	梦	想	之	一	。
y:	<u>机</u>	<u>器</u>	<u>翻</u>	<u>译</u>	<u>是</u>	<u>人</u>	<u>类</u>	<u>的</u>	<u>梦</u>	<u>想</u>	<u>之</u>	<u>一</u>	<u>。</u>
{I,B}	B	I	I	I	B	B	I	B	B	I	B	I	B
{I,B,E}	B	I	I	E	B	B	E	B	B	E	B	E	B
{I,B <sub>1</sub> ,B <sub>2</sub> ,E}	B <sub>1</sub>	B <sub>2</sub>	I	E	B <sub>1</sub>	B <sub>1</sub>	B <sub>2</sub>	B	B <sub>1</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>1</sub>

Since the label sequence can be treated as another form of the tokenization, we can restate the problem as finding the best label sequence given an input:

$$\hat{\mathbf{c}} = \underset{\mathbf{c}}{\operatorname{argmax}} \log \Pr(\mathbf{c}|\mathbf{x}) \quad (3.10)$$

where  $\mathbf{c} = c_1 \dots c_l$  is a label sequence. Many methods have been proposed to model  $\Pr(\mathbf{c}|\mathbf{x})$ . A

classic way is given by rewriting  $\Pr(\mathbf{c}|\mathbf{x})$  using the Bayes' rule:

$$\begin{aligned}\hat{\mathbf{c}} &= \arg \max_{\mathbf{c}} \log \frac{\Pr(\mathbf{x}|\mathbf{c})\Pr(\mathbf{c})}{\Pr(\mathbf{x})} \\ &= \arg \max_{\mathbf{c}} \log \Pr(\mathbf{x}|\mathbf{c}) + \log \Pr(\mathbf{c})\end{aligned}\quad (3.11)$$

In this model,  $\Pr(\mathbf{x})$  is a constant for all  $\mathbf{c}$ 's, and thus can be removed from  $\frac{\Pr(\mathbf{x}|\mathbf{c})\Pr(\mathbf{c})}{\Pr(\mathbf{x})}$  in search.  $\Pr(\mathbf{x}|\mathbf{c})$  is the probability of generating the input  $\mathbf{x}$  (i.e., observations) given the label sequence  $\mathbf{c}$  (i.e., latent variables), and  $\Pr(\mathbf{c})$  is a language model defined on the label sequence. Simplifications are in general required for a tractable model. For example, we can make a **Markov assumption** that the choice of  $c_i$  is dependent only on the choice of  $c_{i-1}$ . This leads to the **hidden Markov model (HMM)** which is widely used in generative modeling for NLP problems.

An alternative method is discriminative modeling. A common idea is to treat sequence labeling as a series of independent classification problems. For example, we can develop a local classifier that conditions the prediction of  $c_i$  on a set of features around position  $i$ . In more sophisticated models, such as **conditional random fields (CRFs)**, the context of the entire sequence can be used in the prediction. While it may be interesting to go more deeply into the details about these sequence labeling models, we simply skip them to make the topic in this section more concentrated. Instead, the reader is referred to [Kupiec, 1992; McCallum et al., 2000; Lafferty et al., 2001] for thorough discussions of how these models are developed and applied. In addition, for a comparison of generative modeling and discriminative modeling, we refer the reader to Chapter 1.

### 3.1.4 Learning Subwords

It is a commonly held belief that words are the basic units in language use. This does not mean that words are the smallest linguistic units. Rather, words can be broken down into smaller pieces that have meanings, such as morphemes. It is this which accounts for the important role of words in the syntactic hierarchy of a language, e.g., words are made up of morphemes, and phrases and sentences are made up of words. It is therefore natural to think of words as distinct components of languages that have some function in forming the structure or meaning of a phrase or a sentence. In NLP, however, viewing sentences as sequences of words is not so desirable sometimes. A problem is that some words are rare, making it difficult to adequately learn a model because of data sparseness. For example, *uncopyrightable* is an English word that rarely occurs. An NLP system may simply recognize it as an unknown word (i.e., an OOV word), although we can get the meaning of this word by decomposing it into parts: *un*, *copy*, *right*, and *able*. Another problem is that linguistics-based tokenization standards somewhat limit the use of computers for automatically learning the way to segment the sentence into units in a machine learning sense. In this case, it is helpful to consider identifying “new” words that are not strictly constrained by linguistics but are better suited to NLP systems.

## 1. Byte Pair Encoding

**Byte Pair Encoding (BPE)** is one of the most successful methods to learn **subword** units from a set of word sequences [Sennrich et al., 2016b]. While the BPE approach stems from data compression [Gage, 1994], it is more often used in NLP as a solution to the open vocabulary problem. The basic idea of BPE is that we repeatedly replace the most frequent pair of bytes in the data to form a new byte. As a result, common bytes are often involved in merging substrings of bytes, and rare bytes are often isolated and considered unique units. The outcome of BPE is a byte vocabulary that can be used to encode new data.

In NLP, a byte can roughly correspond to a character. And each entry of the vocabulary is a character sequence, called a symbol or subword. BPE begins with splitting a given text into a sequence of characters, for example, we can add a space after each occurrence of an English letter or a punctuation mark. This in general results in a very long sequence. While BPE itself has no restrictions on input length, a more common way is to prevent cross-word symbols for efficiency considerations. Thus, we can represent the text as a list of space-separated words, each being associated with the frequency of the word. For example, consider a word list:

```
f l o w # : 2
b l o w # : 2
f l a t # : 1
f l a g # : 4
```

where # is a special symbol indicating the end of a word<sup>3</sup>. From this word list, we can collect an initial vocabulary:

```
f : 7      b : 2
l : 9      a : 5
o : 4      t : 1
w : 4      g : 4
# : 9
```

Then, we count the occurrences of each symbol bigram:

---

<sup>3</sup>Instead of taking # as a separate symbol, another way is to concatenate # with the last character in each word, like this

```
f l o w# : 2
b l o w# : 2
f l a t# : 1
f l a g# : 4
```

where “w#”, “t#”, and “g#” represent characters that occur at the end of a word.

f l : 7	a g : 4
l a : 5	g # : 4
l o : 4	b l : 2
o w : 4	a t : 1
w # : 4	t # : 1

We merge the most frequent symbol bigram “f l” to a new symbol “fl” and replace in the word list each occurrence of “f l” with “fl”:

fl o w # : 2
b l o w # : 2
fl a t # : 1
fl a g # : 4

Accordingly, the symbol “fl” is added to the vocabulary:

f : 7	b : 2
l : 9	a : 5
o : 4	t : 1
w : 4	g : 4
# : 9	fl : 7

Then, this process is repeated again. This time, we merge the symbol bigram “fl a” and create a new symbol “fla”. As such, we have a new word list:

fl o w # : 2
b l o w # : 2
fla t # : 1
fla g # : 4

and a new vocabulary:

f : 7	b : 2	fla : 5
l : 9	a : 5	
o : 4	t : 1	
w : 4	g : 4	
# : 9	fl : 7	

We can run this process a certain number of times. The more times we perform the merge process, the larger the vocabulary is. The entries of the final vocabulary are reordered by symbol frequencies. For example, if we set the number of merge operations to 6, we will have a vocabulary, like this:

l : 9	fla : 5	ow# : 4
# : 9	o : 4	flag : 4
f : 7	w : 4	flag# : 4
fl : 7	g : 4	b : 2
a : 5	ow : 4	t : 1

It corresponds to the word list:

```

fl ow# : 2
b l ow# : 2
fla t # : 1
flag# : 4

```

Having obtained a vocabulary like above, we can apply it to tokenize new words. The subword tokenization follows the same procedure of merging symbol bigrams as that used in building the vocabulary. Given a BPE vocabulary, we first segment the input text into character symbols. Then, we examine each symbol bigram in the sequence, and merge the one that has the highest frequency in the vocabulary. We repeat this operation until there are no further merges. Consider, for example, the following text:

tow a flag

It is first transformed into a character sequence:

t o w # a # f l a g #

By using the BPE vocabulary we have obtained, we can do BPE merging on this sequence, like this

	t o w # a # f l a g #
$f l \Rightarrow fl$	t o w # a # <b>fl</b> a g #
$fl a \Rightarrow fla$	t o w # a # <b>fla</b> g #
$o w \Rightarrow ow$	t <b>ow</b> # a # fl a g #
$ow \# \Rightarrow ow\#$	t <b>ow\#</b> a # fl a g #
...	...
—————→	t ow# a # flag#

This subword sequence can be used as some input and/or output of a downstream NLP task, such as machine translation. Sometimes, we want to map subwords back to words. This is simple: we keep the space after each occurrence of the # symbol, and remove all other spaces and #. Also note that the BPE method we describe here requires word-segmented inputs, that is, we need a pre-tokenizer to roughly tokenize the input sequence into some units. This can be done by using the methods presented in Sections 3.1.1-3.1.3.

## 2. WordPiece

The WordPiece method is very similar to the BPE method in that it first divides the input text into the smallest symbols and then progressively merges pairs of consecutive symbols to form larger symbols [Schuster and Nakajima, 2012]. The difference between them is only in the way of selecting which symbol bigram to merge. In BPE, we merge each time the symbol bigram with the highest frequency. Let  $(x_i, x_{i+1})$  be a bigram in the sequence  $\mathbf{x}$ . The merge rule of BPE can be described as

$$(x_{\hat{i}}, x_{\hat{i}+1}) = \underset{i \in [1, |\mathbf{x}|-1]}{\operatorname{arg\,max}} \operatorname{count}(x_i, x_{i+1}) \quad (3.12)$$

where the function  $\operatorname{count}(x_i, x_{i+1})$  returns the frequency of  $(x_i, x_{i+1})$  in the corpus, and  $(x_{\hat{i}}, x_{\hat{i}+1})$  is the bigram with the highest frequency.

The WordPiece method, instead, adopts a maximum likelihood criterion for bigram selection. More precisely, it merges the bigram so that the likelihood of the data is maximized. This can be formalized as:

$$\begin{aligned} (x_{\hat{i}}, x_{\hat{i}+1}) &= \underset{i \in [1, |\mathbf{x}|-1]}{\operatorname{arg\,max}} \log \frac{\Pr(x_i, x_{i+1})}{\Pr(x_i) \Pr(x_{i+1})} \\ &= \underset{i \in [1, |\mathbf{x}|-1]}{\operatorname{arg\,max}} [\log \Pr(x_i, x_{i+1}) - \log (\Pr(x_i) \Pr(x_{i+1}))] \end{aligned} \quad (3.13)$$

$\log \Pr(x_i, x_{i+1}) - \log (\Pr(x_i) \Pr(x_{i+1}))$  describes the increase in log-likelihood of the text when we replace consecutive symbols  $(x_i, x_{i+1})$  with a single symbol  $x_i x_{i+1}$ <sup>4</sup>. Thus, applications of such a merge rule produce a sequence of coding steps, each of which increases the likelihood a bit on top of the last step. The outcome of this process is a code book (i.e., a vocabulary) by which we can define the most likely code sequence for the given text.

## 3. SentencePiece

Both the BPE and WordPiece methods require that the input text is pre-tokenized in some way. This makes it somewhat complicated to develop a tokenization system. As an alternative, SentencePiece is a more general method that deals with raw texts and considers all characters (including spaces) in tokenization [Kudo and Richardson, 2018]. The main idea of SentencePiece is to scale down a big vocabulary so that the unigram probability of the text is minimized at some level of the vocabulary size<sup>5</sup>, called the **unigram** method [Kudo, 2018].

The unigram method frames subword segmentation as a unigram language modeling problem, resembling the general form of Eqs. (3.3-3.4). Let  $\mathbf{x}$  be a sequence of characters and

<sup>4</sup>In statistics,  $\frac{\Pr(a,b)}{\Pr(a)\Pr(b)}$  is called the **pointwise mutual information** of variables  $a$  and  $b$ . See more details in Section 3.3.1. Another name for this is **information gain**. It can be interpreted by using the Kullback-Leibler divergence or other measures in information theory (see Chapter 1).

<sup>5</sup>The term *vocabulary size* may have different meanings. Here it refers to the number of entries of the vocabulary. Sometimes, on the other hand, it is thought of as the total number of bytes used to store the vocabulary.

$\mathbf{y}$  be a sequence of symbols or subwords yielding  $\mathbf{x}$ . The probability of  $\mathbf{y}$  is given by:

$$\Pr(\mathbf{y}) = \prod_{i=1}^{|\mathbf{y}|} \Pr(y_i) \quad (3.14)$$

Then, we can write the likelihood of  $\mathbf{x}$  in terms of the joint probability of  $\mathbf{x}$  and  $\mathbf{y}$ :

$$\Pr(\mathbf{x}) = \sum_{\mathbf{y} \in Y(\mathbf{x})} \Pr(\mathbf{x}, \mathbf{y}) \quad (3.15)$$

where the sum is over all possible tokenization results  $Y(\mathbf{x})$ . Since  $\mathbf{y}$  can be viewed as a segmentation-annotated version of  $\mathbf{x}$ , the model of  $\Pr(\mathbf{x}, \mathbf{y})$  provides no more information than the model of  $\Pr(\mathbf{y})$  and we have  $\Pr(\mathbf{x}, \mathbf{y}) = \Pr(\mathbf{y})$ . Thus, we can rewrite Eq. (3.15) as:

$$\begin{aligned} \Pr(\mathbf{x}) &= \sum_{\mathbf{y} \in Y(\mathbf{x})} \Pr(\mathbf{y}) \\ &= \sum_{\mathbf{y} \in Y(\mathbf{x})} \prod_{i=1}^{|\mathbf{y}|} \Pr(y_i) \end{aligned} \quad (3.16)$$

Taking this equation, the log-likelihood of a set of strings  $X$  is given by

$$\begin{aligned} \Pr(X) &= \log \prod_{\mathbf{x} \in X} \sum_{\mathbf{y} \in Y(\mathbf{x})} \prod_{i=1}^{|\mathbf{y}|} \Pr(y_i) \\ &= \sum_{\mathbf{x} \in X} \log \left( \sum_{\mathbf{y} \in Y(\mathbf{x})} \prod_{i=1}^{|\mathbf{y}|} \Pr(y_i) \right) \end{aligned} \quad (3.17)$$

If we consider  $-\Pr(X)$  as a loss function, then the task here can be stated as finding the best estimate for each unigram probability  $\Pr(y)$  so as to make  $\Pr(X)$  as large as possible. At first glance this optimization problem looks complicated. Fortunately, there are several powerful tools to solve it. A popular method is to use **the Expectation-Maximization (EM) algorithm** [Dempster et al., 1977], which is commonly used when one tries to find a statistical model that maximizes the likelihood of the data. Note that the EM-based solution to Eq. (3.17) is similar to those for other NLP problems, such as statistical machine translation, and has been well discussed in those contexts. So we refer the reader to [Brown et al., 1993] for details about these methods. In this chapter we just take EM as an off-the-shelf tool to estimate  $\Pr(y)$  given Eq. (3.17).<sup>6</sup>

---

<sup>6</sup>In EM, we can view  $X$  as an observation, and  $\Pr(X|\theta)$  as a statistical model that describes how likely the observation occurs. Here  $\theta$  is the model parameters that we intend to determine. EM is based on an objective of maximum likelihood estimation, that is

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \Pr(X|\theta) \quad (3.18)$$

For the model here, we can view  $\{\Pr(y)\}$  as model parameters. We skip the derivation details about the EM

SentencePiece is essentially a “pruning” method that removes low probability entries from the vocabulary. It starts with a big initial vocabulary  $V$ . For example, we can create the initial vocabulary by enumerating all strings with a length constraint. Typically, cross-word strings are excluded to reduce the vocabulary size. Then, we run the following steps:

- Estimate the probability for each entry  $y$  of  $V$  by optimizing Eq. (3.17).
- Compute the loss for each entry  $y$  of  $V$  via the **remove-one** strategy, that is, the loss is the reduction in the likelihood (see Eq. (3.17)) when  $y$  is removed from the vocabulary.
- Remove a certain percentage of entries of  $V$  with large losses. For example, we keep 80% of the entries, and discard the rest.

The outcome of this process is a new vocabulary as well as the probability assigned to each subword. We can repeat this process a number of times until the vocabulary size is reduced to a desirable level.

SentencePiece differs from BPE and WordPiece in that it considers all possible subword sequences for a given string (see the sum  $\sum_{\mathbf{y} \in Y(\mathbf{x})}$  in Eq. (3.15)). From the machine learning point of view, this can be seen as a way of regularization, that is, we can reduce the risk of overestimating the parameters corresponding to the single-best subword sequence that may have errors. An alternative way is to only consider some of the subword sequences in  $Y(\mathbf{x})$  for the sake of efficiency. For example, we can sample  $k$  subword sequences according to  $\Pr(\mathbf{y})$  to form the candidate set  $Y(\mathbf{x})$ .

Note that the SentencePiece method does not depend on word-separated input sequences. While the BPE and WordPiece methods can also deal with raw text if updated, the SentencePiece method explicitly takes the space and other delimiters as parts of the subwords. See Figure 3.3 for a few tokenization results for *tow a flag*.

Given a learned vocabulary and the corresponding unigram probabilities, we can apply them to deal with a new text. This is in fact a search problem: we find the most likely subword sequence in terms of the unigram probability. As language modeling is a well-studied topic in NLP, many search algorithms are directly applicable to the case here. For example, the

---

estimate of  $\Pr(y)$  but directly present the result. The EM algorithm involves two steps.

- **The Expectation Step** (or the E-step): Given the current estimate of  $\Pr(y)$  (say,  $\Pr_t(y)$ ), we compute the posterior  $\Pr_t(\mathbf{y})$  for each  $\mathbf{y}$  according to Eq. (3.14). Then, we compute the **fractional count** of each subword  $y$  in the vocabulary  $V$ , like this

$$\text{fcount}(y) = \sum_{\mathbf{x} \in X} \sum_{\mathbf{y} \in Y(\mathbf{x})} \left( \Pr_t(\mathbf{y}) \sum_{i=1}^{|\mathbf{y}|} \delta(y, y_i) \right) \quad (3.19)$$

where  $\delta(y, y_i)$  returns 1 if  $y = y_i$ , and 0 otherwise.  $\sum_{i=1}^{|\mathbf{y}|} \delta(y, y_i)$  counts the number of times  $y$  occurs in the subword sequence  $\mathbf{y}$ .

- **The Maximization Step** (or the M-step): Given the fractional counts obtained in the E-step, we re-estimate the unigram probabilities by the equation:

$$\Pr_{t+1}(y) = \frac{\text{fcount}(y)}{\sum_{y' \in V} \text{fcount}(y')} \quad (3.20)$$

The two steps are iterated for a number of rounds until the parameters converge to some values.

subword sequence	unigram probabilities ([subword]:probability)
t/ow_/a/_flag	[t]:0.030 [ow_]:0.002 [a]:0.041 [_flag]:0.001
t/ow/_a_/f/lag	[t]:0.030 [ow]:0.005 [_]:0.113 [a_]:0.093 [f]:0.041 [lag]:0.002
t/ow/_a_/fla/g	[t]:0.030 [ow]:0.005 [_a_]:0.084 [fla]:0.003 [g]:0.027
tow/_a_/f/lag	[tow]:0.001 [_]:0.113 [a_]:0.093 [f]:0.041 [lag]:0.002
t/ow_/a_/flag	[t]:0.030 [ow_]:0.002 [a_]:0.093 [flag]:0.001

Figure 3.3: Different tokenization results for *tow a flag*. Every subword is assigned a probability that is estimated through a unigram language model. Every whitespace is replaced with “\_” for a clear presentation.

methods presented in Section 3.1.2 are straightforwardly applicable here.

## 3.2 Vector Representation for Words

Words have meanings<sup>7</sup>. In the broadest sense, the meaning of a word is the way in which it can be interpreted. This is something behind the surface form of a word but can be understood by language speakers. For example, consider the following lines of text from a poem [Knight, 2018]:

*There was a little sparrow  
Who sat on a wheelbarrow,  
And tweeted to all her friends around.  
A cat with open jaws  
And very pointed claws,  
Spied her as he raced along the ground.*

These words are not merely strings of English letters and punctuation marks but have identifiable meanings that are known by English speakers. For example, “little” means *small in size*, “sparrow” means *a kind of bird*, and “friends” means *people who you like and trust*. From an NLP perspective, a word meaning (or **word sense**) is not just what the word expresses in one’s brain but something computer-readable and computable.

<sup>7</sup>While we have so far discussed several linguistic elements used in NLP, such as subwords, we still use *words* as the basic units in our discussion here. The methods we will present in the remaining part of this chapter could be understood to cover other types of language units one may use in developing NLP systems, including characters, subwords, and so on.

### 3.2.1 One-hot Representation

A natural way to represent word meanings is to use language to describe them. For example, we can find in a dictionary the above words with their ids and meanings. Some of them are<sup>8</sup>:

cat	511	<i>A small animal with fur, four legs, a tail, and claws, usually kept as a pet or for catching mice</i>
her	5220	<i>Used, usually as the object of a verb or preposition, to refer to a woman, girl, or female animal that has just been mentioned or is just about to be mentioned</i>
jaws	6186	<i>The mouth, including the teeth</i>
ground	6402	<i>The surface of the earth</i>
sparrow	8331	<i>A common, small, gray-brown bird</i>
wheelbarrow	9954	<i>A large, open container for moving things in with a wheel at the front and two handles at the back, used especially in the garden</i>

To represent a word, the simplest idea may be to replace it with the id number in the dictionary. In this way, each word representation is a unique number. An equivalent form to this is the **one-hot** representation. It is a vector whose dimensionality is equal to the vocabulary size. In this vector, only the entry corresponding to the word has a value of 1 and all other entries have 0 values. For example, the word *sparrow* can be represented as a one-hot vector based on its id (8331), like this

$$[ 0 \ 0 \ \dots \ 0 \quad 1 \quad 0 \ \dots \ 0 \ 0 ]$$

↑  
id = 8331

### 3.2.2 Distributed Representation

However, it appears that the one-hot representation only provides the “identity” of the word but not the “description” of what the word is. An obvious problem is that every word is orthogonal to other words. This makes it difficult to “compute” the relationship between words because there is no connection among the associated word vectors even though some of the words are thought to be similar in our use of language. Here, our desire is a model in which words are described as countable attributes and the closeness between different words is well explained. A way to do this is to enrich the representation with the word description. Consider again the word *sparrow* for example. In the dictionary, we have its meaning *a common, small, gray-brown bird*. By using the tokenization and normalization methods mentioned in Section 3.1.1, this text can be transformed into a sequence of words

$$[ a \ common \ , \ small \ , \ gray \ - \ brown \ bird ]$$

<sup>8</sup>All these words and their meanings are found in <https://dictionary.cambridge.org/>.

Then, we vectorize this sequence using the bag-of-words model (see Chapter 1), leading to a new vector of numbers

$$\begin{array}{cccccccccccccccc}
 [ & 0 & 0 & \dots & 1 & \dots & 0 & 0 & ] \\
 & & & & \uparrow & & & & \\
 & & & & , & & - & & a & & bird & & brown & & common & & gray & & small & & & & 
 \end{array}$$

where the value of an entry is 1 if the corresponding word is present, and 0 otherwise. This way enables the sharing of content among words. We would say that two words are similar if they have overlaps in their word vectors. Consider a new word *cuckoo*. We can find its meaning in a dictionary, e.g., *a grey bird with a two-note call that sounds similar to its name*. It is easy to know that *sparrow* and *cuckoo* are two words that share something similar because they both mark the “bird” dimension as 1 and the vector similarity between the two word vectors is greater than 0<sup>9</sup>.

Treating words as vectors of numbers offers a general tool to represent words in various different ways. We do not even have to explain a word vector from the viewpoint of semantics. For example, we can introduce a new dimension into the vector to mark if the word belongs to some syntactic category. In a broad sense, we can define an arbitrary function on each entry of the vector and view the function’s output as a feature describing the word. For example, a simple improvement to the above representation is to use a function counting the occurrences of a word instead of the binary-valued function marking the presence of the word. More feature functions can be found in Section 3.3.

Note that it is not necessary to constrain the feature functions to forms that make linguistic sense although linguistically motivated designs of the feature functions are usually of interest to NLP researchers. A more general form for word representation is simply a real-valued, multi-dimensional vector. It is often called the **distributed representation** of a word, or the **word embedding**. For example, the word *sparrow* can be represented as a vector like this

$$\left[ 1.9 \quad -7 \quad 3 \quad -1.2 \quad \dots \quad 2.01 \quad -2.05 \right]$$

In the machine learning point of view, this vector can describe some underlying attributes of a word. These attributes may not be explainable in human understanding but can be learned from data. One of the challenges in learning such a representation is that one can hardly measure the goodness of a vector. In general, it makes no sense to ask whether the distributed representation of a single word is good or not. Rather, we would like to know if the representations of a group of words are well behaved. For example, it is a common belief that similar words should have similar representations. So, the relationship between words is often thought of as some “distance” between the word representations in a vector space. This leads to a number of methods to visualize and evaluate word representations. In Section 3.6, we will give a more detailed discussion about these issues.

<sup>9</sup>The similarity of two vectors can be measured by the cosine of the angle between them.

On the other hand, word representations typically do not work alone in NLP systems but are used as some intermediate states of a model. A standard approach in NLP, to learn distributed representations of words, is to take it as a by-product of training a “big” system. That is, the representation model works as a component of a system, and is optimized together with other components when the system is trained in some way. This inspires a promising paradigm of representation learning: the representation model is learned as a sub-model in an easy-to-train system, and can be used as a plug-in for a completely different system. In neural language modeling, for instance, we can force the model to map each input one-hot word vector into a real-valued, low-dimensional distributed representation. These distributed representations are fed into a neural network that predicts a probability distribution over the vocabulary. The mapping function or embedding function is trained so as to minimize the loss of the language model on some data (see Section 3.4). When applying the learned embedding function, we drop all other parts of the language model and use the function to generate the distributed representation for each word in downstream tasks. An alternative strategy is to specifically tailor the model to the word representation learning problem. Systems of this kind are typically not designed to deal with standard NLP problems, but with an emphasis on specific problems in word representation learning, such as explicitly modeling the relationship between words (see Section 3.5).

### 3.2.3 Compositionality and Contextuality

While we restrict our discussion to word representation learning in NLP, studying the meanings of words is a traditional sub-field of linguistics. In **lexical semantics**, for instance, researchers are concerned with how word meanings are defined and used, and how these meanings form the sentence meanings. In fact, the task of learning to represent words does not concern itself with the issue of semantics in linguistics. Instead, it provides machine learning approaches to transforming linguistic units into computer-friendly forms. However, the semantics issue is critical when one understands and uses a language. It is therefore still worth considering semantics and related problems in the design of word representation models. For example,

- **Compositionality.** Compositionality is a common concept in semantics, logic and related fields. It often comes out with **the principle of compositionality**:

*The meaning of a complex expression is determined by its structure and the meanings of its constituents.*

– Szabó [2020]

This offers a useful tool to describe how the meaning of a big thing is built up from the meanings of its parts. The principle of compositionality is fundamental and exists everywhere in the language world. For example, when you see the phrase *white cat*, it is easy to know its meaning in terms of the meanings of the constituent words *white* and *cat*. Another example at a higher level of language use is compound sentences. A compound

sentence forms its meaning by simply connecting multiple independent clauses with conjunctions. Note that the principle of compositionality is not a simple rule by which we use to describe how a big item is made up of smaller ones, although researchers have tried to define it formally [Montague, 1974]. There are even disagreements and debates on how this principle is interpreted and how it is adequately modeled by semantical theories. Still, if we focus on NLP problems and set aside the theoretical part of linguistics, compositionality is a very useful property that one can make use of in system design and evaluation. Sometimes, if one finds that a problem is compositional, it implies that there are many good methods to address it because a complex thing can be divided into smaller and easier things. For word representation learning, we may wish that the resulting word representations exhibit some compositionality, in response to the compositional nature of language. In Section 3.6, we will see a few examples, e.g., the representations learned by neural networks show meaningful results under linear algebraic operations, though the models are themselves non-linear. However, on the other hand, the principle of compositionality is not the principle of everything. There are many situations in which compositionality is not held, such as collocations and idioms. In this case, natural languages are non-compositional. This explains why the NLP problem is so challenging.

- **Contextuality.** Contextuality is some sort of non-compositionality. It states that a word may have multiple possible meanings and the “true” meaning is determined by looking at the context preceding and/or following this word. For example, consider the following sentences<sup>10</sup>

*They sat round the dinner table, arguing about politics.*

*Come to the table everybody - supper's ready.*

*He came in with four shopping bags and dumped them on the table.*

*The table can help you evaluate the potential risks of investing in the Fund.*

*Building societies dominate the best-value tables for mortgages.*

*This table represents export sales.*

In these example sentences, *table* is a **polysemy** with two meanings:

Sense 1: *a flat surface used for putting things on.*

Sense 2: *an arrangement of items in rows, or columns, or blocks.*

In other words, *table* is an ambiguous word. This ambiguity would be eliminated if we consider the surrounding words. For example, when *table* follows *dinner*, it is easy to figure out that it refers to sense 1. The ambiguity also exists when a word stems from a few different forms or lexemes (call it a **homonymy**). For example, *bear* can be either a

---

<sup>10</sup>All these sentences are from <https://dictionary.cambridge.org/dictionary/english/table>

verb or a noun. Disambiguating a word for a given set of word senses has been studied for decades in NLP and is commonly known as **word sense disambiguation (WSD)** [Kelly and Stone, 1975]. However, the word representation problem discussed here is more challenging because we usually do not have a pre-defined set of word senses in hand. We instead want a contextual representation model that can generate a word representation dependent on its context. Thus, it is important to take the idea that the meaning of a word may not be constant. This makes the problem somewhat different from what we discussed at the beginning of the section, as we no longer have a lookup table for word representations, but a model that produces different representations of a word in different contexts.

In the remaining sections of this chapter, we focus on learning vector representations of words from their distributions in language use. We leave the discussion on the contextual models for learning dense word representations to Chapters 4-6.

## 3.3 Count-based Models

We have framed the induction of word meanings as a problem of learning word vectors. In this section, we proceed by assuming that the meaning of a word is determined by the environment where the word is used. This is usually stated as the **distributional hypothesis** — words are semantically similar if they appear in similar contexts [Harris, 1954; Firth, 1957]. A word representation learned under this hypothesis is also called the **distributional word representation** or **distributional representation**<sup>11</sup>. To ease the reading, however, we will still use the terms *word vector* and *word representation* throughout this book. Next, we introduce several methods for modeling the distribution of words in texts, and then offer some refinements.

### 3.3.1 Co-occurrence Matrices

In **distributional semantics**, words are represented with semantic models that consider various aspects of the context. These models differ in how the context of a word is modeled, for example, how large the context is considered, how each occurrence of a word is counted, how the dimensionality of a distribution is defined, and so on. In this section we assume, as in most models used in NLP, that word representations are learned from a collection of documents.

A way to view a document is as a very simple way of decomposing it into a set of unordered words. Then we can think of each occurrence of these words as an independent context indicator. In this way, the distribution of a word in its context can be described as the number of times the word co-occurs with the context words. We can do this by building a

---

<sup>11</sup>It should be noted that *distributional representation* and *distributed representation* are two different concepts. A distributional representation refers to a representation that describes the distribution of language items in language use. A related term is *non-distributional representation* which means something that is obtained from lexical databases, such as the interpretation of a word in a dictionary. On the other hand, a distributed representation refers to a vector of variables corresponding to some underlying attributes of a language item. In contrast to distributed representation, a one-hot representation just describes the word symbol.

co-occurrence matrix where a cell counts the number of co-occurrences of a row item and a column item. Consider, for example, the following documents<sup>12</sup>:

- Doc 1 *A berry is a small, pulpy, and often edible fruit.*
- Doc 2 *In botanical terminology, a berry is a simple fruit with seeds and pulp produced from the ovary of a single flower.*
- Doc 3 *The term "banana" is also used as the common name for the plants that produce the fruit.*
- Doc 4 *Banana seeds are large and hard and spiky and liable to crack teeth.*
- Doc 5 *A banana is an elongated, edible fruit - botanically a berry - produced by several kinds of large herbaceous flowering plants in the genus Musa.*

For each pair of words, we collect the total number of times they co-occur in these documents, leading to a matrix, called the **word-word co-occurrence matrix** or **term-term co-occurrence matrix**. Here is a subset of the matrix for the above documents.

	<i>flowering</i>	<i>fruit</i>	<i>herbaceous</i>	...	<i>often</i>	<i>plants</i>	<i>seeds</i>
<i>berry</i>	1	3	1	...	1	1	1
<i>terminology</i>	0	1	0	...	0	0	1
<i>common</i>	0	1	0	...	0	1	0
<i>teeth</i>	0	0	0	...	0	0	1
<i>banana</i>	1	2	1	...	0	2	1
<i>simple</i>	0	1	0	...	0	0	1
<i>and</i>	0	2	0	...	1	0	2

In the matrix, each row word is associated with a word vector of  $|V|$  entries. The numbers in the entries describe how often the row word co-occurs with different context words, that is, how a given word is distributed in different “contexts”. In a geometric sense, if two words have similar distributions in co-occurring with the same group of context words, then the angle between the word vectors would be small<sup>13</sup>. For example, if we think of these words as vectors in a vector space, *berry* is closer to *banana* than *teeth* (see Figure 3.4). This geometric intuition is the basis of many representation models. More examples will be given in Chapters 4 and 5.

A problem with this method is that the distance between words is not taken into account although the correlation is not that strong when the context word is distant. A simple solution is to constrain context words in a window, called the **context window** or window for short [Lund and Burgess, 1996]. For example, for each word in a document, we only count the -2 and +2 words surrounding it (i.e., a window of size 5).

<sup>12</sup>The texts are from Wikipedia.

<sup>13</sup>The angle between two vectors does nothing with the lengths of the vectors. If the vectors are normalized in some way (e.g., by vector norm), similar vectors mean that most entries of the two vectors have similar values.

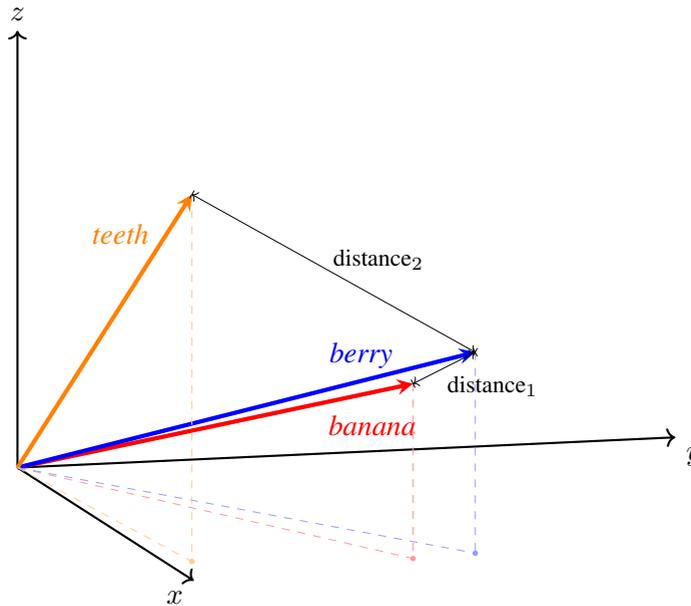


Figure 3.4: Word vectors in a vector space that is built from the word co-occurrence statistics on the English data from WMT 2012. All the vectors are normalized and represented as arrows. For visualization, we project these vectors from a high-dimensional space to a 3-dimensional space via principal component analysis. As expected, *berry* is closer to *banana* than to *teeth*.

Note that the word vectors learned by the bag-of-words model in Section 3.2 is a special instance of the co-occurrence matrix. In that example, we only have one document from which we collect context words. For each entry of a word vector, an indicator function is used to mark the presence of the context word. In addition to the indicator and counting functions, there are other choices for computing word vectors by examining the co-occurrence of words. In practice, the value of an entry of a word vector can be thought of as the degree of the correspondence between words. If two words are correlated with each other in some context, a feature function may assign a score between them in any manner. This score does not necessarily have to be a count, but can be an arbitrary real number. As such, the problem can be stated as measuring the association strength between words. It is common practice to define such a measure on the basis of correlation models. In statistics, correlation describes to what extent two variables are associated, measured by **correlation coefficients**. Common correlation coefficients include the Pearson correlation coefficient (Pearson's  $r$ ), the Spearman's rank correlation coefficient (Spearman's  $\rho$ ), and so on<sup>14</sup>. In NLP, a widely used measure is the **pointwise mutual information (PMI)** [Church and Hanks, 1990]. Let  $a$  and  $b$  be two words. The mathematical form of PMI is given by

$$\text{PMI}(a, b) = \frac{\Pr(a, b)}{\Pr(a)\Pr(b)} \quad (3.21)$$

<sup>14</sup>Some of the correlation coefficients assume certain distributions of the data. For example, the Pearson correlation coefficient is calculated based on two variables following normal distributions.

where  $\Pr(a, b)$  is the joint probability of  $a$  and  $b$  co-occurring, and  $\Pr(a)$  (or  $\Pr(b)$ ) is the probability of  $a$  (or  $b$ ) occurring. These probabilities can be simply estimated on the texts by the relative frequency method<sup>15</sup>. Given a word  $a$  and a vocabulary of context words  $\{b_1, \dots, b_{|V|}\}$ , the PMI-based word vector of  $a$  is written as

$$e(a) = \left[ \text{PMI}(a, b_1) \quad \dots \quad \text{PMI}(a, b_{|V|}) \right] \quad (3.22)$$

Correlation coefficients are generally used to test whether two variables are (linearly) related. So, an alternative method is to define an entry of the word vector as the outcome of a test. For example, the entry  $(a, b)$  chooses a value of 1, if the correlation coefficient between words  $a$  and  $b$  is larger than a threshold, or the correlation of words  $a$  and  $b$  is sufficiently supported by hypothesis testing.

However, modeling words as vectors of correlation scores between words somewhat limits the scope of contextual information one may use. Another idea for word vectorization is to consider each document as a whole and establish the relationship between words and documents. We can do this by using the **word-document co-occurrence matrix** or **term-document co-occurrence matrix**. For example, for the abovementioned documents, we can build a matrix, like this

	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5
<i>berry</i>	1	1	0	0	1
<i>terminology</i>	0	1	0	0	0
<i>common</i>	0	0	1	0	0
<i>teeth</i>	0	0	0	1	0
<i>banana</i>	0	0	1	1	1
<i>simple</i>	0	1	0	0	0
<i>and</i>	1	1	0	2	0

In the matrix, the value of entry  $(a, d)$  is defined to be the number of times the word  $a$  occurs in the document  $d$ , giving the strength of the relationship between  $a$  and  $d$ . This is commonly called the **term frequency (TF)** of  $a$  in  $d$  (denoted by  $\text{tf}(a, d)$ ). Also, we can use a 0-1 indicator function to mark the presence of the word occurrence (see Section 3.2). See Table 3.1 for a few variations of the TF weighting function.

As a co-occurrence matrix, each row of the above matrix is the vector representation of the row word. In addition, each column is a vector representation of a document. Recall the bag-of-words model used in the text classification problem mentioned in Chapter 1. The word-document co-occurrence matrix is basically the same thing as the bag-of-words model

<sup>15</sup>A problem with PMI is that the measure becomes unstable when the words are rare. For example, if a very rare word happens to appear in a document, the PMI value of this word and any other word in this document would be unreasonably large.

Entry	Mathematical form
Binary	$\text{tf}(a, d) = \begin{cases} 1 & a \text{ occurs in } d \\ 0 & \text{otherwise} \end{cases}$
Count	$\text{tf}(a, d) = \text{count}(a; d)$
Exponential Count	$\text{tf}(a, d) = \text{count}(a; d)^\alpha$
Log-scale Count	$\text{tf}(a, d) = \log(1 + \text{count}(a; d))$
Normalized Count (or Frequency)	$\text{tf}(a, d) = \frac{\text{count}(a; d)}{\sum_{a'} \text{count}(a'; d)}$

Table 3.1: Functions of the term-frequency weighting scheme.  $\text{count}(a; d)$  counts the occurrences of the word  $a$  in the document  $d$ .

where the ordering of words is ignored but the word counts matter. Here we perform document vectorization via this model on a collection of documents.

### 3.3.2 TF-IDF

The modeling of word-document associations is known to be important for many NLP tasks. An improvement on using word-document relationships to build word vectors and document vectors simultaneously is the **term frequency-inverse document frequency (TF-IDF)** method. Given a set of documents  $D$ , the TF-IDF weighting scheme assigns a score to each word-document pair  $(a, d)$  by the equation

$$\text{tfidf}(a, d, D) = \text{tf}(a, d) \cdot \text{idf}(a, D) \quad (3.23)$$

where

- $\text{tf}(a, d)$  is the term frequency (see Table 3.1). When  $\text{tf}(a, d)$  is large, the word  $a$  is a good indicator for the document  $d$ . In contrast, when  $\text{tf}(a, d)$  is small, the word-document association is not that strong.
- $\text{idf}(a, D)$  is the **inverse document frequency (IDF)**. It is developed based on the fact that common words across documents are less informative. For example, for a collection of documents on sports, it is likely to see *player* and *players* in most documents. In this case, the words *player* and *players* are less interesting in discriminating different documents or contexts. Let  $\text{df}(a, D)$  be the number of documents in  $D$  containing the word  $a$ . A common form of  $\text{idf}(a, D)$  is given by

$$\text{idf}(a, D) = \log \frac{|D|}{\text{df}(a, D)} \quad (3.24)$$

Eq. (3.25) would penalize a word if it more often appears in the collection of documents.

Similarly, we can have a smoothed version of  $\text{idf}(a, D)$ , like this

$$\text{idf}(a, D) = \log \frac{|D|}{\text{df}(a, D) + 1} + 1 \quad (3.25)$$

Having the TF-IDF feature function in hand, we can build a word-document co-occurrence matrix for a given collection of documents, that is, the value of the entry  $(a, d)$  of the matrix is  $\text{tfidf}(a, d, D)$ . Then, as described in the last subsection, we can treat a row of the matrix as the vector representation of the row word. Note that, traditionally, the TF-IDF method and word-document co-occurrence matrices are often used in document representation. For example, one can represent a query and a number of documents as the TF-IDF (column) vectors in an information retrieval system. This allows us to look at how much the query matches each of these documents via vector similarity. However, the vector space models in information retrieval are beyond the scope of this chapter, but the reader can refer to related textbooks for greater coverage of this topic [Manning et al., 2008; Butcher et al., 2016].

### 3.3.3 Low-Dimensional Models

Co-occurrence matrices are often high dimensional. Suppose, for example, that there is a vocabulary of 20,000 unique words and a collection of 10,000,000 documents. Then, a word-document co-occurrence matrix has  $20,000 \times 10,000,000 = 2 \times 10^{11}$  entries. However, if we consider the computational burden of such a model, it would be hard to imagine that a word is represented as a 10,000,000-dimensional vector and a document is represented as a 20,000-dimensional vector. Instead, we expect that the representation of a word (or a document) requires only a reasonably small number of features. In this subsection, we discuss some standard approaches to transforming words (or documents) into lower-dimensional representations from the co-occurrence matrices. Most of these approaches have been well studied in the literature and have been successfully applied in several disciplines [Barber, 2012; Wright and Ma, 2022]. So we do not dive into the mathematical details behind them, but show how to apply them in the context of learning word (or document) vectors.

#### 1. Latent Semantic Analysis

In NLP, **latent semantic analysis (LSA)** is a method of seeking the latent semantic structure behind the word-document associations [Deerwester et al., 1990; Landauer et al., 1998]<sup>16</sup>. It assumes that either words or documents can be represented as low-dimensional vectors that are distilled from the co-occurrence matrix, preserving the property of the original vector space model, e.g., the angle between vectors is small for similar words.

More specifically, LSA factorizes the co-occurrence matrix into a matrix for word representation, a matrix for document representation, and a third matrix connecting the first two matrices. Mathematically, this can be framed as a **singular value decomposition (SVD)** process [Stewart, 1993]. Let  $\mathbf{M} \in \mathbb{R}^{|V| \times |D|}$  be a co-occurrence matrix over a vocabulary  $V$

<sup>16</sup>Latent semantic analysis is also called **latent semantic indexing (LSI)**. This term is more often used in information retrieval and related fields.

and a document set  $D$ . The SVD produces a factorization of  $\mathbf{M}$ , like this

$$\mathbf{M} = \mathbf{P}\Sigma\mathbf{Q}^T \quad (3.26)$$

where  $\mathbf{P} \in \mathbb{R}^{|V| \times r}$ ,  $\Sigma \in \mathbb{R}^{r \times r}$  and  $\mathbf{Q}^T \in \mathbb{R}^{r \times |D|}$ . In this factorization, the representation model is isolated into two terms  $\mathbf{P}$  and  $\mathbf{Q}^T$  so that both of them are **semi-unitary** (or **semi-orthogonal** in our case)<sup>17</sup>, that is, the columns of either  $\mathbf{P}$  or  $\mathbf{Q}$  are **orthogonal vectors**. Thus, these columns form an orthogonal basis of  $\mathbb{R}^r$ , where  $r$  is the rank of  $\mathbf{M}$ . This means that we use a “minimum” number of dimensions of data to represent  $\mathbf{M}$ .  $\Sigma$  is a diagonal matrix:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_r \end{bmatrix} \quad (3.27)$$

The diagonal entries  $\{\sigma_1, \dots, \sigma_r\}$  are all non-negative real numbers, and are called the **singular values** of  $\mathbf{M}$ . Typically,  $\{\sigma_1, \dots, \sigma_r\}$  are arranged in descending order (i.e.,  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r$ ). Thus, SVD is unique for the given matrix  $\mathbf{M}$ . If we write  $\mathbf{P}$  as a sequence of column vectors (call them **left-singular vectors**)

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1, \dots, \mathbf{p}_r \end{bmatrix} \quad (3.28)$$

and  $\mathbf{Q}^T$  as a sequence of row vectors (call them **right-singular vectors**)

$$\mathbf{Q}^T = \begin{bmatrix} \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_r^T \end{bmatrix} \quad (3.29)$$

then we can write  $\mathbf{M}$  as

$$\mathbf{M} = \sum_i^r \sigma_i \mathbf{p}_i \mathbf{q}_i^T \quad (3.30)$$

For representing words, we can think of  $\mathbf{p}_l$  as the values of a feature function over all the entries of the vocabulary  $V$ . Then, we describe a word  $a_i$  as an  $r$ -dimensional feature vector  $\mathbf{e}_i$  in which the  $l$ -th feature is the  $i$ -th entry of  $\mathbf{p}_l$ . In other words, the vector representation of  $a_i$  is

$$\mathbf{e}_i = \begin{bmatrix} p_1(i) & \dots & p_r(i) \end{bmatrix} \quad (3.31)$$

<sup>17</sup>A non-square matrix  $\mathbf{X}$  is semi-orthogonal if and only if  $\mathbf{X}\mathbf{X}^T = \mathbf{I}$  or  $\mathbf{X}^T\mathbf{X} = \mathbf{I}$ .

Similarly, the vector representation of a document  $d_j$  can be written as

$$\mathbf{h}_j = [q_1(j) \ \dots \ q_r(j)] \quad (3.32)$$

In this way, we have two separate representation models for words and documents:  $\mathbf{P}$  deals with word representations and  $\mathbf{Q}$  deals with document representations. Thus, we can take  $\mathbf{M}$  as a product of these representation models, like this

$$\begin{aligned} \mathbf{M} &= \mathbf{P}\Sigma\mathbf{Q}^T \\ &= \begin{matrix} \text{words} \end{matrix} \begin{bmatrix} \mathbf{e}_1 \\ \vdots \\ \mathbf{e}_{|V|} \end{bmatrix} \begin{bmatrix} \sigma_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_r \end{bmatrix} \begin{matrix} \text{documents} \\ \left[ \mathbf{h}_1^T \ \dots \ \mathbf{h}_{|D|}^T \right] \end{matrix} \end{aligned} \quad (3.33)$$

In practice, the rank  $r$  is usually much smaller than  $|V|$  and  $|D|$ . Thus, we have, for each word (or each document), a new representation whose dimensionality is much smaller than the representation contained in the co-occurrence matrix. A further improvement can make use of the  $r^*$  largest singular values (i.e.,  $\{\sigma_1, \dots, \sigma_{r^*}\}$ ) and throw away the rest. As a consequence, we only keep the first  $r^*$  left-singular vectors and right-singular vectors in  $\mathbf{P}$  and  $\mathbf{Q}$  respectively. Here  $r^* < r$  is a hyperparameter specifying the number of vectors in  $\mathbf{P}$  and  $\mathbf{Q}$ , i.e., the number of features used to describe a word or a document. In this way, we have a new factorization of  $\mathbf{M}$  as

$$\mathbf{M} \approx \sum_i^{r^*} \sigma_i \mathbf{p}_i \mathbf{q}_i^T \quad (3.34)$$

The right hand side of Eq. (3.34) is also known as a **low-rank approximation** of  $\mathbf{M}$ . By specifying  $r^*$ , it can approximate  $\mathbf{M}$  with a matrix having an arbitrary rank  $< r$ .

There are a number of algorithms for implementing the SVD [Cline and Dhillon, 2014]. In fact, most of the modern implementations of the SVD are efficient and scalable. One can use them as off-the-shelf toolkits in NLP applications.

## 2. Principal Component Analysis

In data analysis, **principal component analysis (PCA)** is a widely-used technique for dimension reduction. Given a set of data points, PCA finds a sequence of orthogonal directions in the coordinate space so that the variance of the data points along these directions is maximized. These directions are typically represented as unit vectors, called **principal component loadings** or **principal component coefficients**. As a result, they form a new coordinate space to which we can map the given data points by an orthogonal linear transformation.

Consider a word-document co-occurrence matrix  $\mathbf{M} \in \mathbb{R}^{|V| \times |D|}$ , where each row is a  $|D|$ -dimensional word vector or feature vector. The PCA defines a linear mapping from  $\mathbb{R}^{|D|}$  to  $\mathbb{R}^p$ , that is, we transform each  $|D|$ -dimensional word vector to a  $p$ -dimensional word vector.

This is given by

$$\mathbf{N} = \mathbf{M}\mathbf{C} \quad (3.35)$$

where  $\mathbf{N} \in \mathbb{R}^{|V| \times p}$  is the mapped word vectors over the vocabulary  $V$ , and  $\mathbf{C} \in \mathbb{R}^{|D| \times p}$  is the matrix of the linear mapping. Then, we can write  $\mathbf{C}$  as a sequence of column vectors

$$\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 & \dots & \mathbf{c}_p \end{bmatrix} \quad (3.36)$$

Each column vector  $\mathbf{c}_i = \begin{bmatrix} c_i(1) \\ \vdots \\ c_i(|D|) \end{bmatrix}$  is a group of principal component coefficients, indicating a linear function that combines the input features into a new feature. For example, if we view  $\mathbf{M}$  as the values of a bunch of feature functions (say, column vectors  $\{\mathbf{m}_1, \dots, \mathbf{m}_{|D|}\}$ ), we can map  $\mathbf{M}$  to a new feature space in terms of  $\mathbf{c}_i$ :

$$\begin{aligned} \mathbf{M}\mathbf{c}_i &= \begin{bmatrix} \mathbf{m}_1 & \dots & \mathbf{m}_{|D|} \end{bmatrix} \begin{bmatrix} c_i(1) \\ \vdots \\ c_i(|D|) \end{bmatrix} \\ &= \sum_{k=1}^{|D|} c_i(k) \mathbf{m}_k \end{aligned} \quad (3.37)$$

$\mathbf{M}\mathbf{c}_i$  (i.e., the  $i$ -th column of  $\mathbf{N}$ ) is a column vector where each entry is the new feature for a word in  $V$ . In PCA, we generate  $\{\mathbf{c}_1, \dots, \mathbf{c}_p\}$  in sequence such that they maximize the variance of the linear mapping in Eq. (3.37). Thus, for each  $i \in [1, p]$ , the optimal principal component coefficients are defined to be

$$\begin{aligned} \hat{\mathbf{c}}_i &= \arg \max_{\mathbf{c}_i} \text{Var}(\mathbf{M}\mathbf{c}_i) \\ &= \arg \max_{\mathbf{c}_i} \mathbf{c}_i^T \mathbf{S} \mathbf{c}_i \end{aligned} \quad (3.38)$$

where  $\text{Var}(\mathbf{M}\mathbf{c}_i)$  is the variance of  $\mathbf{M}\mathbf{c}_i$ , and  $\mathbf{S}$  is the covariance matrix of  $\mathbf{M}$ . For a well-defined solution to Eq. (3.38), it is common to impose an additional constraint that  $\mathbf{c}_i$  is a unit vector, i.e.,  $\mathbf{c}_i^T \mathbf{c}_i = 1$ . Then, the problem can be framed as

$$\hat{\mathbf{c}}_i = \arg \max_{\mathbf{c}_i} \mathbf{c}_i^T \mathbf{S} \mathbf{c}_i - \lambda_i (\mathbf{c}_i^T \mathbf{c}_i - 1) \quad (3.39)$$

where  $\lambda_i$  is the Lagrange multiplier. Solving Eq. (3.39) under such a constraint requires  $\hat{\mathbf{c}}_i$  to be an eigenvector of  $\mathbf{S}$  and  $\lambda_i$  to be the corresponding eigenvalue [Jolliffe, 2002]. Since  $\mathbf{S}$  is a  $p \times p$  symmetric matrix, it has exactly  $p$  eigenvectors and eigenvalues. Then, we can order these eigenvectors by the associated eigenvalues, and take the ordered eigenvectors as  $\{\hat{\mathbf{c}}_1, \dots, \hat{\mathbf{c}}_p\}$ . In other words,  $\hat{\mathbf{c}}_1$  is the eigenvector of  $\mathbf{S}$  with the largest eigenvalue,  $\hat{\mathbf{c}}_2$  is the

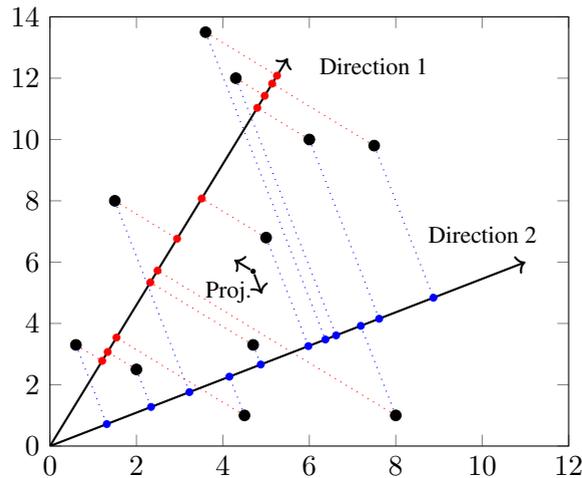


Figure 3.5: Transforming 2-dimensional data to 1-dimensional data via PCA. There are a number of data points (represented by black circles) on a Euclidean plane. By using PCA, we find a direction (represented by an arrow) such that the variance of the projected data (represented by colored circles) in this direction is maximized. Such a direction can be represented by a unit vector, called principal component coefficients. In this example, the principal component coefficients describe a 1-dimensional coordinate space. We can map the data from the 2-dimensional coordinate space to the 1-dimensional coordinate space via linear transformation. The mapped data is called the principal component of the original data points.

eigenvector of  $\mathbf{S}$  with the second largest eigenvalue, and so on. Typically,  $\mathbf{M}\hat{\mathbf{c}}_i$  is called the  $i$ -th **principal component** of  $\mathbf{M}$ .

An intuitive way to think about PCA is to map data points in a Euclidean space from one coordinate system to another. For a data set  $\mathbf{M}$ , we can view each row in  $\mathbf{M}$  as the coordinates of a data point in a  $|V|$ -dimensional coordinate system  $A$ . In PCA, we want to represent these data points in a new  $p$ -dimensional coordinate system  $B$ . The  $i$ -th dimension of the new coordinate system is simply a direction represented by a unit vector  $\mathbf{c}_i$ . For the  $i$ -th coordinate of each data point in  $B$ , we project the data point in  $A$  onto the  $\mathbf{c}_i$  line. The optimal  $\mathbf{c}_i$  is chosen in terms of how these projected data points are spread along  $\mathbf{c}_i$ . In other words, we seek a line along which we can best separate the data points. In this way, we generate a sequence of principal component coefficients, successively solving Eq. (3.38). We illustrate the idea of PCA using an example projecting 2-dimensional data to 1-dimensional data in Figure 3.5.

In real-world applications,  $p$  is commonly set to a number much smaller than  $|D|$ , and PCA can significantly reduce the number of dimensions used in representing words. Note that PCA is a very general method and is found to be useful in many disciplines. In practice,  $\mathbf{M}$  can be extended to represent observations on a set of variables. By applying PCA, one can transform these observations into data values of fewer new variables.

### 3. Others

In machine learning, learning low-dimensional models is a fundamental problem, and has been generalized in several directions. For example, the neural word embedding models described in Sections 3.4 and 3.5 themselves tend to learn low-dimensional, real-valued word vectors from texts. Here we present some of the dimension reduction methods one may come across in the NLP and machine learning literature.

- **Topic models.** Technically, topic models are not ways of dimension reduction, but tools for describing how documents and words are generated based on distributions over topics [Blei, 2012]. For example, **latent Dirichlet allocation (LDA)** models the generation of a document by using document-topic and topic-word distributions [Blei et al., 2003]. As a by-product, we obtain a distribution over words for each topic, indicating how likely a word occurs given a topic. If we write all these topic-word distributions as a matrix, say a  $|V| \times K$  matrix where  $|V|$  is the number of words and  $K$  is the number of topics, then we will have some sort of word representations that are very similar to those described in previous sections.  $K$  is commonly set to a “small” number (e.g., 200). In this case, we have a low-dimensional model for representing words. Although LDA is not so popular in learning word representations in NLP applications, it offers a way to represent words as distributions over latent thematic structures.
- **Auto-encoders.** Undercomplete auto-encoders are a type of neural model that encodes features into low-dimensional codes such that the input features can be reconstructed from the codes. An advantage of auto-encoders is that they do not make assumptions on the hidden structures of the features. Thus, auto-encoders can be used to learn to transform any type of data into low-dimensional representations. For example, in Chapter 7 we will see examples of applying auto-encoders to learn sentence representations. For more details about auto-encoders the reader can refer to Chapter 2.
- **Supervised dimension reduction.** Traditionally, dimension reduction methods (such as PCA) are assumed to work in an unsupervised manner. When the benchmark data of the target task is accessible, it is natural to make use of this information. A common example is supervised dimension reduction for classification. For example, in the **Fisher’s linear discriminant** and **linear discriminant analysis** methods, we find a mapping from high-dimensional data to single-dimensional data so that the separation of the classes associated with the data is maximized. This idea can be generalized to multi-dimensional data in the Canonical Variates method [Barber, 2012].
- **Feature selection.** Feature selection refers to a process of selecting a subset of the features used in representing an object and thus reducing the dimensionality of the representation. Feature selection is a wide-ranging topic in machine learning, and many methods can be seen as instances of feature selection [Guyon and Elisseeff, 2003; Liu and Motoda, 2012]. The simplest is to frame it as a search problem: we search in the space of feature subsets so that the selected features maximize (or minimize) some objective. In general, the design of the objective depends on the task where we apply the features. This makes feature selection somewhat difficult because one has

to consider many factors in such a process, such as the performance measure of the target task, the search efficiency, and the representation of each feature subset. Note that feature selection is generally discussed in supervised learning that requires labeled data to compute loss for optimization. The reader is referred to [Solorio-Fernández et al. \[2020\]](#)’s review paper for unsupervised feature selection methods.

In statistics, many methods can fall under the dimension reduction framework and are related to what we discussed in this section. For example, **factor analysis** is a method similar to PCA because they both seek a linear mapping from the input variables to a smaller number of new variables. The difference between them is that factor analysis focuses on modeling the common variance of variables, while PCA focuses on maximizing the variance of the projected data. Another example is **independent component analysis (ICA)**. Unlike PCA, the goal of ICA is to find independent components that are additively separable. More examples can be found in machine learning and statistics textbooks [[McClave and Sincich, 2006](#); [Freedman et al., 2007](#); [Barber, 2012](#)].

### 3.4 Inducing Word Embeddings from NLMs

Counting word-word or word-document occurrences is a simple way to represent words by using their distributions in texts. While this method is effective in many applications, it imposes a constraint on word representations: the entries of a word vector should be able to be explained as some “evidence” on how the word distributes in different contexts. Ideally, we would like to represent words in a more general form, say, a real-valued vector (call it the word embedding) without constraints or assumptions on how the meaning of each entry of the vector is defined.

Learning word vectors with no constraints comes at a cost. Unlike the count-based methods presented in Section 3.3, we do not use heuristics or prior knowledge to estimate the value of a word vector but wish to induce meaningful word representations directly from data. One of the difficulties here is that there is no gold standard to guide the learning process because it is simply impossible to manually annotate a real-valued word vector. Thus, we are often interested in treating the learning of word vectors as a part of a well-defined task (call it a **background task**). The learned word vectors are then a by-product of the learning on the background task.

A common example is the induction of word vectors from neural language models (NLMs). Recall the NLM described in Chapter 2. Its goal is to build a neural network that predicts the probability of a word given its preceding words [[Bengio et al., 2003a](#)]. More formally, let  $w_i$  be the word we want to predict, and  $\{w_{i-n+1}, \dots, w_{i-1}\}$  be the context words we have seen. First, the words  $\{w_{i-n+1}, \dots, w_{i-1}\}$  are transformed to  $d_e$ -dimensional word vectors  $\{\mathbf{e}_{i-n+1}, \dots, \mathbf{e}_{i-1}\}$  through an embedding layer. Assuming  $w_j$  is the one-hot representation of word  $j$  (a row vector of size  $|V|$ ), the word vector  $\mathbf{e}_j$  is given by

$$\mathbf{e}_j = w_j \mathbf{C} \tag{3.40}$$

where  $\mathbf{C} \in \mathbb{R}^{|V| \times d_e}$  is the parameter of the embedding layer.  $\mathbf{C}$  is often known as the word

embedding table in which the  $k$ -th row is the representation of the  $k$ -th word in  $V$ .

Then, we use a feed-forward neural network to compute the probability distribution of the word at position  $i$ . This is given by

$$\Pr(\cdot | w_{i-n+1}, \dots, w_{i-1}) = F_{\theta}(\mathbf{e}_{i-n+1}, \dots, \mathbf{e}_{i-1}) \quad (3.41)$$

where  $F_{\theta}(\cdot)$  is a feed-forward neural network parameterized by  $\theta$ . Typically, the embedding layer can be seen as a component of the NLM. Here we use slightly different notation to emphasize that the NLM is a function of both  $\theta$  and  $\mathbf{C}$ , like this

$$\Pr_{\theta, \mathbf{C}}(\cdot | w_{i-n+1}, \dots, w_{i-1}) = F_{\theta, \mathbf{C}}(w_{i-n+1}, \dots, w_{i-1}) \quad (3.42)$$

For training, we optimize both  $\theta$  and  $\mathbf{C}$  to minimize a loss function. A popular method is maximum likelihood training which maximizes the sum of log-likelihood over all  $n$ -grams in the data. Given a sequence of words  $w_1 \dots w_m$ , the objective of the training is defined to be<sup>18</sup>

$$(\hat{\theta}, \hat{\mathbf{C}}) = \arg \max_{\theta, \mathbf{C}} \sum_{i=n}^m \log \Pr_{\theta, \mathbf{C}}(w_i | w_{i-n+1}, \dots, w_{i-1}) \quad (3.43)$$

Having obtained the optimized parameters  $\hat{\theta}$  and  $\hat{\mathbf{C}}$ , we can apply  $F_{\hat{\theta}, \hat{\mathbf{C}}}(\cdot)$  to deal with new  $n$ -grams. More importantly, we have some well-trained word vectors (i.e.,  $\hat{\mathbf{C}}$ ) that can be used in systems other than NLMs. This is also known as the pre-training of word vectors. In pre-training, we can define  $F_{\theta, \mathbf{C}}(\cdot)$  as any system that makes use of the word vectors  $\mathbf{C}$ . Thus, the task of learning  $\mathbf{C}$  is transformed to the task of optimizing  $F_{\theta, \mathbf{C}}(\cdot)$  on the background task (see Figure 3.6 for an illustration). The main advantage of this method is that we can reuse existing NLP tasks to train the word vectors. A risk here is that the “best” word vectors found in training  $F_{\theta, \mathbf{C}}(\cdot)$  might not be well suited for the system where the word vectors are in actual use. Interestingly, in many situations, word vectors that are pre-trained by NLMs are of good quality for downstream tasks, or at least provide a good starting point for further tuning of these word vectors in the target system.

### 3.5 Word Embedding Models

In principle word vectors can be learned in any manner. Treating word vectors as components of existing NLP systems is one option, but typically lacks task-specific considerations. Another option is to develop methods specifically tailored to the problem. The training of such systems, therefore, does not need to satisfy the constraints of standard NLP tasks, making it easier to learn word vectors.

<sup>18</sup>This can be generalized to a data set consisting of multiple sequences.

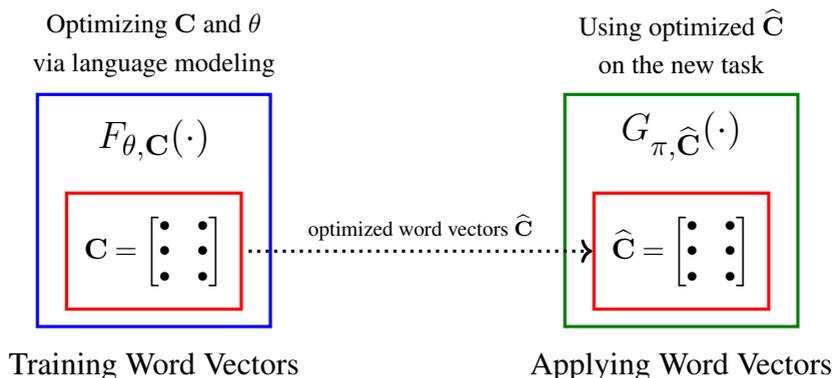


Figure 3.6: Illustration of pre-training word vectors in an NLM. The NLM can be denoted as a function  $F_{\theta, \mathbf{C}}(\cdot)$  of the word embedding table (i.e.,  $\mathbf{C}$ ) and other parameters of the NLM (i.e.,  $\theta$ ). The pre-training of  $\mathbf{C}$  is essentially a process of training  $F_{\theta, \mathbf{C}}(\cdot)$  on a background task. The outcome is the optimized word vectors  $\hat{\mathbf{C}}$  which are then applied to a new system  $G_{\pi, \hat{\mathbf{C}}}(\cdot)$  that might be different from the NLM. In the new system,  $\hat{\mathbf{C}}$  is the word embedding table learned from the NLM and  $\pi$  is the parameters specialized to  $G(\cdot)$ .

### 3.5.1 Word2Vec

**Word2Vec** is a short name for the models proposed in [Mikolov et al., 2013a;c]. As with neural language models, the Word2Vec models are based on neural networks. Rather than resorting to the generative modeling of  $n$ -grams, the Word2Vec models describe the learning of word vectors in a log-linear fashion. In consequence, the architectures of these models are different from those used in language modeling. There are two types of models in Word2Vec:

- **The continuous bag-of-words model (or the CBOW model).** The CBOW model is a word prediction model. It is used to predict how likely a word at position  $i$  occurs given the  $-n$  and  $+n$  word windows around it. The structure of the CBOW model is similar to that of the neural language model introduced in Chapter 2 (see Figure 3.7 (a)). First, we use an embedding layer to transform the context words  $w_{i-n} \dots w_{i-1}$  and  $w_{i+1} \dots w_{i+n}$  to corresponding word vectors. This is performed by multiplying the one-hot representation of each input word  $w_j$  with the embedding table  $\mathbf{C} \in \mathbb{R}^{|V| \times d_e}$ , as shown in Eq. (3.40). These word vectors are then averaged to produce a single representation for the input words, giving us

$$\mathbf{h} = \frac{1}{2n} \left( \sum_{j=i-n}^{i-1} w_j \mathbf{C} + \sum_{j=i+1}^{i+n} w_j \mathbf{C} \right) \quad (3.44)$$

Note that the above defines a model that completely ignores the order of input words because of the use of the sum operation. This explains why the CBOW model is called *bag-of-words*. The output layer of the CBOW model is a standard Softmax layer that

projects  $\mathbf{h}$  to a probability distribution over the vocabulary

$$\mathbf{y} = \text{Softmax}(\mathbf{h}\mathbf{U} + \mathbf{b}) \quad (3.45)$$

where  $\mathbf{U} \in \mathbb{R}^{d_e \times |V|}$  is the parameter matrix of the linear mapping and  $\mathbf{b} \in \mathbb{R}^{|V|}$  is the bias term.  $\mathbf{y}$  is a distribution over the vocabulary, and  $\Pr(w_i | w_{i-n}, \dots, w_{i-1}, w_{i+1}, \dots, w_{i+n}) = y(w_i)$ . Eqs. (3.44-3.45) describe a very simple neural network. An advantage is that the resulting model is small and efficient as compared to NLMs. The training of the CBOW model is regular. We can frame it as finding the maximum likelihood estimation of the parameters of the model. For simplicity, let  $\theta$  denote the parameters other than  $\mathbf{C}$  (i.e.  $\theta = \{\mathbf{U}, \mathbf{b}\}$ ). We have

$$(\hat{\theta}, \hat{\mathbf{C}}) = \underset{\theta, \mathbf{C}}{\text{argmax}} \sum_{i=n+1}^{m-n-1} \log \Pr_{\theta, \mathbf{C}}(w_i | w_{i-n}, \dots, w_{i-1}, w_{i+1}, \dots, w_{i+n}) \quad (3.46)$$

where  $m$  is the length of the word sequence. After training, we can simply drop  $\hat{\theta}$  and use  $\hat{\mathbf{C}}$  as a word vector look-up table.

- **The continuous skip-gram model (or the skip-gram model).** The skip-gram model is another word prediction model. It models the reverse of the task described in Eqs. (3.44-3.45). To be more precise, our objective is to predict each of the  $\pm n$  context words given  $w_i$ . This is generally framed as estimating the probability of  $w_j$  occurring given  $w_i$  ( $i-n \leq j \leq i-1$  or  $i+1 \leq j \leq i+n$ ). Figure 3.7 (b) shows the structure of the skip-gram model. The embedding layer deals with  $w_i$  as usual. The representation of  $w_i$  is given by

$$\mathbf{h} = w_i \mathbf{C} \quad (3.47)$$

It is then passed to a Softmax layer to predict the probability for each context word  $w_j$  (assuming  $j = i+k$ )<sup>19</sup>

$$\mathbf{y}_k = \text{Softmax}(\mathbf{h}\mathbf{V}_k + \mathbf{b}_k) \quad (3.48)$$

where  $\mathbf{V}_k$  and  $\mathbf{b}_k$  are the parameters of the model ( $-n \leq k \leq -1$  and  $1 \leq k \leq n$ ). We have

$$\begin{aligned} \Pr(w_j | w_i) &= \Pr(w_{i+k} | w_i) \\ &= y_k(w_{i+k}) \end{aligned} \quad (3.49)$$

Let  $\theta$  be a short representation of  $\{\mathbf{V}_k\}$  and  $\{\mathbf{b}_k\}$ . The training problem can be defined

<sup>19</sup>When  $k > 0$ ,  $w_j$  is a word in the right context window of  $w_i$ ; when  $k < 0$ ,  $w_j$  is a word in the left context window.

as

$$(\hat{\theta}, \hat{\mathbf{C}}) = \arg \max_{\theta, \mathbf{C}} \sum_{i=n+1}^{m-n-1} \sum_{\substack{-n \leq k \leq -1, \\ 1 \leq k \leq n}} \log \Pr_{\theta, \mathbf{C}}(w_{i+k} | w_i) \quad (3.50)$$

Both of the above models make an analogy to cloze tests by considering only the pairwise dependency between words. A danger is that if complex relationships among words and word order information are required, the resulting probability distributions will be not that precise compared to language models. Note, however, that the goal of these models is not to precisely predict missing words given their contexts, but to learn word representations from some task that captures word-word relationships. It is therefore not so important to care about the word prediction performance of the learned model.

Another merit of these models is that they have very simple, easy-to-train architectures. For example, in both models there are no hidden layers and the embedding layer is directly connected to the output layer. These model structures can be seen as instances of log-linear modeling in machine learning: the input variables are linearly transformed to a feature vector (e.g., Eq. (3.44)), followed by a log-linear function (e.g., Eq. (3.45)).

### 3.5.2 GloVe

**Global vectors**, also known as **GloVe**, are word vectors that are learned by using both global statistics over the corpus and local models of word prediction [Pennington et al., 2014]. The GloVe method starts with a word-word co-occurrence matrix (see Section 3.3), and then forms a neural model by making a series of assumptions.

Given a word-word co-occurrence matrix  $\mathbf{M}$ , where each cell  $M(a, b) = \text{count}(a, b)$  represents the number of co-occurrences of words  $a \in V$  and  $b \in V$ , we can obtain the conditional probability  $\Pr(b|a)$  by using the equation

$$\begin{aligned} \Pr(b|a) &= \frac{\text{count}(a, b)}{\sum_{b'} \text{count}(a, b')} \\ &= \frac{\text{count}(a, b)}{\text{count}(a)} \end{aligned} \quad (3.51)$$

where  $\text{count}(a)$  is the number of times the word  $a$  occurs in the corpus.

Let us now see a motivating example of GloVe. Suppose that we want to distinguish between words *air* and *water*. It is easy to obtain how likely one of these words occurs given a context word in the corpus via Eq. (3.51). See the following table for a small fraction of the  $\Pr(b|a)$  matrix from 3.8M-sentence English data in WMT14.

Entry	$w = \textit{fly}$	$w = \textit{drink}$	$w = \textit{breath}$	$w = \textit{live}$	$w = \textit{flow}$
$\Pr(\textit{air} w)$	$1.5 \times 10^{-4}$	$6.2 \times 10^{-5}$	$2.2 \times 10^{-4}$	$1.6 \times 10^{-4}$	$3.6 \times 10^{-4}$
$\Pr(\textit{water} w)$	$1.3 \times 10^{-5}$	$4.1 \times 10^{-4}$	$1.8 \times 10^{-5}$	$1.4 \times 10^{-4}$	$3.0 \times 10^{-4}$
$\Pr(\textit{air} w)/\Pr(\textit{water} w)$	11.54	0.15	12.2	1.14	1.2

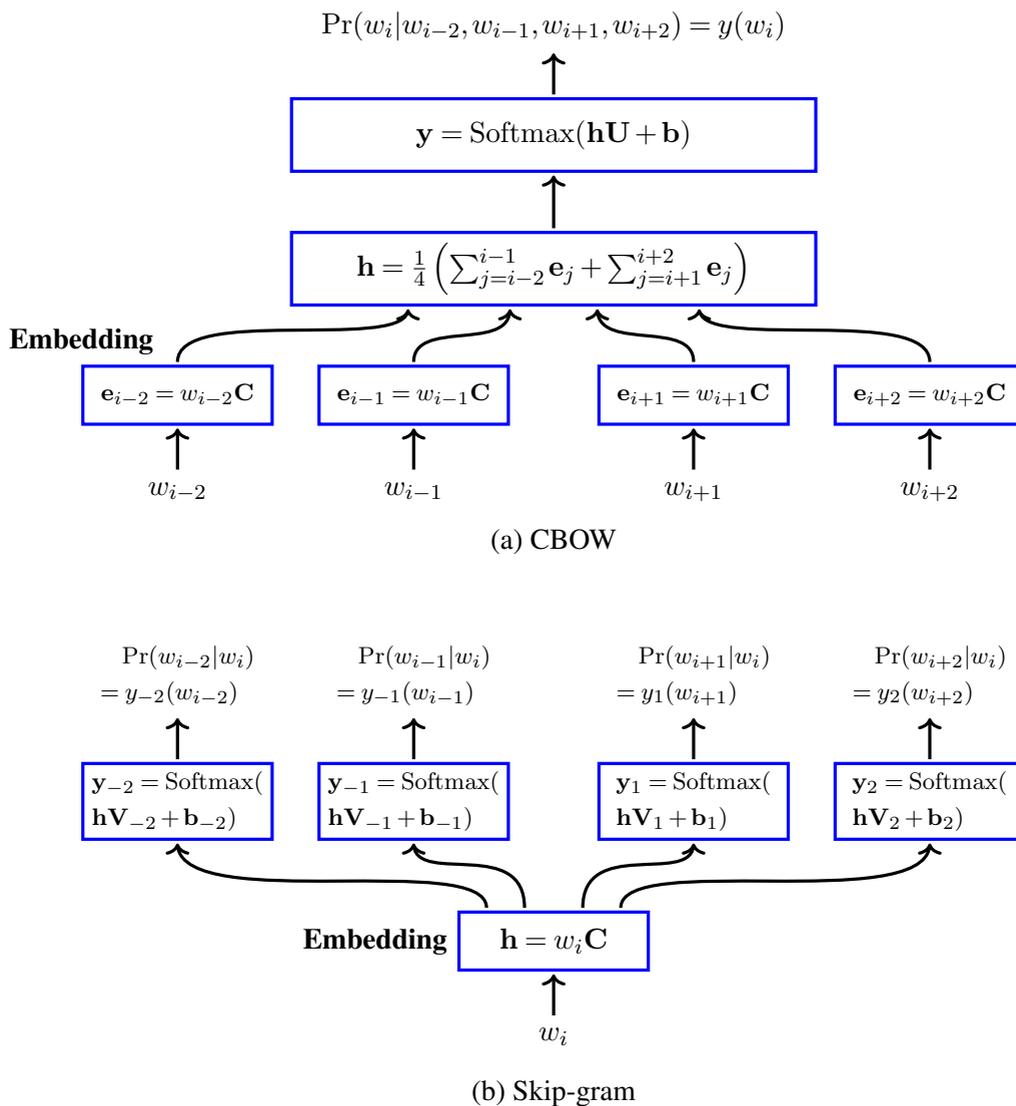


Figure 3.7: The CBOW and skip-gram architectures. The CBOW model computes the probability  $\Pr(w_i|w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2})$  where  $w_i$  is a word in a sequence and  $\{w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}\}$  are words in the  $\pm 2$  context windows. The context representation  $\mathbf{h}$  is the mean of the word vectors that are produced through an embedding layer.  $\mathbf{h}$  is then fed into a Softmax layer to output a distribution over the vocabulary (i.e.,  $\mathbf{y}$ ). The prediction probability of  $w_i$  is  $\Pr(w_i|w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}) = y(w_i)$ . The skip-gram model is also based on the embedding + Softmax structure. It models the probability of each context word  $w_j$  given the word  $w_i$ . This is achieved by simply computing the output of a standard Softmax layer that takes the vector representation of  $w_i$  as input. Both the CBOW and skip-gram models are trained in a maximum likelihood fashion. The resulting lookup table of the embedding layer is the word vectors (or embeddings) for the words in the vocabulary.

In this table,  $\Pr(\text{air}|w)$  and  $\Pr(\text{water}|w)$  indicate how well *air* and *water* correlate with different  $w$ . We also compute the probability ratio  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$  in the last line

of the table. Interestingly, it is found that  $w$  can be viewed as a probe word by which  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$  models the relevance between words. When  $w$  is more relevant to *air* but less relevant to *water* (e.g.,  $w = \text{fly}$  or  $w = \text{breath}$ ),  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$  is large. In contrast, when  $w$  is less relevant to *air* but more relevant to *water* (e.g.,  $w = \text{drink}$ ),  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$  is small. When  $w$  is relevant to both words, or irrelevant to them (e.g.,  $w = \text{live}$  or  $w = \text{flow}$ ),  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$  is around 1.

An insight that we can gain from the above examples is that the word vectors should be able to interpret  $\Pr(\text{air}|w)/\Pr(\text{water}|w)$ . A simple idea is to develop a model to approximate this probability ratio, say,

$$F(\mathbf{e}_a, \mathbf{e}_b, \tilde{\mathbf{e}}_w) = \frac{\Pr(a|w)}{\Pr(b|w)} \quad (3.52)$$

where  $\mathbf{e}_a, \mathbf{e}_b \in \mathbb{R}^{d_e}$  are the vector representations of the words  $a$  and  $b$ , and  $\tilde{\mathbf{e}}_w \in \mathbb{R}^{d_e}$  is the vector representation of the context word  $w$ . Note that the notation has different meanings for  $\mathbf{e}$  and  $\tilde{\mathbf{e}}$ . The former is a word vector from an embedding table  $\mathbf{C}$ , and the latter is a word vector from another embedding table  $\tilde{\mathbf{C}}$ . The use of two embedding tables has several advantages. The main advantage is that combining multiple sets of parameters could mitigate the overfitting of the model. The final word embedding table takes the form  $\frac{\mathbf{C} + \tilde{\mathbf{C}}}{2}$ .

There are many ways to define the function  $F(\cdot)$ . Here we simply treat  $F(\cdot)$  as a neural network parameterized by  $\mathbf{C}$ ,  $\tilde{\mathbf{C}}$  and some other parameters. Considering the subtraction nature in comparing  $a$  and  $b$  in  $\frac{\Pr(a|w)}{\Pr(b|w)}$ , we can assume that  $F(\cdot)$  depends on  $\mathbf{e}_a - \mathbf{e}_b$ . Furthermore, we can take  $\mathbf{e}_a \mathbf{e}_w^T \in \mathbb{R}$  (or  $\mathbf{e}_b \mathbf{e}_w^T \in \mathbb{R}$ ) to model the relationship between the word  $a$  (or  $b$ ) and the context word  $w$ . These lead to a new form of the function

$$F((\mathbf{e}_a - \mathbf{e}_b) \tilde{\mathbf{e}}_w^T) = \frac{\Pr(a|w)}{\Pr(b|w)} \quad (3.53)$$

where  $(\mathbf{e}_a - \mathbf{e}_b) \tilde{\mathbf{e}}_w^T \in \mathbb{R}$  is the difference in representing words  $a$  and  $b$  when taking  $w$  as a probe word.

There are still many solutions to Eq. (3.53), though the input of the function is greatly simplified. For a feasible form of  $F(\cdot)$ , we further assume that Eq. (3.53) holds when we either exchange the embedding tables  $\mathbf{C}$  and  $\tilde{\mathbf{C}}$  (i.e., exchange  $\mathbf{e}$  and  $\tilde{\mathbf{e}}$  for  $a, b$  and  $w$ ), or transpose the word-word co-occurrence matrix (i.e., use  $\mathbf{M}$  instead of  $\tilde{\mathbf{M}}$ ). To make use of these assumptions, one way is to let  $F(\cdot)$  be a homomorphism between two sides of Eq. (3.53). That is

$$F((\mathbf{e}_a - \mathbf{e}_b) \tilde{\mathbf{e}}_w^T) = \frac{F(\mathbf{e}_a \tilde{\mathbf{e}}_w^T)}{F(\mathbf{e}_b \tilde{\mathbf{e}}_w^T)} \quad (3.54)$$

The solution to Eq. (3.54) requires that  $F(\cdot) = \exp(\cdot)$ , and we have

$$\begin{aligned} F(\mathbf{e}_a \tilde{\mathbf{e}}_w^T) &= \exp(\mathbf{e}_a \tilde{\mathbf{e}}_w^T) \\ &= P(a|w) \\ &= \frac{\text{count}(a, w)}{\text{count}(a)} \end{aligned} \quad (3.55)$$

Rewriting this equation, we have

$$\mathbf{e}_a \tilde{\mathbf{e}}_w^T + \log \text{count}(a) - \log \text{count}(a, w) = 0 \quad (3.56)$$

A problem with Eq. (3.56) is that the term  $\log \text{count}(a)$  makes the solution non-exchangeable for  $\mathbf{M}$  and  $\widetilde{\mathbf{M}}$ . To address this, a method is to absorb  $\log \text{count}(a)$  in some terms that are symmetric for  $a$  and  $w$ , like this

$$\mathbf{e}_a \tilde{\mathbf{e}}_w^T + \beta_a + \tilde{\beta}_w - \log \text{count}(a, w) = 0 \quad (3.57)$$

where  $\beta_a$  and  $\tilde{\beta}_w$  are bias terms that depend on  $a$  and  $w$ , respectively. The quantity on the left-hand side of Eq. (3.57) describes how well  $\mathbf{e}_a \tilde{\mathbf{e}}_w^T + \beta_a + \tilde{\beta}_w$  fits the co-occurrence matrix. We wish to find some word vectors to enforce this quantity to be close to 1. Then, we can define the squared loss, as follows

$$L_{a,w} = \left( \mathbf{e}_a \tilde{\mathbf{e}}_w^T + \beta_a + \tilde{\beta}_w - \log \text{count}(a, w) \right)^2 \quad (3.58)$$

The loss over all pairs of  $a$  and  $w$  is given by

$$L_{\text{GloVe}} = \sum_{a,w \in V} \gamma(\text{count}(a, w)) \cdot L_{a,w} \quad (3.59)$$

where  $\gamma(\text{count}(a, w))$  is a scalar for  $L_{a,w}$ . In [Pennington et al. \[2014\]](#)'s paper,

$$\gamma(\text{count}(a, w)) = \begin{cases} \left( \frac{\text{count}(a, w)}{\text{count}_{\max}} \right)^\sigma & \text{count}(a, w) < \text{count}_{\max} \\ 1 & \text{otherwise} \end{cases} \quad (3.60)$$

where  $\text{count}_{\max}$  and  $\sigma$  are hyper-parameters. Typically,  $\sigma$  is set to a number smaller than 1. As such,  $\gamma(\text{count}(a, w))$  will penalize the word-pair  $(a, w)$  if  $\text{count}(a, w) < \text{count}_{\max}$ , that is, the loss function will assign smaller weights to rare word-pairs.

Eqs. (3.58-3.59) provide a very simple way to learn word vectors and can be implemented by using standard neural network building blocks (e.g., vector dot product and summation). An important property of GloVe is that the model  $\mathbf{e}_a \tilde{\mathbf{e}}_w^T + \beta_a + \tilde{\beta}_w - \log \text{count}(a, w)$  is itself linear. The training is even achieved without the need of cross-entropy loss. This differentiates GloVe greatly from NLM and word2vec in which expensive normalization of the output is required. The intuition here is that the relation between two words can be modeled in ways other than probability-based divergence. In fact, Eq. (3.58) looks more like a regression model

that fits the data of  $\log \text{count}(a, w)$ , that is, we tend to learn to predict  $\log \text{count}(a, w)$  for any pair of  $(a, w)$ .

Another note about the use of global data bears repeating. The co-occurrence matrix is a source of information that describes the entire corpus. An important consequence of using such information is that the learning task is framed as finding word vectors that are globally optimized. Of course, this does not make GloVe unique because the learning of many models like NLM and Word2Vec itself admits a simple formulation as a global optimization problem, e.g., maximizing the likelihood over the entire input space. However, the objectives in those problems are complex, and most of them are in practice trained via online learning, e.g., updating the model parameters on a batch of samples each time. Given this, GloVe actually defines a more efficient global model as compared with NLM and Word2Vec.

### 3.5.3 Remarks

We have seen in the previous sections how word vectors are learned by using several different methods. We now turn to discussions of issues that one might be interested in when training and/or applying word vectors.

- **Count-based vs Neural Network-based.** The simplicity and interpretability of count-based methods have long been appreciated. The use of the distributional hypothesis greatly simplifies the problem, but makes a strong assumption on the information source the word vectors can be learned from, and generally leads to data sparsity due to the curse of dimensionality. At the other end of the spectrum is learning with no assumptions. In these methods, we remove the constraints on the meaning of each dimension, but treat word vectors as low-dimensional intermediate states of a neural network that is developed to accomplish some NLP task. This enables the learning of features that are hard to describe in representing a word. The comparison of the two types of methods here can fall under the comparison of two well-known learning paradigms, say, feature engineering vs. end-to-end learning. Here we do not want to get bogged down by this topic. It is, however, worth pointing out that it does not necessarily restrict word vectors to certain forms. In general, the choice of the types of word vectors depends on in what application we apply them and what interpretation we place on them. For example, if we wish to have some interpretable, easy-to-learn word representation, inducing word vectors from co-occurrence matrices might be a good choice; if we wish to have some real-valued, low-dimensional word vectors that will be integrated into a bigger neural network, deep learning methods might be worth a try. Note that, learning continuous word vectors has become more and more common recently, given that the past few years have significant progress toward neural models of NLP. Also, there has been much interest in comparing count-based and neural network-based methods, and in exploring relationships between them [Levy and Goldberg, 2014b; Baroni et al., 2014; Levy and Goldberg, 2014c; Schnabel et al., 2015a; Levy et al., 2015; Gladkova et al., 2016].
- **Shallow Models vs Deep Models.** While it has become popular to solve the word vector learning problem using neural networks, the model structures we introduced in

this chapter are simple. Technically, they all have one or two layers of neurons and are often thought of as instances of shallow models. A similar example is the **vLBL** word embedding model [Mnih and Kavukcuoglu, 2013]. It models the interaction among words using a two-layer neural network. This model, which does not even involve a Softmax function, is one of the simplest word embedding models subject to our knowledge. Such a simple model, however, still works well in many cases. A benefit of shallow models is that they are efficient and scalable to a large amount of data. This makes it easier to use them to deal with more “difficult” NLP problems. A good example is the **fastText** system for text classification [Joulin et al., 2017]. It has a similar architecture to the CBOW model (see Section 3.5.1). In fastText, the input text is represented as a bag of word vectors that are averaged to form a hidden representation of the text. This is followed by an output layer that maps the hidden representation to a distribution over predefined classes. In this way, the classification model and word vectors are trained jointly. Although shallow models are remarkably effective for word vector learning, there are deeper models that one may be interested in for more modeling power. As with most multi-layer neural networks, learning word vectors with deep neural networks has a couple of benefits [Telgarsky, 2016]. First, by using a deep model, we can exploit potentially better hypotheses in a large hypothesis space. Second, deep models introduce more non-linearity into modeling, and thus increase the ability of the model to describe complex problems. There are many examples of learning word vectors in deep models. The simplest of these might be to simply stack more layers on the word embedding layer in those systems. The stacked layers can be feed-forward layers, recurrent layers, convolutional layers, or some combination of them. More recently, word vectors have been employed and/or trained by very deep and complex systems, achieving state-of-the-art performance on many NLP tasks [Radford et al., 2018; Devlin et al., 2019]. However, stronger models come with added computational and training challenges. So there are several lines of research on meeting these challenges [Pascanu et al., 2013; Bapna et al., 2018; Wang et al., 2019a; Zhang et al., 2019a; Pham et al., 2019; Li et al., 2020b]. In Chapters 4-6, we will see several successful NLP systems that are based on very deep neural networks.

- **Training Objectives.** The idea of taking word vector representations as parameters of a model fits well with the latent-variable modeling: a model is parameterized with learnable word vectors, and the values of these word vectors are inferred by maximizing or minimizing some objective function of the entire model. While such a learning process is regular in most situations, the training objective varies somewhat. A difficulty with this is that there is no obvious objective for directly signaling the training of word vectors. A simple solution to this difficulty is to resort to well-defined NLP tasks. For example, we can use word vectors to represent the input of an NLP model (such as language modeling and text classification systems). Hence the word vectors can serve as standard parameters of the model and be optimized as usual. Another solution is to develop “new” training tasks. As in general machine learning problems, however, this is a wide-ranging topic and there are so many choices to design a training objective. So a general method

is to slightly update existing tasks. For example, the training objective of CBOW is essentially based on the general word prediction problem, and has a similar form as that used in language modeling. We will also see several new tasks that stem from language modeling in Chapter 7. Yet in another sense these training tasks do not directly concern themselves with the issue of learning word vectors, but generally offer a way to inject it into a well-designed, efficient training procedure. Note that, in word vector applications, we may not assume a supervised learning scenario: the learned word vectors can be used in various systems that we have no idea of these application systems in the training stage. This makes the problem more like an unsupervised learning problem because there is no supervision information from the task where the word vectors are in actual use. Sometimes, when the target application is accessible, and there is some labeled data, we can have further training on those word vectors that have been trained somewhere.

## 3.6 Evaluating Word Embeddings

Having obtained the vector representation of words, we need to assess the quality of these vectors. Ideally, we wish to evaluate the word vectors against a gold standard. However, unfortunately, there is in general no such gold standard data since no one can annotate a vector of numbers for describing a word. A simple solution in this case is to resort to the result of some working system in which these word vectors are involved. Typically, there are two types of evaluation approaches [Schnabel et al., 2015b].

- **Extrinsic Evaluation** (or end-to-end testing). We directly incorporate the word vectors into an NLP system which is easy to evaluate, and see how the performance of the system is influenced by the word vectors.
- **Intrinsic Evaluation**. We test the ability of the word vectors to model the given aspects of morphological, syntactic, and semantic problems.

We will briefly describe below how these approaches are applied to word vector evaluation.

### 3.6.1 Extrinsic Evaluation

This approach is often taken in practice since it allows researchers and engineers to glean a quick understanding of how a real-world system behaves when changing part of it. Since many NLP systems use words as inputs, it is common to replace the symbolic representation of words in these systems with the word vectors. So far, we have seen several systems of this kind, commonly with an embedding layer transforming the one-hot representation to the real-valued vector representation of each input word, see for example the neural language model in Chapter 2.

Given such a system and a set of learned word vectors, we can use its performance as a measure of the quality of the word vectors. Considering the way we use the word vectors, there are two ways to train the system:

- **Word Vectors as Fixed Parameters**. We fix the word vectors, and train other parameters

of the system as usual.

- **Word Vectors as Initial Parameters.** We train all the parameters in the same manner. In this way, the provided word vectors can be seen as initial values of some of the parameters, and would be updated during training.

Both methods fall under the area of pre-training, and could be extended to cover many problems where part of a model is well trained before seeing the downstream task. By fixing word vectors, we simplify the training process, leading to a quick evaluation of the word vectors. In contrast, treating the word vectors as learnable parameters may increase the difficulty of training, but could learn “new” word vectors that are better suited for the working system.

Note that although extrinsic evaluation is of interest to practitioners, the results from this evaluation are highly dependent on the system in which we apply the word vectors. Because developing a desired NLP system often involves sophisticated training and tuning procedures other than word representation, the conclusion drawn by experimenting with such a complex system is greatly influenced by the way we build and use the system. This is also the case for many other NLP problems. For example, a tokenization method that is helpful for a machine translation system might not be a good choice for an information retrieval system. Therefore, to test the generalizability of the given word vectors, a widely-used approach is to carry out experiments on a variety of NLP systems.

### 3.6.2 Intrinsic Evaluation

Although much of word representation research involves end-to-end tests in NLP applications, it also involves examining the ability of the representation to deal with certain problems, such as interpreting the relationship between two words. There are many ways to design intrinsic evaluation, each addressing a specific problem. In the following we describe some of these methods. For more comprehensive descriptions about intrinsic evaluation, the reader can refer to papers on this subject [Baroni et al., 2014; Bakarov, 2018; Rogers et al., 2018].

#### 1. Semantic Relatedness

Modeling the relatedness between words is perhaps the most popular method to evaluate the quality of word vectors in NLP [Reisinger and Mooney, 2010; Huang et al., 2012; Baroni et al., 2014]. It is fundamentally about computing some distance between words (call it the **word semantic distance** or **word distance** for short). The motivation is that the word distance in a word vector space should agree with the judgments on the word relatedness in our mind [Rubenstein and Goodenough, 1965]. For example, we wish that *dog* is close to *wolf*, and *peach* is far from *television*. Mathematically, there are a lot of ways to calculate the distance (or angle) between two vectors. A simple and commonly used distance measure is the Euclidean distance. Also, we can compute the cosine similarity of two vectors to obtain a score in the interval  $[-1, 1]$ <sup>20</sup>.

In evaluation, we are given a set of word pairs, each of which is assigned an expected

<sup>20</sup>It is often to use the absolute value of the cosine score so that 0 indicates two vectors in the same direction and 1 indicates two orthogonal vectors.

distance by humans. Then, given a pair of words, we compare the expected distance with the distance in the word vector space. The quality of the word vectors is reflected in the difference between the two distances. However, a difficulty here is that there is, in practice, no gold-standard distance between words. Even for humans, it is still very difficult to give an exact number to describe how close a word is to another. An alternative method in this case is to categorize the distance into a few categories or rating scores, such as an integer in  $[1, 5]$  [Reisinger and Mooney, 2010]. This greatly reduces the difficulty in data annotation. Another way to reduce the difficulty is to let the model find the most similar word in a small set of candidates to a given word. Such a method prevents us from predicting an absolute distance between words. Instead we only need some mechanism to obtain the relative distance or similarity between words [Baroni et al., 2014].

Judging the relationship between words, however, may result in a highly ambiguous task because of the ambiguous nature of language use and understanding. In general, many factors may affect one’s thoughts on how words are related [Faruqui et al., 2016]. For example, *corn* and *cornea* are similar if we consider string overlaps in the suffix, but they are semantically dissimilar because they refer to different meanings. The ambiguity also comes from the definition of relatedness. Sometimes, relatedness and similarity are two terms used interchangeably but they may refer to different concepts. For example, *car* is related to *road*, but in another sense *car* is similar to *van*. Another problem is that the meaning of a word is often context-dependent. This makes it more difficult to establish the relationship between words with multiple different meanings (i.e., polysemy). Broadly speaking, this is an inherent problem with statistic word vector models where every word is assumed to be mapped to a single vector. For contextualized modeling of word vectors, we will describe in the following chapters several methods that consider a word to be different in representation given different contexts.

## 2. Word Analogy

Word analogy is concerned with modeling analogical relations between pairs of words. The assumption here is that the relation between words can be captured by performing simple algebraic operations on the corresponding word vectors. A well-known example is the one presented in Mikolov et al. [2013d]’s paper, where it is found that the way a word is related to another word can be described by vector subtraction. This leads to an interesting result: if we subtract *man*’s word vector from *king*’s word vector, and add *woman*’s word vector to it, then we will obtain a word vector close to *queen*’s. That is

$$\mathbf{e}_{king} - \mathbf{e}_{man} + \mathbf{e}_{woman} \approx \mathbf{e}_{queen} \quad (3.61)$$

Formally, word analogy is a task of comparing two word pairs  $(a, a^*)$  and  $(b, b^*)$ . An analogy can be made if the way  $a$  is related to  $a^*$  is similar to the way  $b$  is related to  $b^*$ . This essentially reflects some sort of **linguistic regularity** in word vectors, which can be expressed

by using vector subtraction:

$$\mathbf{e}_{a^*} - \mathbf{e}_a \approx \mathbf{e}_{b^*} - \mathbf{e}_b \quad (3.62)$$

The word analogy can be framed as an analogical reasoning task: we try to predict  $\mathbf{e}_{b^*}$  using  $\mathbf{e}_a$ ,  $\mathbf{e}_{a^*}$  and  $\mathbf{e}_b$ . More specifically, we wish  $\mathbf{e}_{a^*} - \mathbf{e}_a + \mathbf{e}_b$  to be close to  $\mathbf{e}_{b^*}$  if  $(a, a^*)$  and  $(b, b^*)$  hold similar relations. Also, improvements can be made on such a formulation. For example, we can consider the angle between vectors  $\mathbf{e}_{a^*} - \mathbf{e}_a$  and  $\mathbf{e}_{b^*} - \mathbf{e}_b$ , rather than the difference in  $\mathbf{e}_{a^*} - \mathbf{e}_a + \mathbf{e}_b$  and  $\mathbf{e}_{b^*}$  [Levy and Goldberg, 2014b].

Word analogy provides a simple way to examine the linearity property of a word vector model which is not typically involved in classic methods. An interesting point here is that the recent word vector models exhibit good linear behavior, although we do not consider this in modeling and/or training. It also gives researchers useful insights into the models learned by those methods and into potential ways of applying these models [Levy and Goldberg, 2014b; Linzen, 2016; Allen and Hospedales, 2019]. On the other hand, word analogy is not a general-purpose method. In many cases, it does not correlate well with the performance of downstream systems, and is thereby used as a way to study certain issues of word representation.

### 3. Word Categorization (or Clustering)

Another way to see how well the word vectors correlate with our understanding of word meaning is to see how well these vectors can be categorized into meaningful groups. This is often achieved by performing clustering algorithms on the word vectors. We wish that similar words are grouped into the same cluster, and dissimilar words are grouped into different clusters. For example, *apple*, *grape*, *peach*, and *orange* belong to the same group of words because they are all fruits. An advantage of this kind of evaluation is that many clustering algorithms and word clustering benchmarks have been developed and are straightforwardly applicable here. On the other hand, as in most clustering tasks, there are practical issues that we have to deal with, such as determining the number of clusters.

In machine learning, most clustering methods require computing the distance between data points. In this sense, word clustering is essentially based on the same idea of modeling the word relatedness, though we do not need to judge the quality of the distance in this case. This shows some intrinsic connections among different evaluation methods. However, as a side-effect, word clustering inherits the same problem with related methods (such as semantic relatedness). As discussed in Section 3.6.2, it is difficult to design a gold-standard criterion to measure how well the words are clustered, since we can group words into clusters in so many different ways.

### 4. Subconscious Evaluation

The general idea of subconscious evaluation is to examine the correlation between the use of word vectors and subconscious behaviors or brain functions when one reads text. A wide variety of psycholinguistic phenomena can be used as the test [Mitchell and Lapata, 2010]. A well-known method is **priming** which studies how a person responds to stimuli [Schacter and

Buckner, 1998; Tulving and Schacter, 1990; Wiggs and Martin, 1998]. For example, we can design an experiment to test the speed with which a person reads a given word (call it the **target word**) when it follows another word (call it the **prime word**) [Meyer and Schvaneveldt, 1971; Lund, 1995; McNamara, 2005]. If the target word  $t$  is read more quickly when following a word  $a$  than when following another word  $b$ , then we would say that  $t$  correlates more with  $a$  than  $b$ . Then, we can use such a psychological measure to judge the distance or similarity between word vectors. To obtain the time the participant takes in reading, a popular method is to frame it as a **self-paced reading** task<sup>21</sup>. Another method is to use eye-tracking to automatically record the information of the eye movement and position. By using these techniques, several methods and data sets have been used for studying a variety of psycholinguistic issues [Mitchell and Lapata, 2010; Hutchison et al., 2013; Lapesa and Evert, 2014; Klerke et al., 2015; Sjøgaard, 2016; Auguste et al., 2017].

In addition to tracking human behavior in reading, we can monitor brain activity by using neurological tests, such as functional magnetic resonance imaging (fMRI) and electroencephalography (EEG) [Devereux et al., 2010; Sjøgaard, 2016; Bhattasali et al., 2020]. For example, it is often hypothesized that, when a person reads and understands words, some activations occur in his or her brain. Therefore we can link the meaning of words with brain functions. On the other hand, an objection is that the knowledge about the mechanism behind these processes is still limited, making it difficult to correlate the results of these studies with real-world NLP systems [Baroni et al., 2014; Bakarov, 2018].

## 5. Linguistically Motivated Evaluation

Linguistically motivated evaluation is based on an assumption that word vectors learned from data should explain linguistic resources. One interesting approach to performing such evaluation is to align the word vectors with some representations of the entries of a dictionary [Tsvetkov et al., 2015; Acs and Kornai, 2016]. The quality of the word vectors is measured in terms of the correlation between these word vectors and the linguistic representations<sup>22</sup>. Apart from standard dictionaries, we can compare the word vectors against a semantic network, such as WordNet. In this way, the evaluation would be improved if we consider graph-based algorithms on resources of this type [Agirre et al., 2009].

### 3.6.3 Visualization

Taking word vectors as data points, we can adopt general approaches to visualizing multi-dimensional data to locate data points in a 2 or 3-dimensional map. In this way, we can analyze patterns encoded in these word vectors and interpolate the relationship between words. Since a word vector generally has hundreds of dimensions in practical applications, we need dimension reduction techniques to map it to 2 or 3-dimensional data for visualization. One method is PCA which seeks a linear mapping from a high-dimensional space to a low-dimensional space (see

---

<sup>21</sup>In self-paced reading, the text is segmented into words (or phrases), and the participant is asked to press a button to request the display of a segment.

<sup>22</sup>A linguistic representation can be seen as a feature vector that is manually built on a linguistic resource (such as a dictionary).

Section 3.3.3). Another well-known method is **t-distributed stochastic neighbor embedding (t-SNE)** [Hinton and Roweis, 2002; Van der Maaten and Hinton, 2008]. t-SNE is a non-linear dimension reduction method, and has been widely used in visualizing high-dimensional data. Apart from these, one can consider the methods presented in Section 3.3.3 as well as those tailored for visualizing word vectors [Zhang et al., 2019b; Liu et al., 2017].

## 3.7 Summary

In this chapter we discussed two interesting problems in NLP: tokenization and word (or token) representation. First, we introduced models for dividing a sentence into units that are meaningful and/or well suited for downstream tasks. Second, we introduced the idea of word vector models with particular attention to learning both count-based high-dimensional models and real-valued low-dimensional models. While most of these models are simple, they are often used in complex NLP systems and form the basis of many advanced models, as will be shown in the following chapters.

Tokenization (or segmentation) is an important “operation” in NLP, commonly as a pre-processing step for many applications [Webster and Kit, 1992]. However, the use of the term *tokenization* is somewhat misleading because it originally refers to a process of dividing a string into substrings and is more often used as a general computer science term. In NLP, tokenization can draw on concepts and results from several sub-fields. On the linguistics side, tokenization is highly related to two fundamental questions: how words are composed and how words form sentences. It is therefore natural to use theories and methods of morphology and syntax to define the basic units of a language, leading to many rule-based tokenization systems covering a variety of languages. On the machine learning side, tokenization has long been cast as a problem of learning token boundaries from data in either a supervised or unsupervised manner [Mielke et al., 2021]. A common approach is to first annotate some tokenized text with human knowledge about what basic language units should be, and then learn to tokenize on this annotated data (see Section 3.1.3). More recently, learning tokenizers without linguistic constraints has been found to be promising (see Section 3.1.4). Since natural languages are themselves sets of characters or byte sequences, it is also possible to segment a sentence into characters or bytes [Ling et al., 2015; Lee et al., 2017]. The tokenization-free method in general may help when one wants a language-independent tokenizer and a simpler pipeline for processing the text.

From a more mathematical perspective, tokenization can be thought of as a mapping from the input data to a sequence of variables. In this way, the concept of tokenization can be generalized by relaxing the assumption that both the input and output variables are constrained to discrete values. In recent image and speech processing systems, for example, researchers try to transform continuous input data (such as pixels and acoustic signals) into a sequence of vector-based “tokens” [Schneider et al., 2019; Dosovitskiy et al., 2021]. Some interesting extensions of these ideas are even to transform image and speech data to a sequence of indices, leading to approaches bearing a closer relation to NLP [Oord et al., 2017; Baevski et al., 2020; Hsu et al., 2021].

Given that the input text is divided into smaller pieces, a natural next step is to represent these pieces in some way that captures their underlying features. While representing language units as vectors of numbers has been the de facto standard for the development of recent NLP systems, the work on vector representation dates back to the very early days of computational linguistics. According to many popular textbooks and papers [Manning and Schütze, 1999; Jurafsky and Martin, 2008], the idea of using a distribution to represent word meaning, also known as **distributional semantics**, started in the 1950s with the rise of empiricism. At the time, most of the work was influenced by Harris’s distributionalism [Harris, 1954] and related work [Firth, 1957; Wittgenstein, 1953]. In parallel, Osgood [1952] proposed to define the meaning of a concept as a point in a multidimensional space in a psychological manner. All these ideas greatly influenced the way linguistics and NLP people think of word meaning in the following decades.

Modern approaches to distributional semantics appeared in the 1990s, mainly as a result of the revival of empiricism in artificial intelligence [Church, 2011]. Most of these were driven by the distributional hypothesis: words having similar meanings are more likely to occur in similar contexts. In response, a number of methods were developed, differing in the way the contexts are modeled. For example, a context can be the words in a context-window, or the words with a relation to the given word in a syntax tree. Apart from those mentioned in Section 3.3, methods that are not covered in this chapter include hyperspace analogue of language (HAL) [Lund and Burgess, 1996], distributional memory [Baroni and Lenci, 2010], dependency-based semantic space models [Padó and Lapata, 2007], and so on. For comprehensive descriptions of distributional semantics models, the reader can refer to papers that survey this topic [Lenci, 2018; Mitchell and Lapata, 2010]. Note that most of the above-mentioned work can be thought of as instances of the vector space model which can deal with problems beyond lexical semantics. For example, in compositional distributional semantics, the meaning of a phrase or a sentence can be represented as a vector obtained by performing simple algebraic operations on the word vectors [Clark et al., 2008; Mitchell and Lapata, 2010; Blacoe and Lapata, 2012].

While distributional models have attracted attention in the NLP community for many years, word embedding models that learn low-dimensional, real-valued word vectors directly from texts have been a predominant approach recently. As described in Sections 3.4-3.5, models of this type do not depend on strong assumptions like the distributional hypothesis, but learn to represent a word as a vector of hidden attributes (or features) describing the word. The resulting model is an extension of the feature-based semantic model [Markman, 2013]. A recognized difference with traditional feature-based methods is that we do not need to manually define the features. We instead take these features as parameters of the model, and train them in the way as in common (supervised) machine learning systems.

Formulating word representation as an end-to-end learning problem brings with it several benefits. One of the benefits is that new features can be found because no constraints are placed on how these features are learned and interpreted. On the other hand, as shown in Section 3.6.2, the word vectors obtained in this way indeed show some linguistic properties, though the word embedding models are not trained to achieve this. Another benefit is that the word embedding models also fall in the vector space models in NLP, enabling the easy

use of word vectors in various applications. There are also many examples of methods that attempt to improve standard word embedding systems. For example, researchers have tried to incorporate additional linguistic information into word vectors [Levy and Goldberg, 2014a; Cotterell and Schütze, 2015; Tissier et al., 2017], and to learn universal word vectors across multiple languages [Klementiev et al., 2012; Mikolov et al., 2013b; Ammar et al., 2020; Smith et al., 2017; Artetxe et al., 2017].

Widely associated with neural models in NLP, the idea of distributed representation has been successfully applied to problems beyond word representation, e.g., sentence representation [Le and Mikolov, 2014; Kalchbrenner et al., 2014; Kiros et al., 2015; Hill et al., 2016; Arora et al., 2017; Lin et al., 2017; Conneau et al., 2017b], tree/graph-structure representation [Socher et al., 2011; Perozzi et al., 2014; Tai et al., 2015; Grover and Leskovec, 2016], and so on. In particular, contextualized representations of words, though not discussed in this chapter, are generally appreciated for modeling sequential data [McCann et al., 2017; Peters et al., 2018; Devlin et al., 2019].

# Chapter 4

## Recurrent and Convolutional Sequence Models

*The whole is more than the sum of its parts.*

–Aristotle, 384-322 BC [Ross, 1924]

Aristotle might or might not think of linguistic phenomena when having this thought, but it is indeed something we want to express in this chapter: there is something different in a sentence or phrase besides words. Of course, words have meanings, alone. However, when they come together to form a sentence or phrase, the meaning of the whole could be much more complex and diverse. This leads to the most beautiful aspect of language that human beings can express any meaning using a finite set of elements (e.g., words or characters).

The infinite and non-compositional nature of language makes it more difficult to model a sequence of words than to model individual words. A difficulty is that a word may repeatedly alter its meaning in different contexts. Taking the idea of word embedding that a word can be represented as a low-dimensional, real-valued vector, the “meaning” of a language unit could be continuous. It is therefore possible to extend methods of distributed representation from words to sequences of words. This leads us to explore models in which the process of dealing with variable-length word sequences is fundamentally continuous.

Here we consider the general approach to learning the distributed representation of word sequences. In particular, we consider recurrent and convolutional neural networks which have been extensively used in many fields ranging from speech processing to computer vision. For natural language inputs, the result of applying these models is a sequence-level representation of the input. The representation could be either a single real-valued vector, or a sequence of such vectors, each corresponding to a contextualized representation for an input word of the input sequence. Such a model of representation, that can broadly be called an **encoder**, is generally used with a variety of systems whose input is sequential data. We will see several examples of it in this chapter.

## 4.1 Problem Statement

For many NLP applications, our objective is to make predictions based on an input sequence. Let us consider again the text classification problem mentioned in Chapter 1. If we obtain a text that may talk about food or not, we want to assign one of the two classes to it (say Food or Not-food). To do this, a common method of classification is to represent the text as a bag of features, denoted as  $\mathbf{H}$ . Then, a probability is assigned to each of the classes using a probabilistic model  $\Pr(y|\mathbf{H})$ . The predicted class is the one that has the maximum probability  $\hat{y} = \arg \max_y \Pr(y|\mathbf{H})$ .

While this is a standard procedure for classification, the underlying idea can be used to describe a general problem. Formally, let  $\mathbf{w} = w_1 \dots w_m$  be a sequence of words<sup>1</sup>. A sequence-level NLP system can be formulated as a function that maps the sequence  $\mathbf{w}$  to some output  $\mathbf{y}$ . This can be divided into two steps, called the representation (or encoding) step and the prediction step.

- **Representation (or Encoding).** It transforms the input sequence  $\mathbf{w}$  to some “features”  $\mathbf{H}$  by using an encoder  $\text{Enc}(\cdot)$ :

$$\mathbf{H} = \text{Enc}(\mathbf{w}) \quad (4.1)$$

- **Prediction.** A predictor  $\text{Predict}(\cdot)$  takes  $\mathbf{H}$  and generates an output:

$$\mathbf{y} = \text{Predict}(\mathbf{H}) \quad (4.2)$$

A simple form of  $\mathbf{H}$  is a feature vector. For example,  $\mathbf{H}$  could be a set of human-designed indicator features extracted from  $\mathbf{w}$  (as a high-dimensional sparse representation), or a set of real numbers indicating some latent features (as a low-dimensional dense representation). In NLP, another common form of  $\mathbf{H}$  is a sequence of vectors in which each vector  $\mathbf{h}_i$  corresponds to an input word  $w_i$  (see Figure 4.1). In this case,  $\mathbf{h}_i$  can be viewed as a “new” representation of both  $w_i$  and its context in  $\mathbf{w}$ <sup>2</sup>. The correspondence between  $\mathbf{h}_i$  and  $w_i$  enables the representation to make distinctions among different positions of the sequence, and more importantly, to vary its modeling power for variable-length inputs.

The form of  $\mathbf{y}$  is dependent on the problem we intend to deal with. For example, for classification problems,  $\mathbf{y}$  is the index of a class (or a distribution of classes); for regression problems,  $\mathbf{y}$  is a real number; for translation problems,  $\mathbf{y}$  is a sequence of words in another language, and so on. Note that, in the above model, representation and prediction can be regarded as two separate problems. A great advantage of isolating representation and prediction is that we can use the same encoder in many applications with different predictors. This also motivates a promising line of research in which a general-purpose encoder is trained on large-scale data and then used as components in different downstream systems [Peters et al., 2018;

<sup>1</sup>Although we restrict ourselves to word sequences for discussion, the methods can be used to deal with sequences of any language units, e.g., sub-words, characters, etc.

<sup>2</sup>This architecture can be extended to encoders in which the input and output have different lengths, say, the input is  $w_1 \dots w_m$  and the output is  $\mathbf{h}_1 \dots \mathbf{h}_n$  ( $m \neq n$ ).

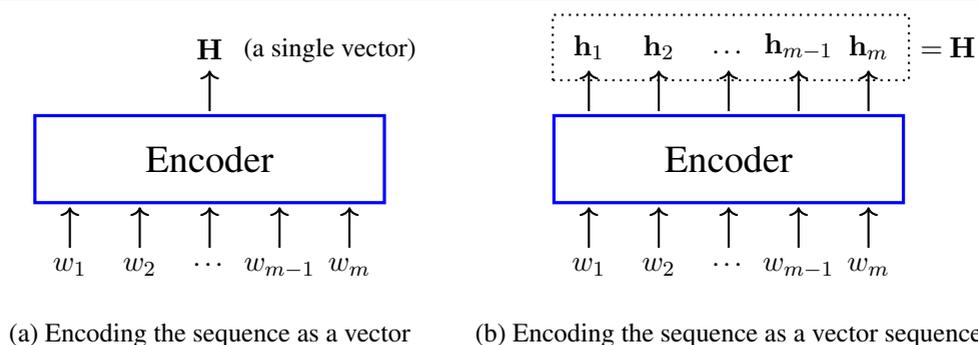


Figure 4.1: Representing a word sequence as (a) a vector or (b) a sequence of vectors.

Devlin et al., 2019].

There are many possible forms for  $\text{Enc}(\cdot)$  and  $\text{Predict}(\cdot)$ . For text classification, for example, one way is to define  $\text{Enc}(\cdot)$  as a function computing a feature vector using a set of hand-crafted feature templates, and define  $\text{Predict}(\cdot)$  as a statistical classification model (such as SVMs and maximum entropy-based models). Another way is to define  $\text{Enc}(\cdot)$  as a multi-layer neural network that outputs a real-valued vector, and define  $\text{Predict}(\cdot)$  as a simple neural network that involves only one Softmax layer. In this chapter we will focus on neural network-based encoders. We will show that such a type of encoder could be applied to a number of NLP tasks in Section 4.5.

## 4.2 Recurrent Models

A study of various sequence models is not easy work. It is convenient, however, to first introduce one of the most common and practical neural models, called recurrent neural networks (RNNs). We will see later that RNNs are extensively used in sequence modeling, and the techniques presented here are generic and applicable to many systems.

### 4.2.1 An RNN-based Language Model

Perhaps the most popular use of sequence models in NLP is estimating the probability of a word sequence, also known as language modeling. Mathematically, language modeling is an instance of a well-known problem in the field of **stochastic processes** (or **random processes**): the problem of modeling **time series** data [Hamilton, 1994; Chatfield, 2003; Fuller, 2009]. As a time series, a sequence of words can be treated as a sequence of data points at time intervals that are equally spaced. In this sense, the methods we present here are somewhat general, although the discussion on a broader range of time series problems is beyond the scope of this book.

Given a sequence of words  $w_1 \dots w_m$ , the goal of language modeling is to compute  $\Pr(w_1, \dots, w_m)$ . This joint probability is typically written as a product of conditional probabili-

ties using the chain rule:

$$\Pr(w_1, \dots, w_m) = \Pr(w_1) \cdot \Pr(w_2|w_1) \cdots \Pr(w_m|w_1, \dots, w_{m-1}) \quad (4.3)$$

In other words, the problem of generating  $w_1 \dots w_m$  is the same as the problem of generating a word  $w_{i+1}$  at a time based on the previous words  $w_1 \dots w_i$ . RNN-based language models represent  $w_1 \dots w_i$  via a recurrent unit  $\text{RNN}(\cdot)$  [Mikolov et al., 2010], like this

$$\mathbf{h}_i = \text{RNN}(\mathbf{h}_{i-1}, \mathbf{x}_i) \quad (4.4)$$

where  $\mathbf{x}_i \in \mathbb{R}^{d_e}$  is the word vector (or word embedding) for  $w_i$ . Let  $V$  be the vocabulary from which we can choose a word. If  $w_i \in \mathbb{R}^{|V|}$  is a one-hot word representation<sup>3</sup>,  $\mathbf{x}_i$  is given by multiplying  $w_i$  with the word embedding table  $\mathbf{C} \in \mathbb{R}^{|V| \times d_e}$ :

$$\begin{aligned} \mathbf{x}_i &= \text{Embed}(w_i) \\ &= w_i \mathbf{C} \end{aligned} \quad (4.5)$$

As shown in Chapter 1, the use of  $\mathbf{C}$  transforms a  $|V|$ -dimensional (and probably high-dimensional) vector to a  $d_e$ -dimensional (and probably low-dimensional) vector. Note that  $\mathbf{C}$  is essentially a lookup table, with a distinct table entry (i.e., a row) for each word in  $V$ . So, the right-hand side of Eq. (4.5) is in practice a function that selects a row from  $\mathbf{C}$  with the word index.

Now we go back to Eq. (4.4). The equation is not difficult to understand: the state of the context we have seen so far (i.e.,  $\mathbf{h}_i$ ) is some representation of the combination of the current input (i.e.,  $\mathbf{x}_i$ ) and the state of the earlier context ( $\mathbf{h}_{i-1}$ ). Put another way, it can be thought of as a process of repeatedly adding information of a new word to a cache of “history”. An elegant aspect of this process is that it can be easily implemented by running Eq. (4.4) a number of times until the end of the sequence.

$\text{RNN}(\cdot)$  can be any function that takes  $\mathbf{h}_{i-1}$  and  $\mathbf{x}_i$ , and produces a new vector  $\mathbf{h}_i$ . The vanilla RNN has a form

$$\text{RNN}(\mathbf{h}_{i-1}, \mathbf{x}_i) = \psi(\mathbf{h}_{i-1} \mathbf{U} + \mathbf{x}_i \mathbf{V}) \quad (4.6)$$

where  $\psi(\cdot)$  is an activation function, such as  $\text{TanH}(\cdot)$  and  $\text{Sigmoid}(\cdot)$ . Together with Eqs. (4.4) and (4.5), we can define  $\mathbf{h}_i$  as a function of  $\mathbf{h}_{i-1}$  and  $w_i$

$$\mathbf{h}_i = \psi(\mathbf{h}_{i-1} \mathbf{U} + w_i \mathbf{C} \mathbf{V}) \quad (4.7)$$

where  $\mathbf{U} \in \mathbb{R}^{d_h \times d_h}$ ,  $\mathbf{V} \in \mathbb{R}^{d_e \times d_h}$ , and  $\mathbf{C} \in \mathbb{R}^{|V| \times d_e}$  are learnable parameters of the model, and  $d_h$  is a hyper-parameter indicating the number of dimensions of  $\mathbf{h}_i$  and  $\mathbf{h}_{i-1}$ .

We now have an encoder that represents the word sequence  $w_1 \dots w_m$  as a sequence of

---

<sup>3</sup>The one-hot representation  $w_i$  is a  $|V|$ -dimensional vector in which only one entry is 1 and all other entries are zeros. Following the notation used throughout this book, a vector is in general represented as a variable in bold text. Here we treat  $w_i$  as a word index and interchangeably use it with the one-hot representation.

RNN's outputs  $\mathbf{H} = \{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ . Given that each  $\mathbf{h}_i$  encodes the sub-sequence spanning from  $w_1$  to  $w_i$ , we can place a Softmax layer on  $\mathbf{h}_i$  to obtain a distribution of words:

$$\mathbf{y}_{i+1} = \text{Softmax}(\mathbf{h}_i \mathbf{O} + \mathbf{b}) \quad (4.8)$$

where  $\mathbf{O} \in \mathbb{R}^{d_h \times |V|}$  and  $\mathbf{b} \in \mathbb{R}^{|V|}$ . Taking the word index  $w_{i+1}$ , we have

$$\Pr(w_{i+1}|w_1, \dots, w_i) = y_{i+1}(w_{i+1}) \quad (4.9)$$

Thus, we have developed a language model that produces a probability  $\Pr(w_{i+1}|w_1, \dots, w_i)$  at each step. Figure 4.2 shows an illustration of the RNN-based language model for an example sequence. To run this model on a word sequence, we surely wish to start with predicting  $w_1$  but this requires a preceding word  $w_0$  that is taken as the input. A simple and widely applicable method for giving an appropriate starting state to RNNs is to add a beginning symbol  $\langle \text{SOS} \rangle$  to the sequence so that all sequences start with the same “word”. Likewise, we can attach an end symbol  $\langle \text{EOS} \rangle$  to the sequence to model the completeness of the sequence. This leads to a new form of the probability of the sequence

$$\begin{aligned} \Pr(\langle \text{SOS} \rangle, w_1, \dots, w_m, \langle \text{EOS} \rangle) &= \Pr(\langle \text{SOS} \rangle) \cdot \\ &\Pr(w_1 | \langle \text{SOS} \rangle) \cdot \\ &\Pr(w_2 | \langle \text{SOS} \rangle, w_1) \cdot \\ &\dots \\ &\Pr(w_m | \langle \text{SOS} \rangle, w_1, \dots, w_{m-1}) \cdot \\ &\Pr(\langle \text{EOS} \rangle | \langle \text{SOS} \rangle, w_1, \dots, w_m) \end{aligned} \quad (4.10)$$

We can simply assume  $\Pr(\langle \text{SOS} \rangle) = 1$ . To obtain  $\Pr(\langle \text{SOS} \rangle, w_1, \dots, w_m, \langle \text{EOS} \rangle)$ , we take  $\langle \text{SOS} \rangle w_1 \dots w_m$  as an input sequence and  $w_1 \dots w_m \langle \text{EOS} \rangle$  as the output sequence.

## 4.2.2 Training

As a neural network, the RNN-based language model can be trained in a regular way. The training problem has been well discussed in Chapter 2. So, we do not give a full description in this chapter, but a little bit about its basic idea as well as some refinements.

RNN-based language modeling can be framed as a next-step-prediction problem. Suppose we are given a collection of word sequences  $S$ . For each sequence  $\mathbf{w} = w_1 \dots w_{|\mathbf{w}|}$  in  $S$ , we have a sequence of pairs of an input word and the corresponding gold-standard answer, like this<sup>4</sup>

$$\{(w_1, w_2), (w_2, w_3), \dots, (w_{|\mathbf{w}|-1}, w_{|\mathbf{w}|})\}$$

The language model takes the input sequence  $w_1 \dots w_{|\mathbf{w}|-1}$  and returns a sequence of

<sup>4</sup>While the  $\langle \text{SOS} \rangle$  and  $\langle \text{EOS} \rangle$  tricks are generally considered in real-world systems, we drop the  $\langle \text{SOS} \rangle$  and  $\langle \text{EOS} \rangle$  symbols from now on for simplification.

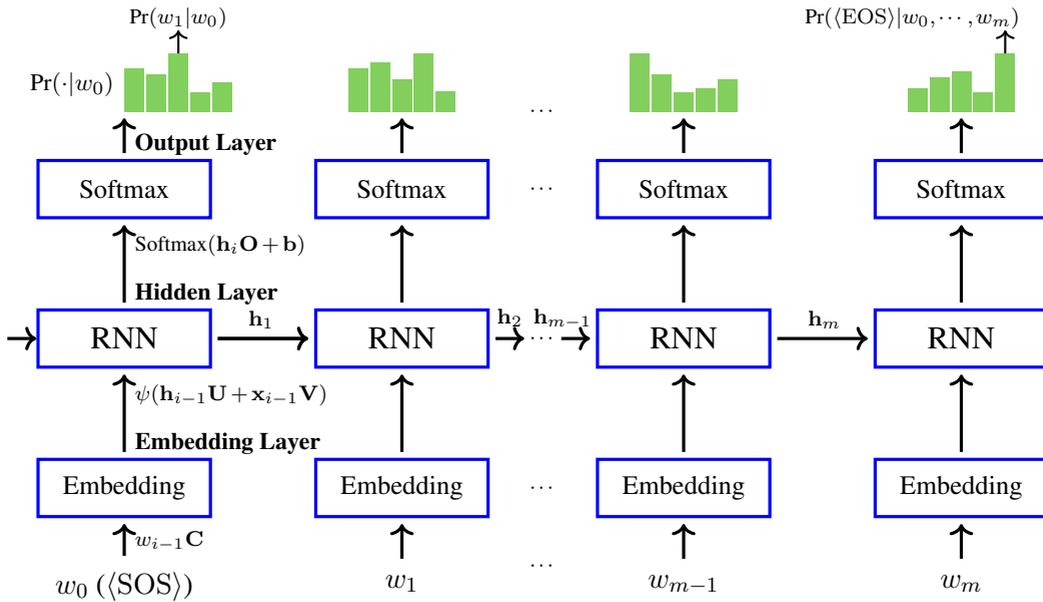


Figure 4.2: Illustration of using an RNN-based language model to calculate  $\Pr(\langle \text{SOS} \rangle w_1 \dots w_m \langle \text{EOS} \rangle)$ . The input is  $\langle \text{SOS} \rangle w_1 \dots w_m$ , and the output is the probability  $\Pr(w_1 | \langle \text{SOS} \rangle) \Pr(w_2 | \langle \text{SOS} \rangle w_1) \dots \Pr(\langle \text{EOS} \rangle | \langle \text{SOS} \rangle w_1 \dots w_m)$ . As  $\Pr(\langle \text{SOS} \rangle) = 1$ , the probability of generating the sequence is simply  $\Pr(\langle \text{SOS} \rangle w_1 \dots w_m \langle \text{EOS} \rangle) = \Pr(\langle \text{SOS} \rangle) \Pr(w_1 | \langle \text{SOS} \rangle) \Pr(w_2 | \langle \text{SOS} \rangle w_1) \dots \Pr(\langle \text{EOS} \rangle | \langle \text{SOS} \rangle w_1 \dots w_m)$ . For each input  $w_i$ , we first represent it as a word vector  $\mathbf{x}_i$  via the embedding layer, resulting in a sequence of word vectors  $\mathbf{x}_0 \dots \mathbf{x}_m$ . The RNN layer maps  $\mathbf{x}_0 \dots \mathbf{x}_m$  to a sequence of hidden states  $\mathbf{h}_1 \dots \mathbf{h}_{m+1}$ . In this process, we repeat the same thing: an RNN unit takes both  $\mathbf{h}_{i-1}$  and  $\mathbf{x}_i$  and produces a new state  $\mathbf{h}_i$ . On top of that, we use the output layer (Softmax) to obtain  $\Pr(w_{i+1} | \langle \text{SOS} \rangle w_1 \dots w_i)$ .

distributions  $\mathbf{y}_2 \dots \mathbf{y}_{|\mathbf{w}|}$ . See the following table for an illustration of the inputs and outputs of the model.

Step	History	Input (One-hot)	Output (Distribution)	Gold-Standard
1		$w_1$	$\mathbf{y}_2$	$w_2$
2	$w_1$	$w_2$	$\mathbf{y}_3$	$w_3$
3	$w_1, w_2$	$w_3$	$\mathbf{y}_4$	$w_4$
...	...	...	...	...
$ \mathbf{w}  - 2$	$w_1, w_2, \dots, w_{ \mathbf{w} -3}$	$w_{ \mathbf{w} -2}$	$\mathbf{y}_{ \mathbf{w} -1}$	$w_{ \mathbf{w} -1}$
$ \mathbf{w}  - 1$	$w_1, w_2, \dots, w_{ \mathbf{w} -3}, w_{ \mathbf{w} -2}$	$w_{ \mathbf{w} -1}$	$\mathbf{y}_{ \mathbf{w} }$	$w_{ \mathbf{w} }$

A loss function  $L(\mathbf{y}_i, w_i)$  is defined to measure how many “errors” we will make if we use  $\mathbf{y}_i$  instead of the one-hot representation  $w_i$ . A common choice is the cross-entropy loss which computes the divergence of a distribution from another [Mitchell, 1997; Bishop, 2006].

Then, the loss over the entire set is defined to be

$$L = \sum_{\mathbf{w} \in S} \sum_{i=2}^{|\mathbf{w}|} L(\mathbf{y}_i, w_i) \quad (4.11)$$

Once we know the loss, the training of the RNN-based language model can be achieved by using gradient descent. A simple form of this method is the delta rule

$$\theta_{\text{new}} = \theta_{\text{old}} - lr \cdot \frac{\partial L}{\partial \theta} \quad (4.12)$$

where  $\theta$  stands for the parameters. For the model described in Section 4.2.1,  $\theta$  includes  $\mathbf{C}$ ,  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{O}$  and  $\mathbf{b}$ .  $\frac{\partial L}{\partial \theta}$  is the derivative of the loss with respect to the parameters, called **error gradient**.

Eq. (4.12) can be understood as a process of moving the current parameters a small step in the steepest downhill direction (i.e., the direction of  $-\frac{\partial L}{\partial \theta}$ ). Here  $lr$  stands for how far we move in each step of going downhill, also called the learning rate. Obtaining  $\frac{\partial L}{\partial \theta}$  often requires a back-propagation process that flushes the error gradient from the output to the input. In modern implementations of deep learning systems, in which neural networks are represented as computation graphs, back-propagation is simple since it is just a by-product of graph traversal and there are many automatic differentiation toolkits to do this. Similar algorithms, called **back-propagation through time (BPTT)**, were also used in earlier systems [Werbos, 1990]. For further information about training neural networks, see Chapter 2 and/or textbooks on this subject [Goodfellow et al., 2016; Zhang et al., 2021].

If the input is a long sequence, the application of RNNs would result in a deep neural network. In this case, the use of the chain rule of ordered derivatives makes large or small loss derivatives accumulate, and the update to the parameters in Eq. (4.12) is consequently very large or small. These are typically known as the **exploding and vanishing gradient problems**. There are several methods to mitigate these problems for RNNs [Sutskever, 2013]. Some of them are

- **Regularization.** Introducing regularization terms (such as the  $l_1$  and  $l_2$  norms on parameter matrices) into training can avoid models in which most of the parameters have large values, and thus help to avoid exploding gradients. Similarly, one can penalize the cases in which the norms of the gradients are too small [Pascanu et al., 2013].
- **Gradient Clipping.** When the norm of the gradients is too large, it is natural to directly scale down their magnitudes. A simple method is to clip the gradient norm in terms of a threshold  $\tau$ . If the norm  $\|\frac{\partial L}{\partial \theta}\|$  is larger than  $\tau$ , we can rescale  $\frac{\partial L}{\partial \theta}$  accordingly, say,  $\frac{\partial L}{\partial \theta} = \frac{\tau}{\|\frac{\partial L}{\partial \theta}\|} \cdot \frac{\partial L}{\partial \theta}$ .<sup>5</sup>

<sup>5</sup>It is usually formulated as an equation

$$\frac{\partial L}{\partial \theta} = \frac{\tau}{\max(\tau, \|\frac{\partial L}{\partial \theta}\|)} \cdot \frac{\partial L}{\partial \theta} \quad (4.13)$$

- **Truncated Back-propagation.** Another idea is to break a long sequence of input-output pairs into shorter pieces, and train RNNs on these separate sub-sequences [Williams and Peng, 1990; Elman, 1990]. This reduces both the cost of training and the risk of too large or small values in accumulating error gradients.
- **Improved Architectures.** It is also possible to redesign the model to overwhelm the limits of standard RNNs, usually using the memory mechanism. In Section 4.3, we will see a few examples of redesigning the recurrent unit for addressing the vanishing gradient problem.
- **Initialization and Constraints of Parameters.** Initializing the model parameters to a desirable region is generally helpful for optimization, and, sometimes, helpful for preventing very small gradients. An alternative method is to randomly set the model and only learn the parameters of the output layers [Jaeger and Haas, 2004].
- **Non-saturating Activations.** Many common activation functions have a compact range of outputs, e.g., the Sigmoid function has a range of  $[0, 1]$ . They are also called **saturating activation functions**<sup>6</sup>. The use of saturating activation functions often leads to the decay of gradients over layers, i.e., the vanishing gradient. It is therefore promising to use non-saturating activation functions instead, e.g., the ReLU function.
- **Normalization of Activations.** Saturating activations may also result in getting stuck in a saturated region of outputs, and we need a large learning rate to escape from local optimums [Ioffe and Szegedy, 2015]. Thus, the training would be unstable, and subtle changes in inputs and/or model parameters would lead to a big variance in model behavior. A possible solution is to normalize the activations to reduce the variance, e.g., subtracting the mean of the activations in a group of samples (e.g., samples in a mini-batch of training), and dividing by their standard derivation.

### 4.2.3 Layer Stacking

If we think of the application of a recurrent unit as a function mapping a variable sequence to a new variable sequence of the same length, it is natural to compose this function with another function of the same type, or even with itself. This makes it very easy to extend RNNs to deep neural networks: all you need is to stack RNNs.

Let  $\mathbf{h}_i^l$  be the output of the  $l$ -th recurrent unit in the stack at position  $i$ . We can apply a new recurrent unit to  $\mathbf{h}_i^l$ , resulting in a new output at level  $l + 1$

$$\mathbf{h}_i^{l+1} = \text{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) \quad (4.14)$$

where  $\mathbf{h}_{i-1}^{l+1}$  is the output of the previous step at level  $l + 1$ . To make Eq. (4.14) well-formed, we typically define  $\mathbf{h}_i^0 = \mathbf{x}_i$ . In other words, the stack starts off with the word vector  $\mathbf{x}_i$ , then a series of RNN outputs (i.e.,  $\mathbf{h}_i^1, \mathbf{h}_i^2, \mathbf{h}_i^3$ , etc).

To illustrate, Figure 4.3 (a) shows a stacked RNN for language modeling. We see that

---

<sup>6</sup>An activation function  $f(x)$  is non-saturating if and only if when  $x \rightarrow \infty$  (or  $-\infty$ ),  $f(x) \rightarrow \infty$ . An activation function is saturating if it is not a non-saturating activation function.

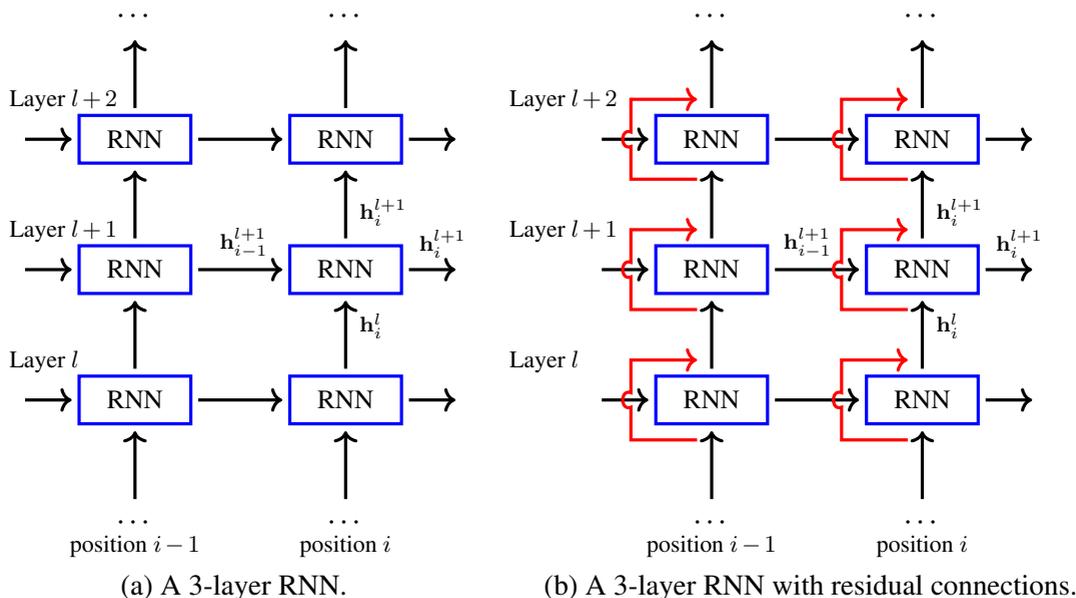


Figure 4.3: 3-layer RNNs (with and without residual connections). To stack RNN layers, we feed the output of layer  $l$  to layer  $l+1$ . Thus the output of layer  $l+1$  is given by  $\mathbf{h}_i^{l+1} = \text{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l)$ . Lines in red color stand for the residual connections which directly add the input of a layer to its output, resulting in  $\mathbf{h}_i^{l+1} = \text{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) + \mathbf{h}_i^l$ .

applying a stack of recurrent units is equivalent to creating multiple layers of RNNs simultaneously. However, there would be a risk of confusion if we call an unrolled recurrent network a *layer*, as the term *layer* typically refers to a set of neurons receiving the same inputs in a feed-forward neural network. Here we extend the term *layer* to cover a more general concept: a group of neurons that are topologically placed on the same level. So, we say that the language model in Figure 4.3 has 3 RNN layers.

Stacking multiple layers of RNNs, we build a model which is deeper but more difficult to train. This difficulty arises in part from the barriers of passing information through many-layered RNNs. To make the training easier, a widely-used approach is to introduce **skip connections** or **residual connections** into a multi-layer neural network [He et al., 2016a]. These connections are intended to leverage an additional path to allow information to skip layers. As described in Chapter 2, the form of a residual neural network is given by

$$\mathbf{y}^{l+1} = F(\mathbf{y}^l) + \mathbf{y}^l \quad (4.15)$$

where  $\mathbf{y}^l$  is the output of layer  $l$ . Extending this formulation to Eq. (4.14) leads to multi-layer RNNs with residual connections, given by

$$\mathbf{h}_i^{l+1} = \text{RNN}(\mathbf{h}_{i-1}^{l+1}, \mathbf{h}_i^l) + \mathbf{h}_i^l \quad (4.16)$$

The only difference from Eq. (4.14) is that we introduce the identity map of  $\mathbf{h}_i^l$  to the right-hand side of Eq. (4.16). Thus, the input  $\mathbf{h}_i^l$  is directly accessible from layer  $l + 1$ . This greatly simplifies the way that the information flows through the neural network, and allows the system to “skip” layers in propagating errors. Figure 4.3 (b) shows a 3-layer RNN with residual connections.

#### 4.2.4 Bi-directional Models

The use of RNNs enables us to formulate the problem of encoding a word sequence as a problem of left-to-right generation of words. One advantage of this approach is that the modeling of context words arises naturally: the output of an RNN unit in some way describes the history words up to that point. This feature makes it very straightforward to model the probability distribution  $\Pr(w_{i+1}|w_1, \dots, w_i)$ , as we can use  $\mathbf{h}_i$  as a representation of the context  $w_1 \dots w_i$ , that is,  $\Pr(w_{i+1}|w_1, \dots, w_i) = \Pr(w_{i+1}|\mathbf{h}_i)$ .

The left-to-right generation is widely used in sequence generation, such as machine translation. It can be viewed as an instance of **autoregressive processes (AR processes)** in which the state of a variable is dependent on the state of the previous variables [Chatfield, 2003; Box et al., 2015]<sup>7</sup>. However, such a method is not the only choice for modeling sequences. We do not even necessarily restrict ourselves to language modeling for training a sequence encoder. This gives rise to an interesting question: how can we develop an encoder of word sequences without assumptions regarding the predictor? Answering the question leads us to isolate the learning of the text encoder from a specific NLP task, and to regard it as a separate task whose result can be applied to many other systems. A more detailed discussion is not the focus here and we leave it to subsequent chapters.

We now present a simple extension of the left-to-right sequence model by returning to RNNs. Note that in sequence modeling our desire is some representation of the entire sequence. A problem with usual RNNs is that they are **uni-directional models** in which the context words following  $w_i$  are absent. To consider both the left and right contexts of a given word, we can instead use **bi-directional models**. Figure 4.4 shows an example of the bi-directional RNN. There are two sub-models: a left-to-right RNN and a right-to-left RNN. They have the

<sup>7</sup>As a stochastic process, an autoregressive process expresses a variable at time  $t$  by relating it to the past values of the process and the current value of an error process [Chatfield, 2003]. Formally, a time series  $\{z_1, \dots, z_T\}$  describes an autoregressive process of order  $p$  if for any  $t \in \{p + 1, \dots, T\}$

$$z_t = \sum_{i=1}^p \alpha_i z_{t-i} + \epsilon_t \quad (4.17)$$

where  $\{\alpha_1, \dots, \alpha_p\}$  are the parameters of the process, and  $\epsilon_t$  is the error at time  $t$ . This process is called *regressive* because it has the same form as the **multiple linear regression model**. The prefix *auto-* comes from the way we regress  $z_t$ :  $z_t$  is dependent on its past values instead of additional independent variables. One way to interpret language modeling in an autoregressive process perspective is to simply treat  $\{z_1, \dots, z_T\}$  as representations of a sequence of words  $\{w_1, \dots, w_T\}$ . Thus, we can gain some idea of predicting  $w_t$  using previous words  $\{w_1, \dots, w_t\}$  by considering the autoregressive property of the problem. However, it should be noted that most of the sequence generation models used in NLP are not mathematically equivalent to Eq. (4.17), although they are often called regressive models. For example, the RNN-based language model discussed here is not a linear model. Rather, it takes layers of non-linearity to describe the complex relationships among words.

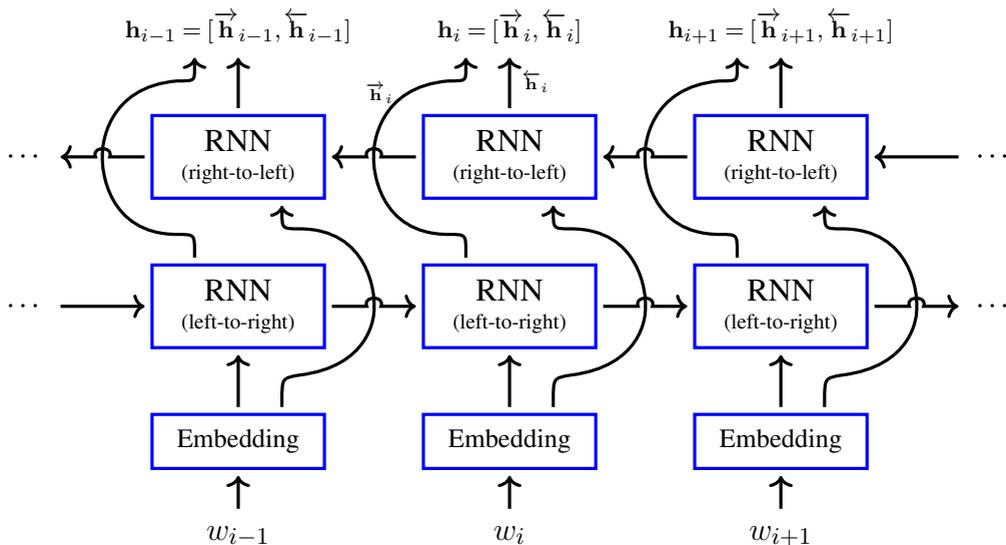


Figure 4.4: A bi-directional RNN model. Given a word sequence, we run an RNN from left to right and another RNN from right to left. Therefore, at each position we obtain a left-to-right representation and a right-to-left representation. The output is the concatenation of the two representations so that it involves both the left and right contexts.

same architecture but work in opposite directions. For each input word  $w_i$ , the left-to-right RNN outputs a vector representing the context  $\{w_1, \dots, w_i\}$  (denoted by  $\vec{\mathbf{h}}_i$ ), and the right-to-left RNN outputs a vector representing the context  $\{w_i, \dots, w_m\}$  (denoted by  $\overleftarrow{\mathbf{h}}_i$ ). We can concatenate  $\vec{\mathbf{h}}_i$  and  $\overleftarrow{\mathbf{h}}_i$  to obtain a bi-directional representation

$$\mathbf{h}_i = [\vec{\mathbf{h}}_i, \overleftarrow{\mathbf{h}}_i] \quad (4.18)$$

Thus, the bi-directional RNN has the same form of output as that of the uni-directional RNN, that is, a sequence of vectors  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ . Unlike the uni-directional RNN, the representation  $\mathbf{h}_i$  here describes the context on both sides.

For a stronger model, the bi-directional RNN can be extended to a neural network of multiple RNN layers. For example, we can run deep RNNs in two directions and combine their results as in Eq. (4.18). Such model architectures have been extensively used in language and speech processing tasks, including machine translation [Wu et al., 2016], sentiment analysis [Tang et al., 2015], POS tagging [Huang et al., 2015], speech recognition [Graves et al., 2013a;b], and so on.

## 4.3 Memory

RNNs can be appropriate for sequence learning in which we summarize at each step the past inputs and then make some prediction on this summary of the “history”. A benefit of RNNs is that we can represent a history of arbitrary length as a fixed-size vector, and update it when

new information arrives. In other words, we have a **memory**, though not explicitly defined, to store the information. Next we show that such a memory mechanism is general and can be used to improve sequence models.

### 4.3.1 Memory as A System

In psychology, memory is the ability of the mind to retain and recall information. There are many cognitive models of psychology. A well-known model is **the multi-store model** [Atkinson and Shiffrin, 1968]. It defines memory as a system consisting of three components: short-lived **sensory memory**, **short-term memory**, and **long-term memory**. The sensory memory retains the sensory information that is very quickly ceased, such as immediate data from the senses of sight and smell. The short-term memory stores information for a longer time but is not permanent. An example of the short-term memory is that we try to memorize a sequence of digits (e.g., a phone number) but may forget it after a short while. The long-term memory is permanent. This also means that the information is retained indefinitely. For example, adults can remember details of the events that occurred in their childhood.

Given this categorization, there appear to be interesting connections between the above model of memory and the neural networks we discuss here. For example, the state of a recurrent unit can be simply thought of as a short-term memory. It maintains information until we get to the end of a sequence and would be reset if we switch to a new sequence<sup>8</sup>. On the other hand, the entire language model and associated parameters perform more like a long-term memory: the language model is intended to learn and memorize some useful information about probabilistic word prediction from the text, so that it can be used whenever we want to. Moreover, there are other concepts that may stem from psychology but are used in several different fields. For example, **coding** or encoding is referred to as how the information is stored in a memory, **duration** is referred to as how long the information is stored in a memory, and **capacity** is referred to as how much information is stored in a memory.

In machine learning and NLP, we can gain an understanding of memory by considering it from an information processing point of view. Broadly, memory can be viewed as a system that writes information to a “storage” and reads it when queried. It has the following functions.

- **Encoding.** The input of the system is encoded in a form that is easy to process. For text inputs, this can be simply thought of as the same encoding process as we discuss in both Chapter 3 and this chapter: a word or a sequence of words is represented as a feature vector or a sequence of feature vectors.
- **Update.** Given the encoded information, we store it in the memory. This operation is generally dependent on the organization of the memory. For example, one can treat a group of encoded items as a datastore with an indexing system. In this case, storing an item requires finding the right place to keep it. Alternatively, one can represent the memory as a single vector of numbers.
- **Retrieval.** The stored information can be retrieved. This typically involves matching

---

<sup>8</sup>Another explanation is that the state of a recurrent unit at step  $i$  may contain little information about very early steps.

each item in the memory against an input query. If the memory is represented in a simpler form, such as a vector, it may not be explicitly retrieved, and we return the entire memory when required.

These functions can be designed in many different ways, leading to a variety of NLP systems. One simple example is information retrieval [Manning et al., 2008]. A typical information retrieval system indexes a large number of documents (or other resources) and allows users to search for interested information in this collection of documents. To enable search, documents are represented in forms that are convenient to use, for example, we may use the bag-of-words model to compute the matching score between a document and a query, and may use the inverted index to make an efficient mapping of a document to its location in the storage. Systems of this type cover a wide range of applications, including translation memory, dialogue, summarization, document classification, and so on.

Another design choice made for memory systems is to consider, either partially or fully, a continuous form for the above components. One method is to encode each input item as a real-valued vector (e.g., a word embedding) but use the same modules of update and retrieval as in usual information retrieval-like systems [Weston et al., 2015; Khandelwal et al., 2019]. An alternative method is to adopt differentiable functions for all the steps in building and accessing the memory. These models are typically implemented using neural networks and trained using gradient descent [Sukhbaatar et al., 2015; Graves et al., 2014; Kumar et al., 2016; Graves et al., 2016; Miller et al., 2016]. This idea motivates work on exploring approaches to coupling neural networks with memories, such as **end-to-end memory networks** and **neural Turing machines**. Note that the above models are sometimes called **external memories**, as they are used as separate modules working with other systems.

Memory can also work as an internal or hidden component of a system. In this case, the memory is typically rebuilt for each input sample, and so it can be regarded as an instance of the short-term memory. There are various ways of using this type of memory to improve sequence models. In the remainder of the section, we will focus on using the memory mechanism in RNNs. In Chapter 5, we will see how the idea of memory is extended to model the correspondence between tokens of two sequences.

### 4.3.2 Long Short-Term Memory

In the vanilla RNN presented in Section 4.2.1, the summarization of the context words was given by the output of a recurrent unit. It implicitly defines a memory, and thus enables the prediction based on past information for an arbitrary duration. The memory simply combines the representations of the earlier history  $w_1 \dots w_{i-1}$  and the input at the current step  $w_i$ , but does not consider how much information from different steps should be squeezed into a fixed-length representation. A problem with this model is that, if long-term dependencies are required for prediction, memory may provide little information about it, and it may be hard to learn these dependencies through back-propagation [Bengio et al., 1994; Pascanu et al., 2013]. A more powerful approach, therefore, is to compute what should be retained at each step, and to let the model learn to decide whether to memorize or forget.

**Long short-term memory (LSTM)** is perhaps the best-known variant of RNNs to accomplish the above goal [Hochreiter and Schmidhuber, 1997]. The basic idea of LSTM is that a recurrent unit can learn to memorize useful things and forget unuseful things by maintaining an explicit memory [Gers et al., 2000]. To this end, the vanilla recurrent unit is replaced with an LSTM unit that is made up of an output vector (call it a **recurrent cell**), a memory vector (call it a **memory cell**), and three gates to control the information flow inside the LSTM unit. As an extension to RNNs, an LSTM network deals with an input sequence as usual: it starts with some initial states, and then repeatedly takes an input and outputs a vector. A key difference between LSTM networks and RNNs is that the LSTM unit of step  $i$  takes both the recurrent cell and the memory cell of its previous step. The form of an LSTM unit is given by

$$(\mathbf{h}_i, \mathbf{c}_i) = \text{LSTM}(\mathbf{h}_{i-1}, \mathbf{c}_{i-1}, \mathbf{x}_i) \quad (4.19)$$

where  $\mathbf{h}_i \in \mathbb{R}^{d_h}$  is the recurrent cell of step  $i$ ,  $\mathbf{c}_i \in \mathbb{R}^{d_c}$  is the memory cell of step  $i$ , and  $\mathbf{x}_i \in \mathbb{R}^{d_e}$  is the input of step  $i$ . Given  $\text{LSTM}(\cdot)$ , applying the LSTM model is straightforward. We simply repeat the call of  $\text{LSTM}(\cdot)$  for the inputs  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  and obtain the outputs  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ . This resembles the way we use vanilla RNNs, making it very easy to extend LSTM to multi-layer models (see Section 4.2.3) and bi-directional models (see Section 4.2.4).

We can divide  $\text{LSTM}(\cdot)$  into three steps.

- **Step 1: Forget.** Assuming that  $\mathbf{c}_{i-1}$  contains the information that the model memorizes at step  $i-1$ , we need to determine how much information in  $\mathbf{c}_{i-1}$  is discarded in building  $\mathbf{c}_i$ . To do this, a gate is used to control to what extent we forget for each dimension of  $\mathbf{c}_{i-1}$ . The **forget gate** is defined to be:

$$\mathbf{f}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_f + \mathbf{x}_i\mathbf{V}_f + \mathbf{b}_f) \quad (4.20)$$

where  $\mathbf{f}_i \in [0, 1]^{d_c}$  is a vector with the same number of dimensions as  $\mathbf{c}_{i-1}$ . The Sigmoid function maps the input data to the range  $[0, 1]$ . Thus, an entry of  $\mathbf{f}_i$  indicates how much is preserved for the same entry of  $\mathbf{c}_{i-1}$ . Taking this further,  $\mathbf{f}_i \odot \mathbf{c}_{i-1}$  describes the memory that is left out after passing through the forget gate. See Figure 4.5 (a) for an illustration of the forget gate in the LSTM unit.

- **Step 2: Update.** Next we update the memory by considering both the previous state of the memory (i.e.,  $\mathbf{c}_{i-1}$ ) and the input of the LSTM unit (i.e.,  $\mathbf{x}_i$  and  $\mathbf{h}_{i-1}$ ). We first combine  $\mathbf{x}_i$  and  $\mathbf{h}_{i-1}$  using a simple neural network, like this

$$\hat{\mathbf{c}}_i = \text{TanH}(\mathbf{h}_{i-1}\mathbf{U}_c + \mathbf{x}_i\mathbf{V}_c + \mathbf{b}_c) \quad (4.21)$$

$\hat{\mathbf{c}}_i$  can be treated as the new information we intend to add to the memory at step  $i$ . Again, we need a way to control the amount of information coming into the memory. Hence we define an **input gate** as

$$\mathbf{g}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_g + \mathbf{x}_i\mathbf{V}_g + \mathbf{b}_g) \quad (4.22)$$

This equation is similar to Eq. (4.20) but with different parameters. We then define  $\mathbf{g}_i \odot \hat{\mathbf{c}}_i$  to be the actual new information that we are interested in. Taking both  $\mathbf{f}_i \odot \mathbf{c}_{i-1}$  and  $\mathbf{g}_i \odot \hat{\mathbf{c}}_i$ , the memory cell at step  $i$  is given by

$$\mathbf{c}_i = \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{g}_i \odot \hat{\mathbf{c}}_i \quad (4.23)$$

In other words, we forget something old in  $\mathbf{c}_{i-1}$  and memorize something new in  $\hat{\mathbf{c}}_i$ . See Figure 4.5 (b) for an illustration of the update step.

- **Step 3: Output.** In the last step we generate the output  $\mathbf{h}_i$  based on the memory  $\mathbf{c}_i$ . Instead of copying  $\mathbf{c}_i$  to  $\mathbf{h}_i$ , we feed  $\mathbf{c}_i$  to a hyperbolic function and multiply its result with the output gate. Like Eqs. (4.20) and (4.22), the output gate is given by

$$\mathbf{o}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_o + \mathbf{x}_i\mathbf{V}_o + \mathbf{b}_o) \quad (4.24)$$

Then, the output of the LSTM unit is defined to be

$$\mathbf{h}_i = \mathbf{o}_i \odot \text{TanH}(\mathbf{c}_i) \quad (4.25)$$

See Figure 4.5 (c) for an illustration of the output step.

The LSTM model is parameterized by  $\mathbf{U}_f, \mathbf{U}_c, \mathbf{U}_g, \mathbf{U}_o \in \mathbb{R}^{d_h \times d_h}$ ,  $\mathbf{V}_f, \mathbf{V}_c, \mathbf{V}_g, \mathbf{V}_o \in \mathbb{R}^{d_e \times d_h}$ , and  $\mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_g, \mathbf{b}_o \in \mathbb{R}^{d_h}$ . Compared with vanilla RNNs, additional parameters are introduced here because of the use of three gates. In practice one can implement them in many different ways, e.g., using activation functions other than  $\text{Sigmoid}(\cdot)$  and  $\text{TanH}(\cdot)$ , removing the bias terms  $\mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_g$ , and  $\mathbf{b}_o$ , and so on. Training LSTM models follows the standard paradigm of training RNN-based models. For example, we can build an LSTM-based language model and train it by using the methods presented in Section 4.2.2.

### 4.3.3 Gated Recurrent Units

Above, we saw the important role played by the gate units and the memory cell. In general the use of these neural networks makes the model computationally more expensive. An alternative to LSTM in a cheap case, namely **gated recurrent units (GRUs)**, uses a simplified model structure with fewer gate functions [Cho et al., 2014; Chung et al., 2014]. Unlike LSTM, a GRU does not have a memory cell so, as an RNN unit, it takes both the previous state vector  $\mathbf{h}_{i-1}$  and the current input vector  $\mathbf{x}_i$ , and produces the current state vector  $\mathbf{h}_i$ .

In GRUs, there are two gate units: the **reset gate** and the **update gate**. The reset gate, as the name suggests, is used to reset (or rescale) the state of the GRU (i.e.,  $\mathbf{h}_{i-1}$ ). Following the gate functions used in LSTM, the reset gate is defined to be

$$\mathbf{r}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_r + \mathbf{x}_i\mathbf{V}_r + \mathbf{b}_r) \quad (4.26)$$

where  $\mathbf{r}_i \in [0, 1]^{d_h}$  is a vector of scalars, each dimension describing how much information in the corresponding dimension of  $\mathbf{h}_{i-1}$  is retained. Thus, we have a representation of retained

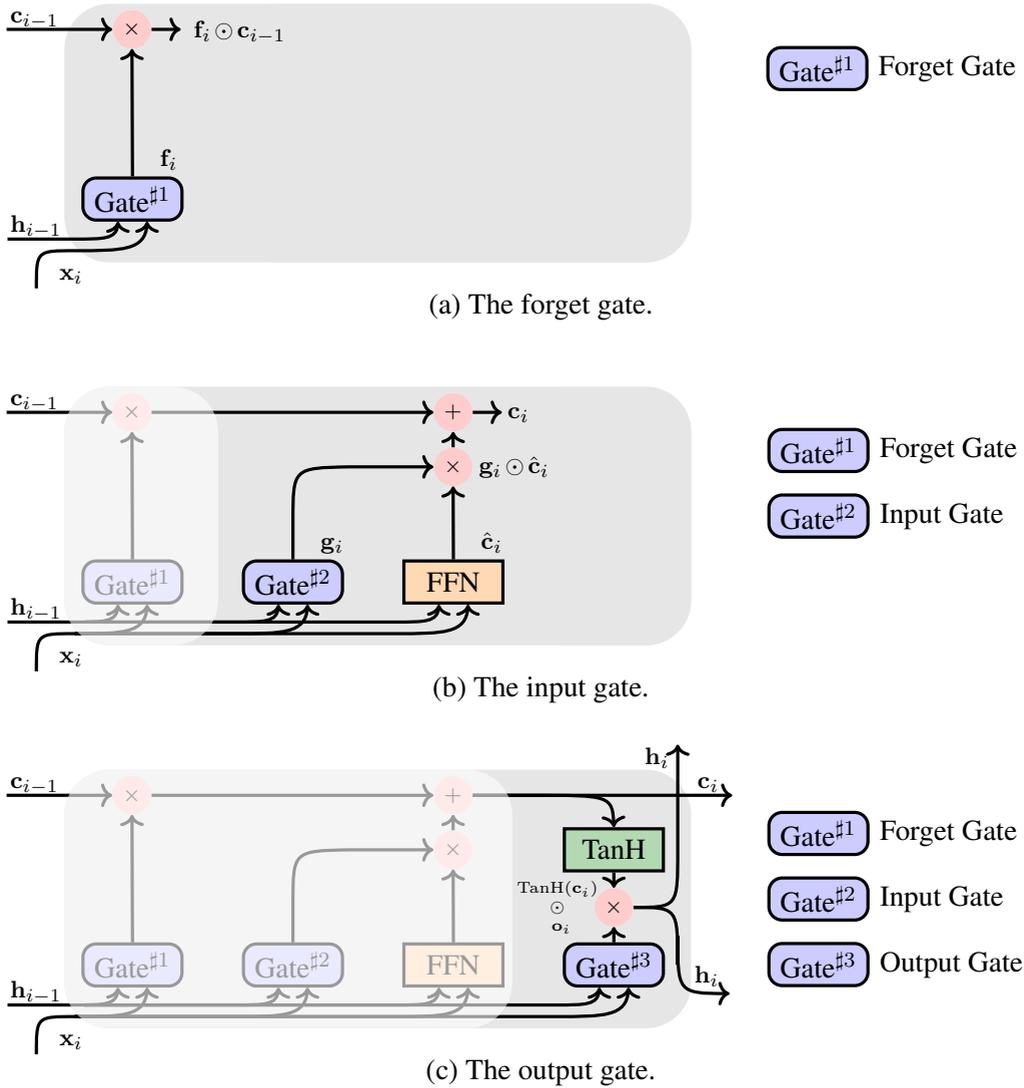


Figure 4.5: The architecture of the LSTM unit. At step  $i$ , it takes the input  $x_i$ , and then updates both the memory cell ( $c_{i-1} \rightarrow c_i$ ) and the recurrent cell ( $h_{i-1} \rightarrow h_i$ ). This process involves three gates: the forget gate controls how much information in  $c_{i-1}$  is retained at step  $i$ , the input gate controls how much information in  $c_{i-1}$  and  $x_i$  is retained at step  $i$ , and the output gate controls how much information in  $c_i$  is used to form  $h_i$ .

information

$$v_{i-1} = r_i \odot h_{i-1} \tag{4.27}$$

Taking both the retained information  $v_{i-1}$  and the current input  $x_i$ , a new state vector is defined

to be

$$\hat{\mathbf{h}}_i = \text{TanH}(v_{i-1}\mathbf{U}_h + \mathbf{x}_i\mathbf{V}_h + \mathbf{b}_h) \quad (4.28)$$

The update gate is then given by

$$\mathbf{u}_i = \text{Sigmoid}(\mathbf{h}_{i-1}\mathbf{U}_u + \mathbf{x}_i\mathbf{V}_u + \mathbf{b}_u) \quad (4.29)$$

$\mathbf{u}_i$  can be thought of as a coefficient vector which could be used to control the trade-off in choosing the new state vector  $\hat{\mathbf{h}}_i$  or the old state vector  $\mathbf{h}_{i-1}$ . Finally, the output of the GRU is defined as a linear interpolation of  $\hat{\mathbf{h}}_i$  and  $\mathbf{h}_{i-1}$

$$\mathbf{h}_i = \mathbf{u}_i \odot \hat{\mathbf{h}}_i + (1 - \mathbf{u}_i) \odot \mathbf{h}_{i-1} \quad (4.30)$$

Figure 4.6 shows how the information flows in a GRU unit. The parameters here are  $\mathbf{U}_r, \mathbf{U}_h, \mathbf{U}_u \in \mathbb{R}^{d_h \times d_h}$ ,  $\mathbf{V}_r, \mathbf{V}_h, \mathbf{V}_u \in \mathbb{R}^{d_e \times d_h}$ , and  $\mathbf{b}_r, \mathbf{b}_h, \mathbf{b}_u \in \mathbb{R}^{d_h}$ . Therefore, the GRU model is smaller than the LSTM model because of the use of fewer gate units. Note that removing the memory cell makes GRUs more efficient. In this case, the role of memory is implicitly played by GRU's output  $\mathbf{h}_i$ , and we maintain it by memorizing more “important” information.

## 4.4 Convolutional Models

In this section we describe another type of model for sequence modeling, called convolutional neural networks (CNNs). Our description is mostly standard, but not a full introduction to the numerous variants of CNNs and cutting-edge techniques. In particular, we focus on using CNNs to deal with sequential data and presenting some refinements.

### 4.4.1 Convolution

CNNs feature their shared-weight architectures by which a kernel or filter slides over the input data and produces a map of features. The idea is that the filter only receives signals from a restricted region of data at a time (call it the **receptive field**), and computes the weighted sum of these input signals. To illustrate this, we follow the convention that a filter in CNNs is generally used to deal with 2D data. Consider a  $3 \times 3$  data matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 9 & 7 \\ 3 & 1 & 2 \\ 0 & 1 & -1 \end{bmatrix} \quad (4.31)$$

and a  $2 \times 2$  filter with a weight matrix

$$\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 2 & 2 \end{bmatrix} \quad (4.32)$$

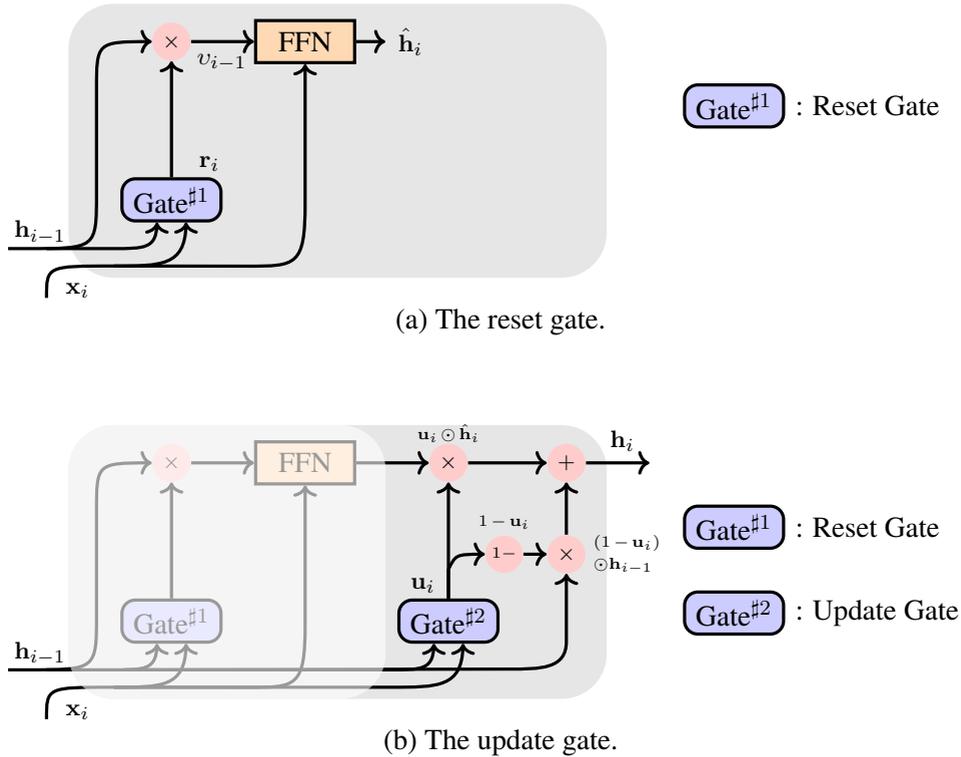


Figure 4.6: The architecture of the GRU. Unlike the LSTM unit, the GRU does not involve a memory cell, and thus follows the same input and output forms of a standard RNN unit. There are two gates in the GRU. The reset gate controls how much information in  $h_{i-1}$  is retained at step  $i$ . The retained information is then taken to fuse with the input  $x_i$ , generating the candidate output  $\hat{h}_i$ . The update gate seeks a balance between  $\hat{h}_i$  and  $h_{i-1}$  in computing the final output of the GRU.

We can apply the filter to every  $2 \times 2$  sub-matrix of  $\mathbf{A}$  (there are four  $2 \times 2$  sub-matrices here), and compute the sum of the  $2 \times 2$  entries weighted by  $\mathbf{W}$ . For example, consider the  $2 \times 2$  sub-matrix in the upper left corner of  $\mathbf{A}$ . The output of the filter is given by

$$\begin{aligned}
 \text{Conv}\left(\begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}, \mathbf{W}\right) &= \text{Conv}\left(\begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 2 & 2 \end{bmatrix}\right) \\
 &= 1 \times 2 + 9 \times 0 + 3 \times 2 + 1 \times 2 \\
 &= 10
 \end{aligned} \tag{4.33}$$

$\text{Conv}(\cdot)$  defines a **convolution operation** that sums the entries of the element-wise product of the two matrices. The convolution operation can be extended to cover the entire input matrix

by sliding the filter over it, as follows

$$\begin{aligned}
 \text{Conv}(\mathbf{A}, \mathbf{W}) &= \text{Conv}\left(\begin{bmatrix} 1 & 9 & 7 \\ 3 & 1 & 2 \\ 0 & 1 & -1 \end{bmatrix}, \mathbf{W}\right) \\
 &= \begin{bmatrix} \text{Conv}\left(\begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}, \mathbf{W}\right) & \text{Conv}\left(\begin{bmatrix} 9 & 7 \\ 1 & 2 \end{bmatrix}, \mathbf{W}\right) \\ \text{Conv}\left(\begin{bmatrix} 3 & 1 \\ 0 & 1 \end{bmatrix}, \mathbf{W}\right) & \text{Conv}\left(\begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}, \mathbf{W}\right) \end{bmatrix} \\
 &= \begin{bmatrix} 10 & 24 \\ 8 & 2 \end{bmatrix} \tag{4.34}
 \end{aligned}$$

The output  $\begin{bmatrix} 10 & 24 \\ 8 & 2 \end{bmatrix}$  is also called the **feature map** for the filter  $\mathbf{W}$  on  $\mathbf{A}$ . Sometimes, the convolution operation  $\text{Conv}(\mathbf{A}, \mathbf{W})$  is written as  $\mathbf{A} * \mathbf{W}$  where the symbol  $*$  stands for the **convolution product**.<sup>9</sup>

Now let us consider a more general description of convolution in CNNs. Suppose that  $\mathbf{A}$  is a multi-dimensional data array. A filter defines a window (or receptive field) on  $\mathbf{A}$ . We can move the window on  $\mathbf{A}$  in different directions. This results in a set of data arrays, denoted by  $\Omega$ . Each data array  $\mathbf{a}_p \in \Omega$  is formed by the elements from the corresponding region of  $\mathbf{A}$ . For example, there are four sub-matrices in Eq. (4.34):  $\mathbf{a}_1 = \begin{bmatrix} 1 & 9 \\ 3 & 1 \end{bmatrix}$ ,  $\mathbf{a}_2 = \begin{bmatrix} 9 & 7 \\ 1 & 2 \end{bmatrix}$ ,  $\mathbf{a}_3 = \begin{bmatrix} 3 & 1 \\ 0 & 1 \end{bmatrix}$ , and  $\mathbf{a}_4 = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}$ . Also, we suppose the filter is parameterized by a weight array  $\mathbf{W}$  with the same size of  $\mathbf{a}$ , i.e.,  $|\mathbf{a}_p| = |\mathbf{W}|$ . The result of applying the filter to  $\mathbf{A}$  is an array of features

$$\text{Conv}(\mathbf{A}, \mathbf{W}) = \begin{bmatrix} v_1 & \dots & v_{|\Omega|} \end{bmatrix} \tag{4.37}$$

<sup>9</sup>In mathematical analysis, given two integrable functions  $f(\cdot)$  and  $g(\cdot)$ , **convolution** defines a new integrable function  $f * g(\cdot)$  to describe the integral of  $f(\cdot)$  weighted by reflected, shifted  $g(\cdot)$ . More formally, the convolution for continuous functions is defined as

$$f * g(x) = \int_{\mathbb{R}} f(y)g(x-y)dy \tag{4.35}$$

where  $f(y)$  is the function that we are concerned with, and  $g(x-y)$  is the weight function which is translated by reflecting  $g(y)$  along the  $y$ -axis and then shifting it by  $x$ . A special case is that  $x$  and  $y$  are both integers. In this case, we can define  $f * g(\cdot)$  as

$$f * g(x) = \sum_y f(y)g(x-y) \tag{4.36}$$

which is the basic form of Eq. (4.33). In CNNs,  $x$ ,  $y$  and  $x-y$  can be seen as indices of items in data arrays.  $f(y)$  is a data item in the input array, and  $g(x-y)$  is the corresponding weight in the filter. By using Eq. (4.36), we calculate the value of the item indexed by  $x$  in the output array  $f * g(x)$  (i.e., the feature map).

Each feature  $v_p$  is given by

$$\begin{aligned}
 v_p &= \text{Conv}(\mathbf{a}_p, \mathbf{W}) \\
 &= \mathbf{a}_p \cdot \mathbf{W} \\
 &= \sum_{k=1}^{|\mathbf{W}|} a_p(k) \cdot W(k)
 \end{aligned} \tag{4.38}$$

where  $a_p(k)$  and  $W(k)$  are the  $k$ -th elements of  $\mathbf{a}_p$  and  $\mathbf{W}$ , respectively. Note that the array  $[v_1 \dots v_{|\Omega|}]$  can be organized into different shapes, such as a matrix or a 3D tensor, though they are essentially the same thing from the data storage viewpoint. For example, for 2D input data and a 2D filter, the feature map is a matrix like Eq. (4.34).

Furthermore, we need to consider two things to make the model practical. First, we need to specify the stride of each move of the filter over  $\mathbf{A}$ . In the above example, we simply use  $\text{stride} = 1$ . By choosing a larger stride, we can compress  $\mathbf{A}$  into a smaller number of features. Second, in some situations, to ensure that the feature map has a desired size, we can add dummy elements (or paddings) around the input data. A common method of padding is to set zeros to the elements outside the input region. For example, consider a  $2 \times 2$  data matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & 9 \\ 7 & 3 \end{bmatrix} \tag{4.39}$$

We can add zero-valued entries around it to obtain a  $4 \times 4$  matrix, like this

$$\mathbf{A}_{\text{padding}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 9 & 0 \\ 0 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.40}$$

Using the same filter as in Eq. (4.33) with  $\text{stride} = 1$ , we have a  $3 \times 3$  feature map

$$\text{Conv}_{\text{stride}=1}(\mathbf{A}_{\text{padding}}, \mathbf{W}) = \begin{bmatrix} 2 & 20 & 18 \\ 14 & 22 & 24 \\ 0 & 14 & 6 \end{bmatrix} \tag{4.41}$$

If  $\text{stride} = 2$ , then we would have a feature map with the same size of the input data

$$\text{Conv}_{\text{stride}=2}(\mathbf{A}_{\text{padding}}, \mathbf{W}) = \begin{bmatrix} 2 & 18 \\ 0 & 6 \end{bmatrix} \tag{4.42}$$

#### 4.4.2 CNNs for Sequence Modeling

Following the formulation in the previous sections, we assume that the input of a sequence model is a vector sequence  $\mathbf{x}_1 \dots \mathbf{x}_m$  and the output is another vector sequence  $\mathbf{h}_1 \dots \mathbf{h}_m$ . For example, we can think of  $\mathbf{x}_1 \dots \mathbf{x}_m$  as a matrix  $\mathbf{X} \in \mathbb{R}^{m \times d_e}$  in which the  $i$ -th row vector is the

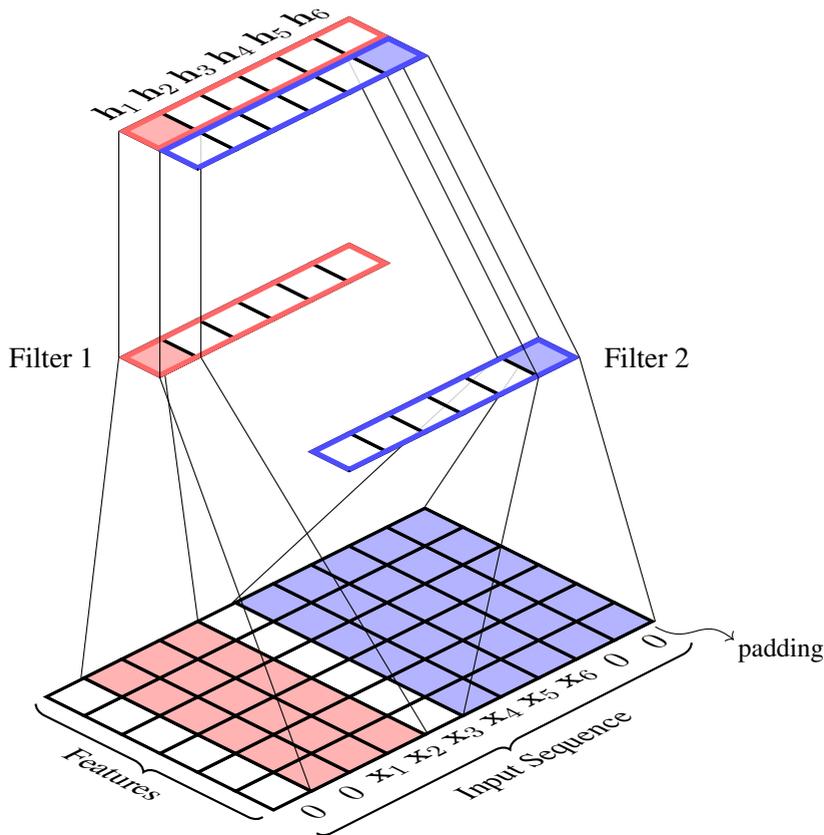


Figure 4.7: Two filters applied to a sequence of word vectors. The input involves ten word vectors (words  $x_1 \dots x_6$  and two padding words on each of the two ends of the sequence). Each word vector has 6 dimensions, and so, the input is a  $10 \times 6$  matrix. Filter 1 has a receptive field of size  $3 \times 6$ . By sliding it over the input matrix, we obtain a sequence of outputs, each corresponding to a position (i.e., a sequence of 6 outputs). Similarly, we apply filter 2 to the input sequence and obtain another sequence of outputs. The two output sequences are then organized as a  $2 \times 6$  matrix in which the  $i$ -th row vector is  $h_i$ .

representation of the  $i$ -th word of the sequence.

It is straightforward to perform convolution on  $\mathbf{X}$ . Since  $x_i$  is just a set of unordered features, it is not necessary to slide a filter over different features. Hence we can use a receptive field of size  $r \times d_e$ , and consider all the dimensions of  $x_i$  in the convolution operation. In practical applications, there might be multiple filters for representing the inputs in different aspects. For example, one can use a filter with a large receptive field to involve more contexts in modeling, and use a filter with a small receptive field to concentrate more on local features. See Figure 4.7 for two filters that are used to deal with a sequence.

To distribute features to  $\{h_1, \dots, h_m\}$ , we can associate each application of a filter to a position of the sequence. To ensure the input and output sequences are of the same length, a padding vector is added to each end of the sequence. The following shows the input and output

of a CNN for an example sequence.

Position	Input	Receptive Field	Output
0	$\mathbf{x}_0 (= 0)$	N/A	N/A
1	$\mathbf{x}_1$	$\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2\}$	$\mathbf{h}_1$
2	$\mathbf{x}_2$	$\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$	$\mathbf{h}_2$
3	$\mathbf{x}_3$	$\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$	$\mathbf{h}_3$
4	$\mathbf{x}_4$	$\{\mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5\}$	$\mathbf{h}_4$
5	$\mathbf{x}_5 (= 0)$	N/A	N/A

An activation function is typically used to introduce some non-linearity to the final output. In this way, we build a standard convolutional layer which can be viewed as a sequence of fully connected neural networks, each taking inputs from a fixed-size window. For the  $i$ -th position, the output of the convolutional layer is given by

$$v_i = \psi(\text{Conv}(\mathbf{a}_i, \mathbf{W})) \quad (4.43)$$

where  $\mathbf{a}_i$  is the inputs in the receptive field<sup>10</sup>, and  $\mathbf{W}$  is the parameters of the filter. In situations involving multiple filters (say  $d_h$  filters), we have a set of parameters  $\{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(d_h)}\}$ , a set of activation functions  $\{\psi^{(1)}, \dots, \psi^{(d_h)}\}$ , and a set of inputs  $\{\mathbf{a}_i^{(1)}, \dots, \mathbf{a}_i^{(d_h)}\}$ . Each tuple  $(\mathbf{W}^{(k)}, \psi^{(k)}, \mathbf{a}_i^{(k)})$  gives an output by

$$v_i^{(k)} = \psi^{(k)}(\text{Conv}(\mathbf{a}_i^{(k)}, \mathbf{W}^{(k)})) \quad (4.44)$$

Note that  $v_i^{(k)}$  is simply an entry of  $\mathbf{h}_i$ . Thus,  $\mathbf{h}_i$  can be written as

$$\mathbf{h}_i = \begin{bmatrix} v_i^{(1)} & \dots & v_i^{(d_h)} \end{bmatrix} \quad (4.45)$$

Many CNN-based systems of practical interest comprise two or more convolutional layers. The simplest way to achieve this is layer stacking, as in multi-layer RNNs (see Section 4.2.3). That is, we treat the output of a convolutional layer as the input of the following layer. See Figure 4.8 for an example of a CNN involving three convolutional layers. One of the benefits of multi-layer CNNs is a larger scope for representation. As seen from Figure 4.8, a neuron in layer 1 connects three input vectors, while a neuron in layer 3 connects, though not directly, seven input vectors. Since neurons of the higher-level layers receive and process signals from a larger span of the sequence, they are expected to produce a higher-level representation of the sequence and to be able to deal with more difficult problems, such as long-distance dependencies.

In some applications, we need a fixed-length, low-dimensional representation of the entire sequence. A common way is to add a pooling layer to merge  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  into a single vector. For example, we can select the maximum value (i.e., max-pooling) or average the values (i.e.,

<sup>10</sup>For an  $r \times d_c$  receptive field,  $\mathbf{a}_i$  is defined to be  $\{\mathbf{x}_{\lceil i-\frac{r}{2} \rceil}, \dots, \mathbf{x}_{\lfloor i+\frac{r}{2} \rfloor - 1}\}$  or  $\{\mathbf{x}_{\lfloor i-\frac{r}{2} \rfloor}, \dots, \mathbf{x}_{\lceil i+\frac{r}{2} \rceil - 1}\}$ .

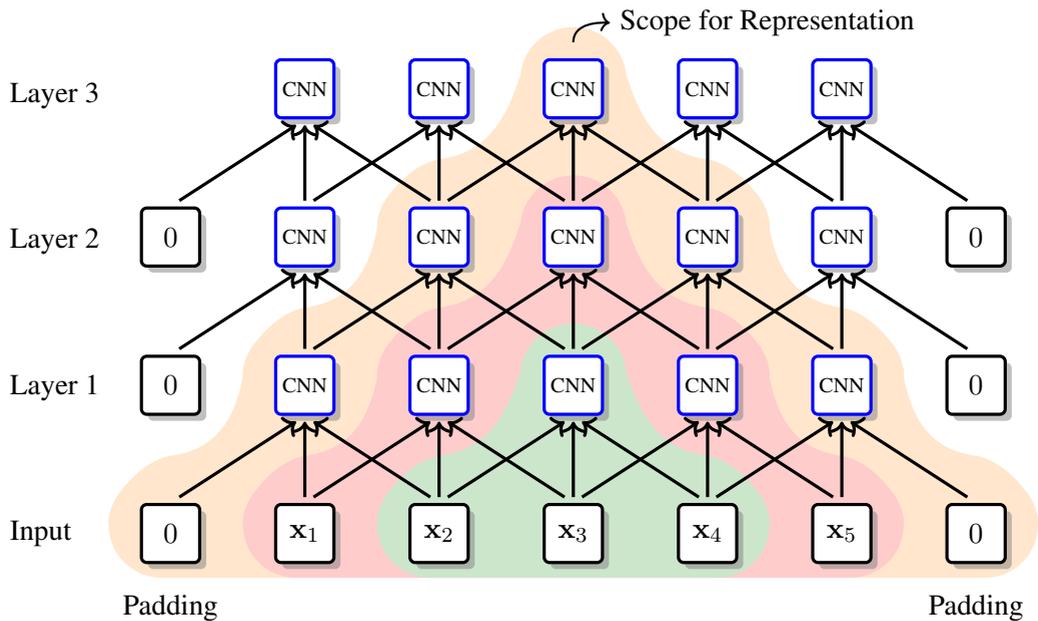


Figure 4.8: A CNN with 3 convolutional layers ( $stride = 1$  and  $r = 3$ ). Each layer takes a sequence of vectors and produces another sequence of vectors. In this process a filter moves over the input and performs the convolution operation in each move. In layer 1, the receptive field of the filter is a region of three input items (see green shadows). The higher a layer is, the larger receptive field a filter has. For example, in layer 3, an application of the filter can at most cover the entire input sequence (see orange shadows).

max-pooling) along each dimension. This is a generic method in machine learning and is applicable to most of the sequence models discussed in this book.

### 4.4.3 Handling Positional Information

One interesting property of CNNs is their ability to balance complexity and efficiency. This is achieved by restricting full connectivity to only a small region of the input data. This also leads us to describe a convolution layer using the same mathematical form of a layer in a fully-connected neural network: the output of a neuron is some transformation of the weighted sum of the input numbers. Despite the simplicity inherent in modeling, a problem with such models is that the order of inputs is completely ignored. An interesting point, however, is that, if we restrict ourselves to sequence modeling, this should not be a problem because the output of the model is itself a sequence. It seems reasonable to assume that the output sequence preserves the ordering information of the input sequence. On the other hand, applying CNNs to sequential data does not guarantee a one-to-one mapping between the input and output items. Technically,  $h_i$  is not simply a representation of  $x_i$ . It instead encodes a window of inputs centered at  $x_i$ . This, in turn, makes the problem very complicated, since it is difficult to work out from  $h_i$  how those inputs are ordered.

Explicitly modeling word orders is very important in NLP, and has been extensively studied in tasks like machine translation [Lopez, 2008; Koehn, 2010]. For neural network-based models, one may address the problem by resorting to order-sensitive model architectures like RNNs. A more popular approach in recent systems is to develop a **positional encoding** sub-model and incorporate it into existing sequence models [Gehring et al., 2017b; Vaswani et al., 2017; Shaw et al., 2018; Dufter et al., 2022]. Formally, we say that the input at position  $i$  is a combination of the original input  $\mathbf{x}_i$  and the positional encoding of  $i$  (denoted by  $\text{PE}(\cdot)$ ):

$$\mathbf{x}\mathbf{p}_i = \text{Merge}(\mathbf{x}_i, \text{PE}(i)) \quad (4.46)$$

where  $\text{Merge}(\cdot)$  combines  $\mathbf{x}_i$  and  $\text{PE}(i)$  in some way. The use of positional encoding is straightforward: all you need is to replace  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  with  $\{\mathbf{x}\mathbf{p}_1, \dots, \mathbf{x}\mathbf{p}_m\}$  in a sequence model. So, this approach is model-free.

In this subsection, we present several versions of  $\text{PE}(\cdot)$  and ways to combine them with  $\mathbf{x}_i$ . Note that the following discussion is not specific to CNNs. We consider it here because positional encoding is useful for models that are insensitive to the order of inputs, and CNNs are a good example to see how it is used [Gehring et al., 2017a;b]. In Chapter 6, we will see an application of positional encoding in Transformer which is a state-of-the-art neural model in many areas.

## 1. Offset-based Positional Encoding

The simplest way to describe a position  $i$  is to just leave it as it is. This can be formulated as the “distance” from a reference point

$$\text{PE}(i) = i - i_0 \quad (4.47)$$

where  $i_0$  is an integer indicating where we start counting. If  $i_0 = 0$ ,  $\text{PE}(i) = i$  gives the normal way to define a position. Note that  $\text{PE}(i)$  could be a negative number if  $i_0 > i$ . In this sense,  $\text{PE}(i)$  is not a real distance but it is fine with considering it as a feature in a machine learning system. To design a non-negative measure, the right-hand side of Eq. (4.47) can be defined as an absolute value

$$\text{PE}(i) = |i - i_0| \quad (4.48)$$

Treating positions as simple integers leads to unbounded, discrete positional encoding. A more desirable method might be to use a continuous representation in a range of values, because it allows the system to work within a sample space that is smooth and easy to optimize. A simple way to do this is normalization. For example, dividing  $i - i_0$  by some maximum value, we obtain a normalized version of the offset-based encoding

$$\text{PE}(i) = \frac{i - i_0}{i_{\max} - i_0} \quad (4.49)$$

For example, we can set  $i_{\max} =$  the maximum possible length of the sequence and  $i_0 = 0$ ,

so that  $\text{PE}(i)$  chooses its value in  $[0, 1]$ . Another common choice is to set  $i_{\max} = m$  (i.e., the length of the input sequence) and define  $\text{PE}(i)$  as a ratio whose value varies as  $m$  changes.

To make use of these scalar positions, it is straightforward to enrich the original input vectors by adding new dimensions, provided they can be viewed as new features. Thus,  $\mathbf{x}\mathbf{p}_i$  is given by

$$\mathbf{x}\mathbf{p}_i = [\mathbf{x}_i, \text{PE}(i)] \quad (4.50)$$

where  $[\cdot]$  stands for the concatenation operation.

## 2. Sinusoidal Positional Encoding

The next obvious step is to represent positions as vectors instead of scalars. Although vectorizing the representations of positions sounds complicated, a simple idea is to use a carrying system which describes how a natural number is expressed by a polynomial with respect to a base [Kernes, 2021]. For example,  $i$  can be written as

$$i = \sum_{k=0}^{k_{\max}} a(i, k)b^k \quad (4.51)$$

where  $a(i, k)$  is the  $k$ -th digit,  $k_{\max} + 1$  is the maximum number of digits, and  $b$  is the base of the system. The carrying occurs when  $a(i, k)$  reaches  $b$ : we increase  $a(i, k + 1)$  by 1 and roll back  $a(i, k)$  to 0. In this way we can change  $a(i, k)$  with a period of  $b^k$ , that is,  $a(i, 0)$  changes with a period of  $b^0$ ,  $a(i, 1)$  changes with a period of  $b^1$ ,  $a(i, 2)$  changes with a period of  $b^2$ , and so on.

Using this system,  $i$  can be represented as a vector

$$\text{PE}(i) = [a(i, 0) \quad a(i, 1) \quad \dots \quad a(i, k_{\max})] \quad (4.52)$$

For example, when  $b = 2$ ,  $\text{PE}(11) = [1 \quad 1 \quad 0 \quad 1]$ . However, in Eq. (4.52),  $\text{PE}(i)$  is still a discrete function. As discussed throughout this book, we may want a continuous vector representation that can describe intermediate states between discrete events. Considering  $a(i, k)$  as a periodic function, a common choice is the **sine** function. Thus  $a(i, k)$  can be re-defined, as follows

$$a(i, k) = \sin(i \cdot \omega_k) \quad (4.53)$$

This function has an amplitude of 1 and a period of  $\frac{2\pi}{\omega_k}$ . Using an analogous form of periods to that used in Eq. (4.51), we define  $\omega_k$  as

$$\omega_k = \frac{1}{(b_{\text{model}})^{k/d_{\text{model}}}} \quad (4.54)$$

where  $b_{\text{model}} > 0$  and  $d_{\text{model}} > 0$  are hyper-parameters of the model. Obviously, we have  $\frac{2\pi}{\omega_0} < \frac{2\pi}{\omega_1} < \dots < \frac{2\pi}{\omega_{k_{\max}}}$ .

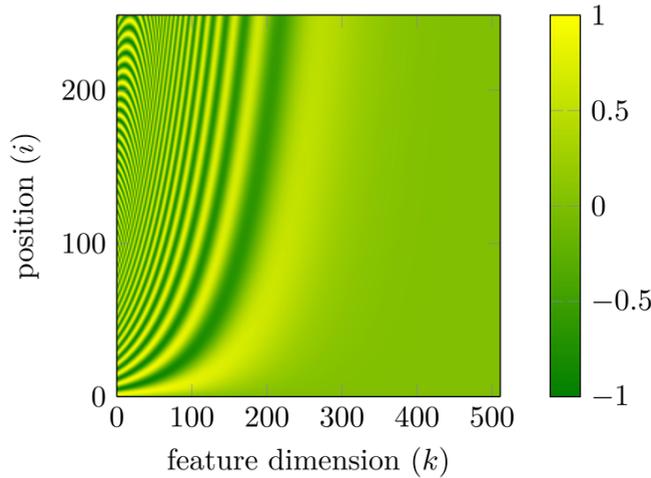


Figure 4.9: A heat map of the positional embedding model of Eqs. (6.14) and (6.15) ( $b_{\text{model}} = 10,000$  and  $d_{\text{model}} = 512$ ). Consider a position  $i$  (i.e., the  $i$ -th row), then move another position  $j$  from  $i$  upwards or downwards. Intuitively, when  $i$  and  $j$  are closer, the corresponding row vectors are more similar. By contrast, when  $j$  moves away from  $i$ , the similarity is not that obvious. This property helps explain the idea behind the positional embedding model: the “distance” between two positions is implicitly modeled by comparing their multi-dimensional representations.

Similarly, we can define  $a(i, k)$  via the **cosine** function

$$a(i, k) = \cos(i \cdot \omega_k) \quad (4.55)$$

Taking both Eqs. (4.53) and (4.55), we create a new representation of  $i$ , as follows

$$\text{PE}(i) = \left[ \sin(i \cdot \omega_0) \quad \cos(i \cdot \omega_0) \quad \dots \quad \sin(i \cdot \omega_{k_{\text{max}}}) \quad \cos(i \cdot \omega_{k_{\text{max}}}) \right] \quad (4.56)$$

Vaswani et al. [2017] instantiated the above form by setting  $b_{\text{model}} = 10,000$ . Let  $\text{PE}(i, k)$  be the  $k$ -th dimension of  $\text{PE}(i)$ . Vaswani et al. [2017]’s version of positional encoding is written as

$$\text{PE}(i, 2k) = \sin\left(i \cdot \frac{1}{10000^{2k/d_{\text{model}}}}\right) \quad (4.57)$$

$$\text{PE}(i, 2k+1) = \cos\left(i \cdot \frac{1}{10000^{2k/d_{\text{model}}}}\right) \quad (4.58)$$

Choosing  $b_{\text{model}} = 10,000$  is empirical. One can adjust it for specific tasks. Figure 4.9 plots the positional encoding for different positions. We see that, when  $k$  becomes larger, the change of the color follows a larger period.

Note that Eqs. (6.14) and (6.15) have a useful property that  $\text{PE}(i + \mu)$  can be easily

expressed by a linear function of  $\text{PE}(i)$  for a given offset  $\mu$ <sup>11</sup>

$$\begin{aligned} \text{PE}(i + \mu, 2k) &= \text{PE}(i, 2k) \cdot \text{PE}(\mu, 2k + 1) + \\ &\quad \text{PE}(i, 2k + 1) \cdot \text{PE}(\mu, 2k) \end{aligned} \quad (4.61)$$

$$\begin{aligned} \text{PE}(i + \mu, 2k + 1) &= \text{PE}(i, 2k + 1) \cdot \text{PE}(\mu, 2k + 1) + \\ &\quad \text{PE}(i, 2k) \cdot \text{PE}(\mu, 2k) \end{aligned} \quad (4.62)$$

The resulting benefit is that the encoding can somewhat model relative positions. That is, the state at position  $i + \mu$  can be described by starting with  $i$  and then appending it with the offset  $\mu$ .

When applying the sinusoidal positional encoding, one way is to use Eq. (4.50) to concatenate  $\mathbf{x}_i$  and  $\text{PE}(i)$ . In Vaswani et al. [2017]’s work, they instead assume  $\text{PE}(i)$  to be a vector of the same size as  $\mathbf{x}_i$  (i.e.,  $|\text{PE}(i)| = |\mathbf{x}_i| = d_e$ ), and add  $\text{PE}(i)$  to  $\mathbf{x}_i$ , like this

$$\mathbf{x}\mathbf{p}_i = \mathbf{x}_i + \text{PE}(i) \quad (4.63)$$

This sinusoidal additive model has been the basis of many positional encoding approaches [Dehghani et al., 2018; Likhomanenko et al., 2021; Su et al., 2021].

### 3. Learnable Positional Encoding

The result of sinusoidal positional encoding is a lookup table  $\mathbf{C}_{\text{PE}} \in \mathbb{R}^{m_{\max} \times d_e}$  (where  $m_{\max}$  is the maximum sequence length we can choose)

$$\mathbf{C}_{\text{PE}} = \begin{bmatrix} \text{PE}(1) \\ \dots \\ \text{PE}(m_{\max}) \end{bmatrix} \quad (4.64)$$

In this table, each row vector  $\text{PE}(i)$  corresponds to the embedding of a position  $i$ . These vectors, as described above, are computed based on some assumptions and heuristic algorithms. An alternative approach is to treat vectors of positions as parameters of the model and learn them as usual. In this case, both word embeddings and position embeddings can be trained in the same manner. See Chapters 2 and 3 for more information about learning word embeddings in neural language models.

One last note on positional encoding. What we have shown in this section can broadly be characterized as an **absolute positional encoding** paradigm: a position is described by its location in a coordinate system. Another concept that is worth exploring is **relative positional encoding** [Shaw et al., 2018]. For example, we can extend Eq. (4.48) to define the distance

<sup>11</sup>One can derive these by taking

$$\sin(\alpha + \beta) = \sin(\alpha) \cdot \cos(\beta) + \cos(\alpha) \cdot \sin(\beta) \quad (4.59)$$

$$\cos(\alpha + \beta) = \cos(\alpha) \cdot \cos(\beta) - \sin(\alpha) \cdot \sin(\beta) \quad (4.60)$$

between two positions  $i$  and  $j$

$$\text{PE}(i, j) = |i - j| \quad (4.65)$$

In this case, the positional encoding is no longer an attribute of the  $i$ -th input but some representation of the distance relative to a reference position  $j$ . In fact, most of the methods for relative positional encoding are variants on a theme in which positions are described by their pair-wise relationships. This forms the basis of several models of this type as we will see in Chapter 6.

## 4.5 Examples

Both recurrent and convolutional models have been successfully used in numerous applications. Here we discuss a few of the interesting examples. While these models are mostly basic, they form the foundations of many state-of-the-art systems.

### 4.5.1 Text Classification

To illustrate how sequence models could be used, we first consider the text classification problem in which we assign one of some pre-defined classes to a text. It can be extended to cover a broad range of problems in NLP, including classifying news texts, flagging sentiment sentences, identifying spam emails, detecting fake comments, and so on.

In text classification we are interested in selecting the best class from a set  $C$ , given a word sequence  $w_1 \dots w_m$ :

$$\hat{c} = \underset{c \in C}{\text{arg max}} \text{Score}(c, w_1 \dots w_m) \quad (4.66)$$

Here  $\text{Score}(c, w_1 \dots w_m)$  measures how well a class  $c$  is predicted for the input sequence  $w_1 \dots w_m$ . Here we map the sequence of words to the sequence of word vectors (or word embeddings), that is,  $w_1 \dots w_m \rightarrow \mathbf{x}_1 \dots \mathbf{x}_m$ . Assuming  $\text{Score}(\cdot)$  is a probabilistic function that describes the distribution of the classes, we can reformulate the problem as

$$\begin{aligned} \hat{c} &= \underset{c \in C}{\text{arg max}} \text{Pr}(c | w_1 \dots w_m) \\ &= \underset{c \in C}{\text{arg max}} \text{Pr}(c | \mathbf{x}_1 \dots \mathbf{x}_m) \end{aligned} \quad (4.67)$$

The central issue here is the modeling of  $\text{Pr}(c | \mathbf{x}_1 \dots \mathbf{x}_m)$ . We define  $\text{Pr}(c | \mathbf{x}_1 \dots \mathbf{x}_m)$  by following the general encoder + predictor framework, as follows

- The input  $\mathbf{x}_1 \dots \mathbf{x}_m$  is represented as a feature vector  $\mathbf{H} \in \mathbb{R}^{d_h}$  by using an encoder

$$\mathbf{H} = \text{Encoder}(\mathbf{x}_1 \dots \mathbf{x}_m) \quad (4.68)$$

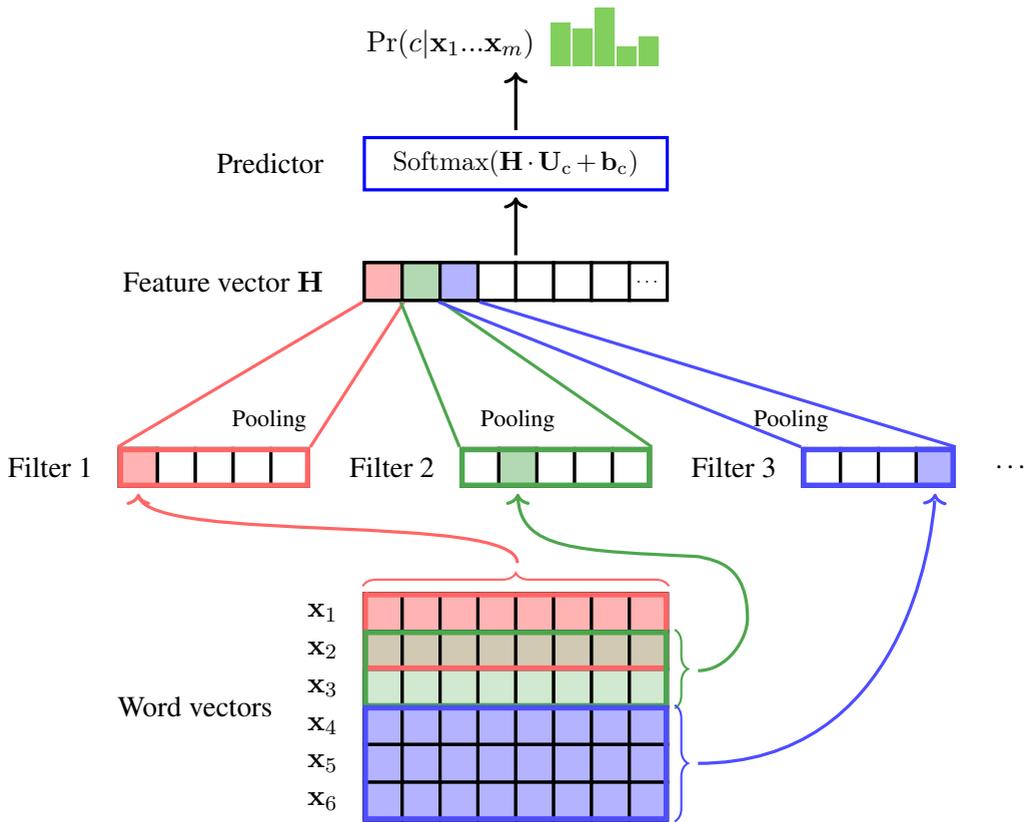


Figure 4.10: A CNN-based text classifier [Kim, 2014]. The input is a sequence of word vectors. A convolutional layer involving multiple filters is used to extract features in different dimensions. A pooling layer is used to reduce the number of features for representing the input text, leading to a low-dimensional feature vector  $\mathbf{H}$ . The prediction conditions on  $\mathbf{H}$  and is made by using a standard Softmax layer.

- $\mathbf{H}$  is fed to a standard Softmax layer to predict the class distribution

$$\Pr(\cdot|\mathbf{H}) = \text{Softmax}(\mathbf{H} \cdot \mathbf{U}_c + \mathbf{b}_c) \quad (4.69)$$

where  $\mathbf{U}_c \in \mathbb{R}^{d_h \times |C|}$  and  $\mathbf{b}_c \in \mathbb{R}^{|C|}$  are model parameters.

Encoder( $\cdot$ ) is exactly the same thing we discussed in the preceding sections. There are, therefore, many encoding models that are applicable here. For example, consider the CNN-based encoder presented in Kim [2014]’s work. Kim [2014]’s model is based on a single convolution layer involving  $d_h$  filters. The application of a filter produces a set of features, each being associated with a position of the sequence. Since we want a single vector for representing the entire sequence, a pooling layer is added so that the number of features for each filter is reduced to one. Then, for any  $c$ , the probability  $\Pr(c|\mathbf{H})$  can be computed trivially according to Eq. (4.69). See Figure 4.10 for an illustration of this classifier.

To train such a model, we just need to optimize it on some loss, and, as mentioned several

times in this book, one of the most common methods is to minimize the cross-entropy loss using gradient descent. Also, we can use regularization to improve the training of CNNs. More details about these techniques can be found in Chapter 2.

Note that while the model described here is quite “simple”, it is among the most effective models known for text classification. There are, of course, improvements to this kind of classifier. Examples of such systems include deep CNNs [Conneau et al., 2017c], character-based CNNs [Santos and Gatti, 2014; Zhang et al., 2015], recurrent CNNs [Lai et al., 2015], and so on.

## 4.5.2 End-to-End Speech Recognition

Speech recognition is a task of taking a sequence of acoustic signals and mapping it to a sequence of words or characters (call it a **transcription**) [Reddy, 1976; Rabiner and Juang, 1993]. Since the original input is an acoustic waveform over the time domain, it is common to transform it into a sequence of waveform fragments (call them **frames**). Typically, a frame is represented as a feature vector, denoted by  $\mathbf{x}_i$ . This is achieved by using either feature functions in signal processing [Davis and Mermelstein, 1980; Picone, 1993; Campbell, 1997] or learnable embeddings [Chorowski et al., 2019; Schneider et al., 2019]. Regarding the output, speech recognition systems generally do not produce words. Instead, they produce sequences of **transcription units** (or **transcription labels**), e.g., phonemes, characters, sub-words, etc. In this subsection we assume that the output of a speech recognition system is a sequence of English letters, denoted by  $y_1 \dots y_n \in V_y^n$ . The alphabet  $V_y$  consists of normal English letters (a – z), numbers (0 – 9), spaces ( $\langle \text{sp} \rangle$ ), periods ( $\langle \text{pe} \rangle$ ), and other punctuation marks. As with most modern speech recognition systems, we add a blank symbol  $\epsilon$  to the alphabet in order to indicate the null output.

The goal here is to find a string  $\hat{y}_1 \dots \hat{y}_n$  for a given input sequence  $\mathbf{x}_1 \dots \mathbf{x}_m$ , so that

$$\hat{y}_1 \dots \hat{y}_n = \arg \max_{y_1 \dots y_n} \Pr(y_1 \dots y_n | \mathbf{x}_1 \dots \mathbf{x}_m) \quad (4.70)$$

This model is relatively difficult compared to the classification model described in the previous subsection, as the output can be an arbitrary string, rather than a class in a predefined class set. The string generation problem leads to two difficulties. First, we need some mechanism to model  $\Pr(y_1 \dots y_n | \mathbf{x}_1 \dots \mathbf{x}_m)$  for an exponentially large number of pairs of input and output sequences. Second, in practice  $y_1 \dots y_n$  is often much shorter than  $\mathbf{x}_1 \dots \mathbf{x}_m$  (i.e.,  $n < m$ ), and so we need some mechanism to align a long sequence to a short one. However, we do not need to consider these difficulties in the stage of representing the input sequence, and can still encode the input sequence  $\mathbf{x}_1 \dots \mathbf{x}_m$  in the same way as other sequence models. Specifically, we represent  $\mathbf{x}_1 \dots \mathbf{x}_m$  in the following form

$$\mathbf{h}_1 \dots \mathbf{h}_m = \text{Encoder}(\mathbf{x}_1 \dots \mathbf{x}_m) \quad (4.71)$$

The encoder can be RNNs, CNNs, or more advanced models (such as Transformer). Here we consider the encoder architecture used in Graves et al. [2013b] and Graves and Jaitly

[2014]’s work. It is a multi-layer bi-directional LSTM model. We skip the details of this model without loss of continuity, as the reader may be already familiar with it in Sections 4.2.3, 4.2.4 and 4.3.2.

Having obtained the sequence representation  $\mathbf{H} = \mathbf{h}_1 \dots \mathbf{h}_m$ , a softmax layer is used to map each  $\mathbf{h}_i \in \mathbb{R}^{d_h}$  to a distribution of transcription labels, given by

$$\Pr(\cdot | \mathbf{h}_i) = \text{Softmax}(\mathbf{h}_i \cdot \mathbf{U}_s + \mathbf{b}_s)$$

where  $\mathbf{U}_s \in \mathbb{R}^{d_h \times |V_y|}$  and  $\mathbf{b}_s \in \mathbb{R}^{|V_y|}$  are model parameters.  $\Pr(\cdot | \mathbf{h}_i) \in \mathbb{R}^{|V_y|}$  is a probability distribution over  $V_y$ , and the probability of transcription label  $l_i$  at position  $i$  is simply  $\Pr(l_i | \mathbf{h}_i)$ . We can then write the probability of a label sequence in the form

$$\Pr(l_1 \dots l_m | \mathbf{H}) = \prod_{k=1}^m \Pr(l_k | \mathbf{h}_k) \quad (4.72)$$

This formulation looks simple. We can appeal to the  $\arg \max$  operation to find the most probable label sequence as usual. However,  $l_1 \dots l_m$  cannot be straightforwardly used as a system output, because it often contains many duplicate and blank symbols. To “post-process”  $l_1 \dots l_m$ , we first merge the sub-sequence of labels to a single label when they are the same, and then remove the blank symbols. For example, consider a label sequence

s s  $\epsilon$  e e  $\epsilon$   $\epsilon$  e  $\langle \text{sp} \rangle$  y  $\epsilon$  o o u

By merging “s s”, “e e”, and “o o”, we have

s  $\epsilon$  e  $\epsilon$   $\epsilon$  e  $\langle \text{sp} \rangle$  y  $\epsilon$  o u

Then, we remove all  $\epsilon$  and obtain

s e e  $\langle \text{sp} \rangle$  y o u

The above sequence is what we would call a transcription. Obviously, different label sequences can correspond to the same transcription. Let  $B(y_1 \dots y_n)$  be the set of label sequences corresponding to  $y_1 \dots y_n$ <sup>12</sup>. We now turn to the following form of the transcription probability (see Figure 4.11 for an illustration)

$$\Pr(y_1 \dots y_n | \mathbf{x}_1 \dots \mathbf{x}_m) = \sum_{l_1 \dots l_m \in B(y_1 \dots y_n)} \Pr(l_1 \dots l_m | \mathbf{H}) \quad (4.73)$$

A problem with this model is that the number of the sequences in  $B(y_1 \dots y_n)$  grows exponentially with  $n$  (and  $m$ ). Fortunately, there are very efficient methods for computing  $\sum_{l_1 \dots l_m \in B(y_1 \dots y_n)} \Pr(l_1 \dots l_m | \mathbf{H})$ . See [Graves et al., 2006] for a dynamic programming

<sup>12</sup> $B(y_1 \dots y_n)$  may contain label sequences of arbitrary lengths. However, if we restrict input to the sequence  $\mathbf{x}_1 \dots \mathbf{x}_m$ , then each sequence in  $B(y_1 \dots y_n)$  is of length  $m$ .

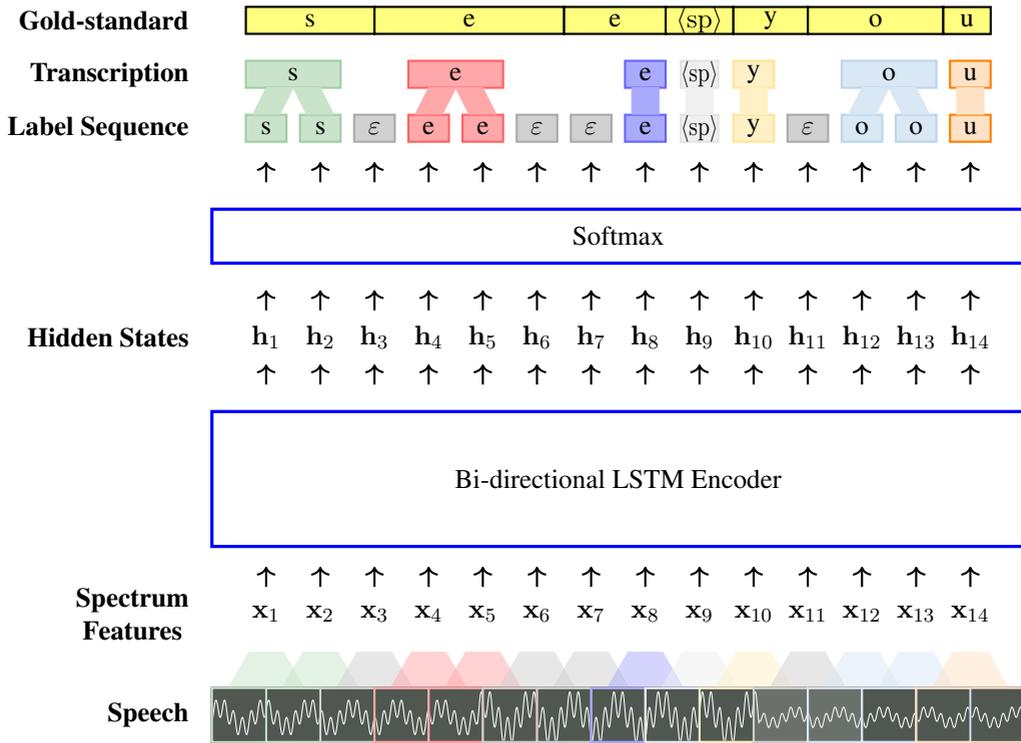


Figure 4.11: An end-to-end speech recognition architecture. The input of the system is a sequence of acoustic signals that are represented as a sequence of feature vectors (i.e.,  $x_1 \dots x_{14}$ ). These feature vectors are taken by a bi-directional LSTM encoder. The output of the encoder is a sequence of contextualized representations (i.e.,  $h_1 \dots h_{14}$ ) which is then fed into a softmax layer for generating a distribution of labels at each position. We can then draw a sequence of labels from these distributions. We map each label sequence to a form of final output by eliminating duplicate symbols and blank symbols. An output of the system corresponds to a number of label sequences, and the probability of the output is the sum of the probabilities of these label sequences.

algorithm for solving this problem.

Note that Eq. (4.73) is also known as a form of **connectionist temporal classification (CTC)** [Graves et al., 2006]. It is one of the most widely used methods for training end-to-end speech recognition and speech translation systems. One of the merits of CTC is that it allows us to align any label sequence to a transcription in a very simple and efficient way. It is easy to make use of CTC in training a speech recognition system. Suppose there is a set of pairs of input sequence and transcription, denoted by  $S$ . A common training objective is to maximize the likelihood of these transcriptions given the corresponding inputs, written as

$$\hat{\theta} = \operatorname{argmax}_{\theta} \sum_{(y_1 \dots y_n, x_1 \dots x_m) \in S} \log \Pr(y_1 \dots y_n | x_1 \dots x_m; \theta) \quad (4.74)$$

where  $\Pr(y_1 \dots y_n | x_1 \dots x_m; \theta)$  is the probability computed via Eq. (4.73), and  $\theta$  is the parameters

of the model.

When testing on new data, we search for an optimal transcription as in Eq. (4.70). This process, also known as **decoding**, generally involves optimized search algorithms and pruning techniques. For example, we can use the 1-best label sequence instead of all possible label sequences to obtain an approximation to Eq. (4.73), that is,  $\Pr(y_1 \dots y_m | \mathbf{x}_1 \dots \mathbf{x}_m) = \max \Pr(l_1 \dots l_m | \mathbf{H})$ . This leads to an efficient decoding method, called **Viterbi decoding**, which has been extensively used in speech recognition and machine translation [Lopez, 2008]. For more details about the decoding of sequence generation, we refer the reader to Chapter 5.

### 4.5.3 Sequence Labeling with LSTM and Graphical Models

Sequence labeling is a conceptually straightforward approach to classifying data in sequence. In NLP, it has penetrated many sub-areas like word segmentation, part-of-speech tagging, and chunking. Learning in these models consists of simply predicting a label in a label set  $V_y$  at each position of a sequence. Ideally, we wish to perform a sequence of labeling actions based on the entire input, given by

$$\hat{y}_1 \dots \hat{y}_m = \underset{y_1 \dots y_m}{\operatorname{argmax}} \Pr(y_1 \dots y_m | \mathbf{x}_1 \dots \mathbf{x}_m) \quad (4.75)$$

where  $\mathbf{x}_1 \dots \mathbf{x}_m$  is an input sequence (such as a sequence of word vectors), and  $y_1 \dots y_m$  is a label sequence in which each label  $y_i$  corresponds to an input item  $\mathbf{x}_i$ .

As we have seen in this chapter, Eq. (4.75) perfectly fits the form of the sequence modeling problem. As a first step we use an encoder to map  $\mathbf{x}_1 \dots \mathbf{x}_m$  to a sequence of contextualized representations, as follows

$$\mathbf{h}_1 \dots \mathbf{h}_m = \operatorname{Encoder}(\mathbf{x}_1 \dots \mathbf{x}_m) \quad (4.76)$$

We define  $\operatorname{Encoder}(\cdot)$  as a bi-directional LSTM model because it involves a memory mechanism for modeling long-range dependencies in both left and right contexts. Hence, the architecture of the encoder is the same as that used in the preceding subsection.

$\mathbf{h}_1 \dots \mathbf{h}_m$  can then be taken to be the input of a usual sequence labeling system (see Figure 4.12). The sequence labeling problem has been discussed in Chapter 1, and many models are applicable to it. The simplest is the one that involves a classifier (such as maximum entropy and SVM-based models) for predicting a label distribution for each  $\mathbf{h}_i$ . A problem with these models is that the predictions are made independently. A more powerful approach is to use **graphical models** to consider dependencies among predicted labels. For example, **hidden Markov models (HMMs)** describe how a sequence of observations (i.e.,  $\mathbf{x}_1 \dots \mathbf{x}_m$ ) is generated given a sequence of variables (i.e.,  $y_1 \dots y_m$ ). The key idea is to rewrite  $\Pr(y_1 \dots y_m | \mathbf{x}_1 \dots \mathbf{x}_m)$  using the Bayes' rule and approximate  $\Pr(y_1 \dots y_m | \mathbf{x}_1 \dots \mathbf{x}_m)$  by a product of simple factors. However, these models require probability density functions of continuous variables (e.g.,  $\Pr(\mathbf{x}_i | y_i)$ ) which are difficult to estimate. This differentiates the use of HMMs in neural models greatly from that in conventional models where all states and observations are discrete

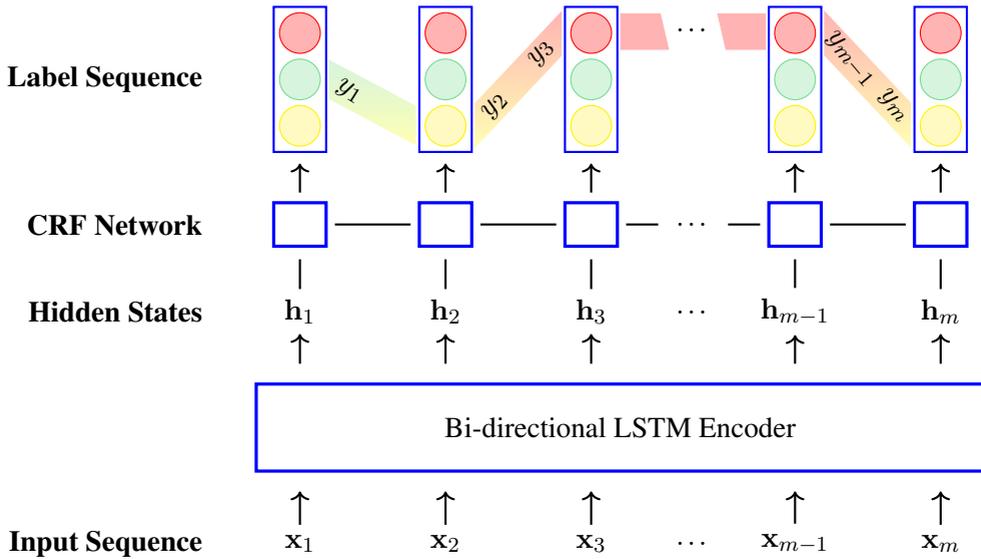


Figure 4.12: The BiLSTM + graphical model architecture for sequence labeling. The encoder is a standard bi-directional LSTM model. Given a sequence of input feature vectors (i.e.,  $x_1 \dots x_m$ ), it produces a new sequence of feature vectors for mapping the input to contextualized representations (i.e.,  $h_1 \dots h_m$ ). A CRF network is placed on  $h_1 \dots h_m$  to predict a distribution of label sequences. The optimal label sequence is the one that has the maximum probability.

variables<sup>13</sup>.

HMMs and their descendants can be viewed as instances of generative models. Another type of model that has been commonly used to solve sequence labeling problems is discriminative models. One such model is **conditional random fields (CRFs)** [Lafferty et al., 2001]. The CRF model features its ability to directly model the joint probability of the entire input and

<sup>13</sup>In HMMs, a sequence of variables can be viewed as a path of transiting over some states whose values are chosen from a pre-defined set. In each transition from one state to another, something is observed (call it an observation). By making some assumptions, we can approximate  $\Pr(y_1 \dots y_m | x_1 \dots x_m)$  in the following fashion

$$\begin{aligned}
 \Pr(y_1 \dots y_m | x_1 \dots x_m) &= \frac{\Pr(y_1 \dots y_m) \cdot \Pr(x_1 \dots x_m | y_1 \dots y_m)}{\Pr(x_1 \dots x_m)} \\
 &\approx \frac{\prod_{i=1}^m \Pr(y_i | y_{i-1}) \cdot \prod_{i=1}^m \Pr(x_i | y_i)}{\Pr(x_1 \dots x_m)} \\
 &= \frac{\prod_{i=1}^m \Pr(y_i | y_{i-1}) \Pr(x_i | y_i)}{\Pr(x_1 \dots x_m)} \quad (4.77)
 \end{aligned}$$

where  $\Pr(y_i | y_{i-1})$  is the **transition probability** of moving from  $y_{i-1}$  to  $y_i$  (when  $i = 1$ , we define  $\Pr(y_i | y_{i-1}) = \Pr(y_1 | y_0) = \Pr(y_1)$ ), and  $\Pr(x_i | y_i)$  is the **emission probability** of observing  $x_i$  given  $y_i$ . As the denominator  $\Pr(x_1 \dots x_m)$  is a constant number for different  $y_1 \dots y_m$ , it can be dropped in the argmax operation of Eq. (4.75), as follows

$$\hat{y}_1 \dots \hat{y}_m = \arg \max_{y_1 \dots y_m} \prod_{i=1}^m \Pr(y_i | y_{i-1}) \Pr(x_i | y_i) \quad (4.78)$$

To estimate  $\Pr(x_i | y_i)$ , a possible solution is to take  $\Pr(x_i | y_i) = \frac{\Pr(y_i | x_i) \Pr(x_i)}{\Pr(y_i)}$ , and use a neural network to compute  $\Pr(y_i | x_i)$ .

label sequences, and to allow us to make use of a variety of features to do this. Consider, for example, the **linear-chain CRF** [Sutton and McCallum, 2012]. It defines  $\Pr(y_1 \dots y_m | \mathbf{h}_1 \dots \mathbf{h}_m)$  in the following form

$$\begin{aligned} \Pr(y_1 \dots y_m | \mathbf{h}_1 \dots \mathbf{h}_m) &= \frac{\Pr(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m)}{\Pr(\mathbf{h}_1 \dots \mathbf{h}_m)} \\ &= \frac{\exp(\text{Score}(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m))}{Z(\mathbf{h}_1 \dots \mathbf{h}_m)} \end{aligned} \quad (4.79)$$

where  $Z(\mathbf{h}_1 \dots \mathbf{h}_m)$  is a normalization factor, and has the form

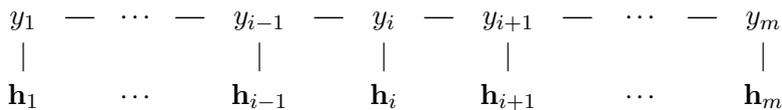
$$Z(\mathbf{h}_1 \dots \mathbf{h}_m) = \sum_{y'_1 \dots y'_m} \exp(\text{Score}(y'_1 \dots y'_m, \mathbf{h}_1 \dots \mathbf{h}_m)) \quad (4.80)$$

$\text{Score}(\cdot)$  is a score for weighting the sequence pair  $(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m)$ . It is given by summing over the values of a set of feature functions  $\{f_1(\cdot), \dots, f_J(\cdot)\}$ , like this

$$\text{Score}(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m) = \sum_{i=1}^m \sum_{j=1}^J f_j(y_i, y_{i-1}, \mathbf{h}_i) \quad (4.81)$$

The outer loop of the summation corresponds to a visit to each position  $i$ . Given  $i$ , each function  $f_j(\cdot)$  takes the current label  $y_i$ , the previous label  $y_{i-1}$  and the current input vector  $\mathbf{h}_i$ , and then returns the value of a feature.

This model is called *linear-chain* because it represents  $y_1 \dots y_m$  as a chain structure where each node  $y_i$ , along with an observed variable  $\mathbf{h}_i$ , only connects to its preceding node  $y_{i-1}$  and its following node  $y_{i+1}$ <sup>14</sup>, like this



In CRFs, it is assumed that the variables in the graph is only dependent on its neighboring variables. Therefore,  $f_j(y_i, y_{i-1}, \mathbf{h}_i)$  can be defined according to how  $y_i$  is connected. There are generally two types of features.

- **Transition-like Features.** This type of features models the connection between consecutive labels  $(y_{i-1}, y_i)$ , given by

$$f_1(y_i, y_{i-1}, \mathbf{h}_i) = u(y_{i-1}, y_i) \quad (4.82)$$

where  $u(y_{i-1}, y_i)$  is an entry of a weight matrix  $\mathbf{u}$ , indexed by  $(y_{i-1}, y_i)$ .

- **Emission-like Features.** The second type of features models the connection between a

<sup>14</sup>CRFs can broadly be categorized as a type of **undirected graphical models**. They define a graph over a set of observed variables and a set of unobserved variables. These variables are connected in some way that forms a graph.

label  $y_i$  and the associated input  $\mathbf{x}_i$ , given by

$$f_2(y_i, y_{i-1}, \mathbf{h}_i) = g_i(y_i) \quad (4.83)$$

where  $g_i(y_i)$  is the entry  $y_i$  of a vector  $\mathbf{g}_i \in \mathbb{R}^{|V_y|}$ . The vector  $\mathbf{g}_i$  represents the weights of associating  $\mathbf{h}_i$  with each label in the form

$$\mathbf{g}_i = \mathbf{h}_i \cdot \mathbf{v} \quad (4.84)$$

where  $\mathbf{v} \in \mathbb{R}^{d_h \times |V_y|}$  is a weight matrix.

To simplify notation, we use  $y_i$  (or  $y_{i-1}$ ) to denote the one-hot representation for a label<sup>15</sup>. Then, substituting the above feature functions into Eq. (4.81) allows the scoring function to be written in the form

$$\begin{aligned} \text{Score}(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m) &= \sum_{i=1}^m u(y_{i-1}, y_i) + g_i(y_i) \\ &= \sum_{i=1}^m y_{i-1} \cdot \mathbf{u} \cdot y_i^T + \mathbf{h}_i \cdot \mathbf{v} \cdot y_i^T \\ &= \sum_{i=1}^m (y_{i-1} \cdot \mathbf{u} + \mathbf{h}_i \cdot \mathbf{v}) \cdot y_i^T \end{aligned} \quad (4.85)$$

The right-hand side of the equation only involves simple algebraic operations on vectors and matrices, allowing viewing this model as a normal neural network. In this way, it is convenient to implement the sequence labeling system with various neural network toolkits. We just need to stack a CRF network on an encoder network and learn the entire network as usual. For example, one can train this system by maximum likelihood, and optimize the loss function by gradient descent. Note that, as with other chain or lattice-based models, the CRF network can be efficient because there are dynamic programming algorithms, called the forward-backward methods, for computing both  $\text{Score}(y_1 \dots y_m, \mathbf{h}_1 \dots \mathbf{h}_m)$  and  $Z(\mathbf{h}_1 \dots \mathbf{h}_m)$ . We refer the interested reader to related papers for more detailed discussions [Lafferty et al., 2001; Sutton and McCallum, 2012].

One advantage of marrying the worlds of distributed representation and sequence labeling is that we do not need to specify any feature templates as in conventional approaches. Instead, the model is free to learn features that describe whatever input sequences are most effective at optimizing some objective for sequence labeling. Such an architecture has been used as the backbone model for several state-of-the-art systems [Huang et al., 2015; Lample et al., 2016; Ma and Hovy, 2016; Li et al., 2020c].

<sup>15</sup>In this case,  $y_i \in \mathbb{R}^{|V_y|}$ , although it is originally used as a scalar.

### 4.5.4 Hybrid Models for Language Modeling

As we have already noted, many sequence modeling problems can be dealt with by either RNN-based or CNN-based models. Each of these two types of models has its own characteristics: RNNs are originally designed for dealing with variable-length temporal data, and CNNs are more effective in interpreting local information in restricted regions of input. Here we consider a hybrid approach to language modeling for obtaining the benefits of both.

Recall from Section 4.2.1 that a neural language model is learned to predict a probability distribution over a vocabulary, given some representation of the history words. Let  $w_1 \dots w_m$  be a word sequence to which we want to assign a probability. First, we represent each word  $w_i$  as a word vector  $\mathbf{x}_i$ . Then, an RNN model takes a word vector at a time and outputs the probability

$$\begin{aligned} \Pr(w_{i+1}|w_1 \dots w_i) &= \Pr(w_{i+1}|\mathbf{x}_1 \dots \mathbf{x}_i) \\ &= \Pr(w_{i+1}|\mathbf{h}_i) \end{aligned} \quad (4.86)$$

where  $\mathbf{h}_i$  is the state of the recurrent unit at step  $i$ .

The process of converting words from symbols to continuous representations plays an important role in this model. While it is common for practitioners to obtain  $\mathbf{x}_i$  from a word embedding table, this approach treats each word as a whole and simply ignores its internal structure. In consequence, it might be difficult to learn distinct vectors for rare words in languages with large vocabularies [Bojanowski et al., 2017].

Here we consider a different way of representing words. The idea is simple: an additional neural network is used to embed words [Ling et al., 2015; Kim et al., 2016]. Suppose every word  $w_i$  can be expressed as a sequence of characters. We represent these characters as real-valued vectors  $\mathbf{e}_{i,1} \dots \mathbf{e}_{i,\text{len}_i}$  via a character embedding table. Following Kim et al. [2016]’s work, we can use a CNN to represent  $\mathbf{e}_{i,1} \dots \mathbf{e}_{i,\text{len}_i}$  as a word vector in the following form

$$\begin{aligned} \mathbf{x}_i &= \text{CNN}(\mathbf{e}_{i,1} \dots \mathbf{e}_{i,\text{len}_i}, \mathbf{W}) \\ &= \text{Pooling}(\text{TanH}(\text{Conv}(\mathbf{e}_{i,1} \dots \mathbf{e}_{i,\text{len}_i}, \mathbf{W}))) \end{aligned} \quad (4.87)$$

where  $\text{Conv}(\cdot, \mathbf{W})$  is a convolutional layer with parameters  $\mathbf{W}$ ,  $\text{TanH}(\cdot)$  is a hyperbolic tangent function, and  $\text{Pooling}(\cdot)$  is a pooling layer.

Figure 4.13 shows the architecture of the model. We see that there is a hierarchical structure behind this model, that is, characters form a word, and words form a sentence or phrase. On a practical side, in many NLP tasks it is quite natural to consider the hierarchical nature of language. We will see a few examples of making use of the relationships between different levels of language representations in later chapters.

## 4.6 Summary

This chapter has introduced the recurrent and convolutional neural approaches to modeling sequences of words. On one hand, recurrent neural networks are designed for dealing with

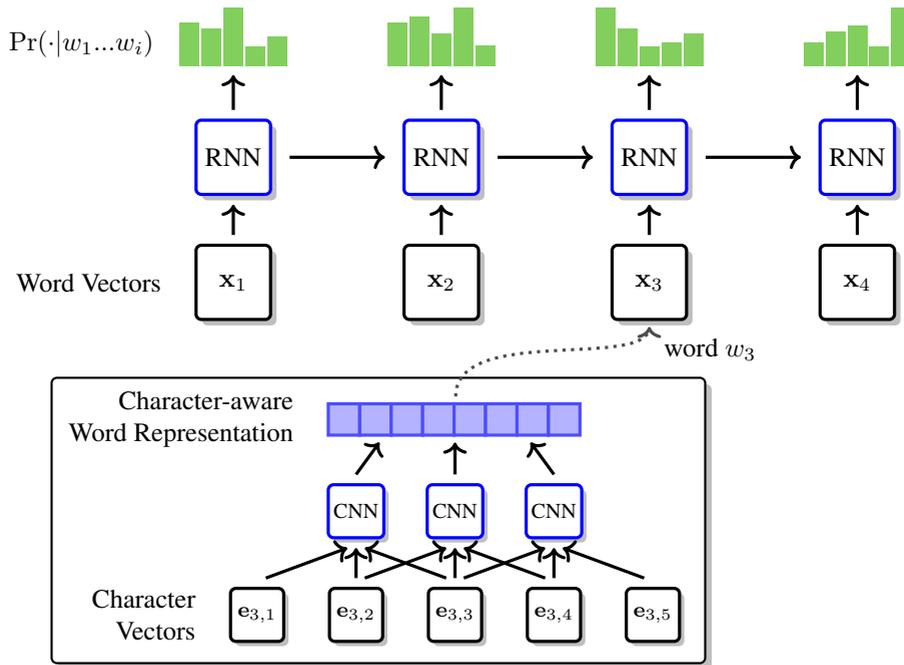


Figure 4.13: A language model with character-aware word representations [Kim et al., 2016]. As a language model, the goal of this model is to compute the probability  $\Pr(w_{i+1}|w_1 \dots w_i)$  for each  $i$ . We represent each word  $w_i$  as a real-valued vector  $x_i$ . This vector is the output of a CNN that takes a sequence of characters corresponding to this word. Then, the sequence of the word vectors  $x_1 \dots x_m$  is used as the input to an RNN + Softmax model. The model outputs at each position  $i$  a distribution of words, where the entry  $w_{i+1}$  describes  $\Pr(w_{i+1}|w_1 \dots w_i)$ . This hierarchical structure provides a multi-scale approach to language modeling: a sentence is modeled by considering words, and a word is modeled by considering characters.

sequential data, and have broad applicability in NLP. To improve the modeling power of these models, the memory mechanism is generally used. In particular, we have introduced LSTM and GRU which are two popular types of models in dealing with long sequence problems. On the other hand, while convolutional neural networks are commonly used to process vision data, they are straightforwardly applicable to sequence modeling. We have seen that all these models can be used in several language and speech processing tasks, including text classification, speech recognition, sequence labeling, and language modeling.

The roots of modeling sequences of language units can be traced back to early work in several different fields. For example, the process of generating a sequence of words can be described as a Markov chain where the prediction of a word only depends on a limited number of previous words [Markov, 1913]. This idea motivates the  $n$ -gram methods for sequence modeling [Shannon, 1948a], as well as hidden Markov models which later appeared and became popular in modeling sequences of pairs of observed and unobserved variables [Baum and Petrie, 1966; Baum et al., 1970]. These models and their variants lay the foundations of many successful NLP systems in past decades [Manning and Schütze, 1999; Jurafsky and

Martin, 2008].

The idea of using neural networks in sequence modeling has also been investigated for some time. One example to see how neural networks are developed and applied to sequence modeling is speech recognition [Lippmann, 1989]. Most of the studies in the early days of this research area try to either combine neural networks with existing models [Bouillard and Wellekens, 1990; Bouillard and Morgan, 1993; Trentin and Gori, 2001], or address sub-problems of speech recognition [Tank and Hopfield, 1987; Waibel et al., 1989; Lang et al., 1990; Bengio, 1991]. However, scaling neural networks up in size was challenging because training deep neural networks requires a lot of computation resources and data. The field had long been dominated by approaches based on hidden Markov models and **Gaussian mixture models (GMMs)**, with a pipeline of several modules that require careful tuning. On the other hand, while fully neural approaches were not state-of-the-art during that time, researchers were aware of their potential in learning representations of acoustic inputs and freeing them from hand-crafted features [LeCun and Bengio, 1995].

A dramatic shift from conventional pipelined approaches to end-to-end approaches comes with the revival of neural networks in the 2000s [Hannun et al., 2014; Graves et al., 2013b]. The shift is so influential that a broad set of fields comes together like never before, e.g., in computer vision and speech processing, the past ten years have, meanwhile, witnessed great performance gains brought by very deep neural networks and end-to-end learning [Hinton et al., 2006; Graves et al., 2013b; He et al., 2016a; Krizhevsky et al., 2017]. In NLP, the paradigm shift starts with the work on word embeddings [Mikolov et al., 2013a; Pennington et al., 2014], and continues as more powerful sequence representation models are developed [Vaswani et al., 2017]. A simple approach to sequence modeling, though not discussed in depth in this chapter, is **compositional models** [Janssen, 2012]. For example, we can use the bag-of-words model to sum or average word vectors of a sequence. Despite the simple architectures of these approaches, they achieve satisfactory results in many tasks, providing strong baselines for further research on more advanced models [Conneau et al., 2018]. As the next step, applying recurrent and convolutional neural models to sequence modeling is straightforward. This is not surprising because these models are fairly well studied in other fields [Lipton et al., 2015; Li et al., 2021d; Khan et al., 2020]. In particular, the LSTM model is well suited to deal with long sequences and thus of great interest to NLP researchers [Sundermeyer et al., 2012; Huang et al., 2015; Wu et al., 2016]. However, we are always on the way. Learning sequence models is one of the most active research fields with no end in sight. There are many models that are based on new architectures and show stronger performance in various tasks. More discussions on some of these models can be found in Chapters 6, 7 and 8.

Note that the term *sequence modeling* is currently used in many different ways, referring to different tasks. In many cases it is more common to use the terms *encoding* and *encoder* to emphasize the process of mapping a sequence of symbols to a continuous representation. As discussed in the previous sections, a benefit of viewing encoding as an individual task is that we can learn a general representation model that is not dependent on where we apply it. It opens the door to a wide range of pre-trained encoders for learning to represent various types of data, such as text [Peters et al., 2018; Devlin et al., 2019], speech [Oord et al., 2018;

Hsu et al., 2021; Chen et al., 2022], vision [Chen and He, 2021; Bao et al., 2021; He et al., 2022], and combinations of them [Chuang et al., 2020; Li et al., 2021c; Kim et al., 2021]. A closely related concept to text encoding is **text embedding** or **sentence embedding** [Conneau et al., 2017a; Cer et al., 2018]. These can be broadly considered the same thing. In general, an embedding model in NLP means a process of transforming the input text into a single low-dimensional vector rather than producing sequences of contextualized vectors [Kiros et al., 2015; Hill et al., 2016].

In many NLP problems, systems are not necessarily sequential on their input and/or output. For example, in text classification, a system may take tree-structured input and produces a label [Tai et al., 2015; Yang et al., 2016]. In this case we need some mechanism to encode hierarchical structures. An alternative approach is to convert trees to sequences (or **linearized trees**) so that we can directly make use of sequence models to handle non-sequential data [Vinyals et al., 2015]. This is a great idea because it opens up the possibility of developing a universally applicable encoder to represent various types of data if the input of the encoder can be linearized in some way. For example, by representing an image as a sequence of patches, sequence models can be directly applied to image classification, achieving state-of-the-art results on several tasks [Chen et al., 2020a; Dosovitskiy et al., 2021].

# Chapter 5

## Sequence-to-Sequence Models

天下万物之理，无独必有对。

*According to the Principle of Heaven and Earth and all things, nothing exists in isolation but everything necessarily has its opposite.*

– 《近思录》

*Reflections on things at hand*

朱熹/Xi Zhu (AD 1130-1200)

吕祖谦/Zuqian Lv (AD 1137-1181)

translated by [Chang \[1967\]](#)

In the language world, things often come in pairs. If there is a question, there would be an answer; if there is a Chinese text, there would be an English translation; if there is a sentence, there would be a parse of it according to some syntax. Many NLP systems are designed to model the correspondence between these pairs, i.e., one of the two is taken as input and the other is taken as output. These problems can be expressed in a form that we have encountered several times, like this

$$\hat{y} = \arg \max_y \Pr(y|x) \quad (5.1)$$

where  $x$  is an input variable,  $y$  is an output variable, and  $\Pr(y|x)$  is a model that estimates how likely  $y$  would be the true output given  $x$ .

This chapter is more interested in a particular family of problems where both  $x$  and  $y$  are sequences of words, called **sequence-to-sequence** (or **seq2seq**) problems. Unlike classification problems where the output  $\hat{y}$  is selected from a fixed set of classes, sequence-to-sequence problems require producing an output from an exponentially larger set of sequences. Obtaining  $\hat{y}$  in this case turns out to be a much more complex problem than the case of classification, because we need more powerful models to describe  $\Pr(y|x)$  and more efficient search algorithms to solve Eq. (5.1).

This chapter will discuss the well-known **encoder-decoder architecture** for sequence-to-sequence modeling. Also, this chapter will discuss the attention mechanism which is an improvement on this architecture. Both of these models lay the foundation of discussions of several state-of-the-art models in the following chapters. Furthermore, this chapter will discuss the search problem which plays an important role in sequence generation and related problems.

## 5.1 Sequence-to-Sequence Problems

We choose machine translation as an illustrative example throughout this chapter, because it is now one of the most popular sequence-to-sequence tasks. We use  $\mathbf{x} = x_1 \dots x_m$  to denote a sequence of words in one language (call it a **source-side sequence** or **source sequence**), and use  $\mathbf{y} = y_1 \dots y_n$  to denote a sequence of words in another language (call it a **target-side sequence** or **target sequence**). We can write Eq. (5.1) using the new notation, as follows

$$\begin{aligned} \hat{\mathbf{y}} &= \arg \max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{x}) \\ &= \arg \max_{y_1 \dots y_n} \Pr(y_1 \dots y_n | x_1 \dots x_m) \end{aligned} \quad (5.2)$$

As discussed in Chapter 1 and in [Brown et al., 1993], this formulation implies three fundamental issues.

- **Modeling.** First, we need to define the form of  $\Pr(\mathbf{y}|\mathbf{x})$ . In this chapter we show that  $\Pr(\mathbf{y}|\mathbf{x})$  can be computed using a single neural network based on the encoder-decoder architecture and the attention mechanism. Note that sometimes we just need a model for discriminating “good” from “bad” target sequences. In this case, it is not necessary to require the model to make probability sense, and we can take a discriminant function instead.
- **Training.** Then, we need to learn parameters of the model  $\Pr(\mathbf{y}|\mathbf{x})$  given some training data. As  $\Pr(\mathbf{y}|\mathbf{x})$  is expressed as a neural network, we can train it in a regular way: we optimize some loss by gradient descent. See Chapter 3 for common approaches to training neural networks. We will also discuss techniques that are tailored for specific tasks in this and the following chapters.
- **Search (or Decoding).** Once we have learned a model, we will obtain  $\hat{\mathbf{y}}$  by searching for the target sequence that maximizes  $\Pr(\mathbf{y}|\mathbf{x})$ . This is a computational challenge because the number of candidate sequences grows with the maximum length of the sequences and the size of the vocabulary. In Section 5.4, we will discuss efficient and effective search methods for sequence-to-sequence problems, particularly for machine translation.

Many NLP problems that fit the form of Eq. (5.2) can fall into sequence-to-sequence problems, and the research on these problems is largely motivated by discussions of the above issues. Table 5.1 shows common examples of sequence-to-sequence problems taken from the literature. When the target-side is a text, the problems can broadly be categorized as the **text generation** problems, although a general text generation system does not require the

Task	Source	Target
Machine Translation	Text in One Language	Translation in Another Language
Question Answering	Question	Answer
Dialogue Systems	Text/Speech for Conversation	Response
Summarization	Long Text	Summaries of the Text
Text Simplification	Text	Simpler Text
Text Style Transfer	Text in One Style	Same Content in Another Style
Grammar Correction	Text with Errors	Corrected Text
Speech Recognition	Speech	Transcription
Speech Synthesis	Text	Speech
Speech Translation	Speech in One Language	Translation in Another Language

Table 5.1: Examples of sequence-to-sequence problems.

source-side to be sequential. In addition to language and speech processing, sequence-to-sequence problems can be generalized to cases where the input and/or output of a system are not naturally sequential. For example, **image-to-text generation** (or **image captioning**) and **text-to-image generation** systems both involve dealing with images that are typically represented as 2D data. By representing images as sequences in some way (such as sequences of patches), sequence-to-sequence models are directly applicable to these tasks.

Historically, most systems in these tasks were developed somewhat independently, resulting in different architectures, features, and training methods for different tasks. However, as shown in this chapter, when we represent these models as neural networks and train them in an end-to-end fashion, there appears to be a “universal” paradigm for all these problems. This is a big change for the AI community because many research fields come together and systems can be shared across them. We can gain some insight into the common nature of a broad variety of problems, though there are many task-specific considerations in practice. In the following sections, we will discuss some of the common threads among sequence-to-sequence models.

## 5.2 The Encoder-Decoder Architecture

In this section we discuss the encoder-decoder architecture and a simple neural machine translation model based on this architecture.

### 5.2.1 Encoding and Decoding

From a supervised learning viewpoint, we would ideally like to learn a model from a number of sequence pairs such that any source-side sequence can be mapped to the corresponding target-side sequence. However, learning the mapping between sequences of discrete variables

is typically a problem of learning from high-dimensional data. It inevitably suffers from the curse of dimensionality, making the modeling and training difficult.

One approach to learning such a mapping is to divide the problem into “simpler” sub-problems. We assume that there is a low-dimensional representation shared by  $\mathbf{x}$  and  $\mathbf{y}$ , denoted by  $\mathbf{H}$ . Then, the mapping  $\mathbf{x} \rightarrow \mathbf{y}$  can be achieved by mapping  $\mathbf{x}$  to  $\mathbf{H}$  and then to  $\mathbf{y}$ . Formally, given a source-side sequence  $\mathbf{x}$ , we map it to the representation  $\mathbf{H}$  by using an **encoding system** (call it an encoder)

$$\mathbf{H} = \text{Encode}(\mathbf{x}) \quad (5.3)$$

Then, we map  $\mathbf{H}$  to the target-side sequence  $\mathbf{y}$  by using a **decoding system** (call it a decoder)<sup>1</sup>

$$\mathbf{y} = \text{Decode}(\mathbf{H}) \quad (5.4)$$

This architecture, also known as the encoder-decoder architecture, is widely used in recent sequence-to-sequence systems (see Figure 5.1 for an illustration). It is easy to see that the form of Eq. (5.3) is the same as those of the sequence models mentioned in Chapter 4, and so there are many encoding models to choose from, such as bi-directional LSTM. The goal of the decoder is to produce a “best” target-side sequence given the representation of the source-side sequence. Like classification models, the prediction is made by first producing a distribution over all possible sequences, and then selecting the one with the maximum probability. As such, we can re-define  $\text{Decode}(\cdot)$  as a probability function

$$\begin{aligned} \text{Pr}(\cdot|\mathbf{H}) &= \text{Decode}(\mathbf{H}) \\ &= \text{Decode}(\text{Encode}(\mathbf{x})) \end{aligned} \quad (5.5)$$

In other words, given a target-side sequence  $\mathbf{y}$ , the decoder assigns it a probability

$$\text{Pr}(\mathbf{y}|\mathbf{x}) = \text{Pr}(\mathbf{y}|\mathbf{H}) \quad (5.6)$$

Then, the optimal sequence  $\hat{\mathbf{y}}$  is obtained by performing  $\arg \max_{\mathbf{y}} \text{Pr}(\mathbf{y}|\mathbf{x})$  as in Eq. (5.2). In many systems based on the encoder-decoder architecture, both  $\text{Encode}(\cdot)$  and  $\text{Decode}(\cdot)$  are models constructed from neural networks. Thus, we can treat the sequence-to-sequence model

---

<sup>1</sup>It is important to distinguish between the concept of *decoding* (or *decoder*) used in conventional sequence-to-sequence systems and that used in the encoder-decoder architecture. The two are often confused, though they are different somehow. In many machine translation or speech recognition systems, *decoding* has the same meaning as *translation* or *transcription*, that is, we recover the optimal  $\mathbf{y}$  from  $\mathbf{x}$ . As pointed out in Eq. (5.2), this process involves a search over all candidate  $\mathbf{y}$ . Therefore, the conventional use of decoding in these systems is to refer to a search process (i.e., the  $\arg \max$  operation in Eq. (5.2)) [Koehn, 2010]. By contrast, in the encoder-decoder architecture *decoding* means a process of recovering the target-side sequence  $\mathbf{y}$  from the intermediate representation  $\mathbf{H}$ . It is all about modeling rather than searching. It is also worth noting that, while the term *decoding* (or *decoder*) is used in different ways, it can be thought of as a process of mapping an encoded message back to the original message in a communication system as defined in information theory [Shannon, 1948b]. In this sense, the *decoding* processes in these systems do the same thing as the word sounds like: convert something to its original form.

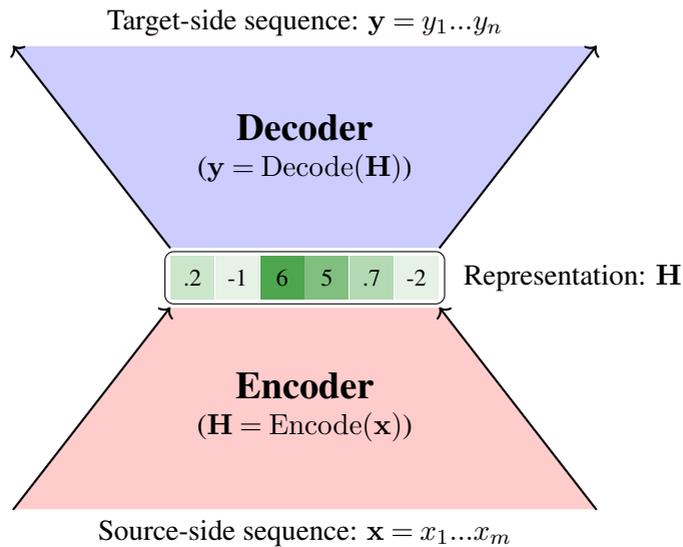


Figure 5.1: The encoder-decoder architecture. In the case of sequence-to-sequence problems, it transforms a source-side sequence  $\mathbf{x} = x_1 \dots x_m$  to a target-side sequence  $\mathbf{y} = y_1 \dots y_n$ . This procedure involves two steps:  $\mathbf{x}$  is first encoded as a representation  $\mathbf{H}$ , and this representation is then decoded to  $\mathbf{y}$ .

as a single neural network and train it as usual, provided the entire model is some combination of  $\text{Encode}(\cdot)$  and  $\text{Decode}(\cdot)$ .

To apply the encoder-decoder architecture to a real-world task, we need to make a number of design choices, such as the forms of  $\mathbf{H}$ ,  $\text{Encode}(\cdot)$  and  $\text{Decode}(\cdot)$ . As a very simple example, consider the task of regenerating an input word. We can define  $\text{Encode}(\cdot)$  as a feed-forward neural network that takes a word (in one-hot representation) and outputs a word vector. In this way,  $\mathbf{H}$  is a distributed representation of the word. Then, we define  $\text{Decode}(\cdot)$  as another feed-forward neural network that takes the word vector and generates a distribution over the vocabulary. For training, we wish to learn a system that assigns the largest probability to the input word. As discussed in Chapter 2, we can call this an auto-encoder which is a special instance of the encoder-decoder architecture.

## 5.2.2 Example: Neural Machine Translation

Next we illustrate the application of the encoder-decoder architecture using a working example — **neural machine translation (NMT)**. We consider a well-known NMT model which uses RNN or its variants for building both the encoder and decoder [Cho et al., 2014; Sutskever et al., 2014]. The encoder of the NMT model is a standard RNN-based encoder. As the RNN-based sequence model has been discussed in detail in Chapter 4, we just give a brief review of this model here. Suppose that the source-side vocabulary is  $V_x$  and each source-side word  $x_j$  is represented as a one-hot vector in  $\mathbb{R}^{|V_x|}$ . Then,  $x_j$  is transformed into a  $h_s$ -dimensional vector

(or word embedding)

$$\mathbf{x}_j^e = \text{Embed}_s(x_j) \quad (5.7)$$

where  $\text{Embed}_s(\cdot)$  is the word embedding function. More details about word embedding models can be found in Chapter 3.

The RNN model takes the sequence of the word vectors  $\mathbf{x}_1^e \dots \mathbf{x}_m^e$  and produces a sequence of RNN state vectors  $\mathbf{h}_1 \dots \mathbf{h}_m$ . An RNN state vector  $\mathbf{h}_j \in \mathbb{R}^{d_h}$  is defined to be

$$\mathbf{h}_j = \text{RNN}(\mathbf{h}_{j-1}, \mathbf{x}_j^e) \quad (5.8)$$

Here  $\text{RNN}(\cdot)$  is an RNN unit that summarises the information up to position  $j$  by combining the previous state  $\mathbf{h}_{j-1}$  and the current input  $\mathbf{x}_j^e$  in some way. Then, the last state  $\mathbf{h}_m$  can be treated as a representation of the input sequence  $x_1 \dots x_m$ , and we can use  $\mathbf{h}_m$  as the output of the encoder, written as

$$\mathbf{h}_m = \text{Encode}(x_1 \dots x_m) \quad (5.9)$$

Figure 5.2 (a-b) shows an illustration of the encoding process. Note that the model described above just involves a single-layer RNN. In practical systems, this framework can be easily extended to include multiple layers and more powerful recurrent units (such as LSTM units).

The decoder of the NMT model is a standard RNN-based language model, that is, we predict the next word  $y_{i+1}$  given all previous words  $y_1 \dots y_i$ . To incorporate the source-side information into translation, a simple and straightforward method is to treat  $\mathbf{h}_m$  as the initial state of the target-side RNN. Let  $\mathbf{y}_0^e \in \mathbb{R}^{d_s}$  be the word vector of the start symbol (SOS) (denoted by  $y_0$ ). The corresponding RNN state is given by

$$\mathbf{s}_0 = \text{RNN}(\mathbf{h}_m, \mathbf{y}_0^e) \quad (5.10)$$

Here  $\text{RNN}(\cdot)$  has the same form as the recurrent unit used in the encoder, but with different parameters.

For  $i > 0$ , the state vector  $\mathbf{s}_i \in \mathbb{R}^{d_s}$  is given in the form

$$\mathbf{s}_i = \text{RNN}(\mathbf{s}_{i-1}, \mathbf{y}_i^e) \quad (5.11)$$

Then,  $\mathbf{s}_i$  is fed into a Softmax layer to produce a distribution over the target-side vocabulary  $V_y$ . The output of the Softmax layer is given by

$$\begin{aligned} \Pr(\cdot | y_1 \dots y_i, x_1 \dots x_m) &= \Pr(\cdot | \mathbf{s}_i) \\ &= \text{Softmax}(\mathbf{s}_i \mathbf{U}_y + \mathbf{b}_y) \end{aligned} \quad (5.12)$$

where  $\mathbf{U}_y \in \mathbb{R}^{d_s \times |V_y|}$  and  $\mathbf{b}_y \in \mathbb{R}^{|V_y|}$  are the parameters of the Softmax layer.  $\Pr(y_{i+1} | y_1 \dots y_i, x_1 \dots x_m)$  can be seen as the probability of predicting word  $y_{i+1}$  by conditioning on both the translated

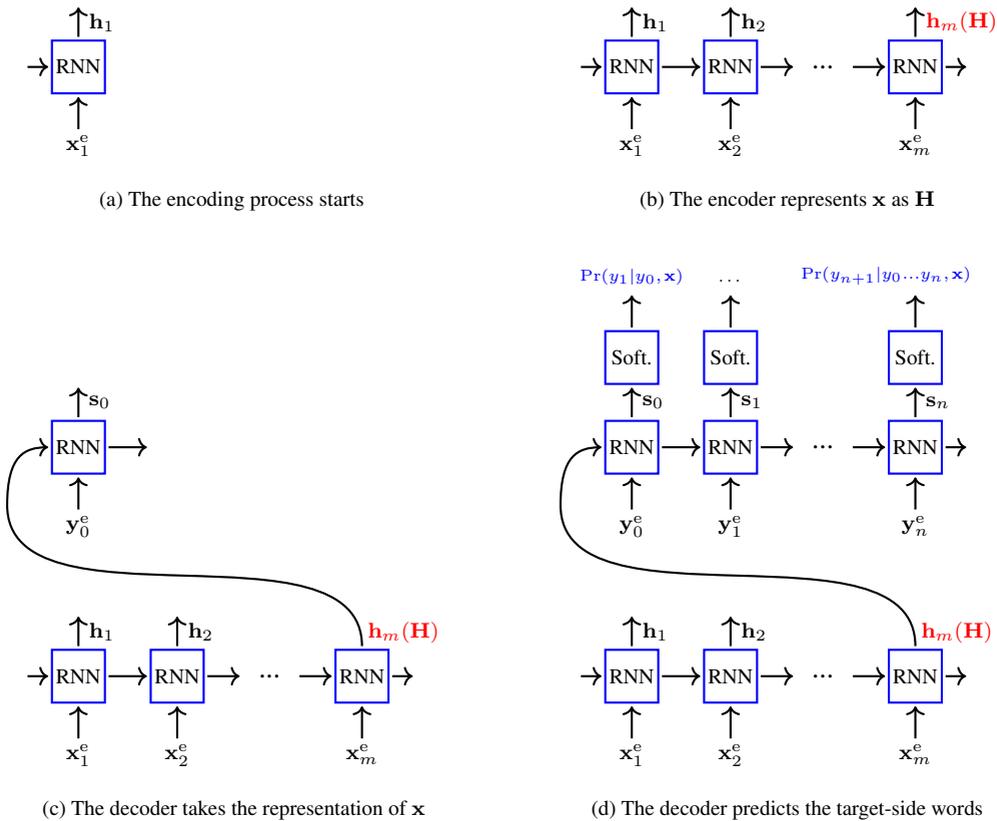


Figure 5.2: The encoding and decoding steps for an RNN-based NMT system. The encoder is a standard RNN. The encoding process starts with the first source-side word and ends up with the last source-side word. The last state of the RNN is taken to be the representation of the entire source-side sequence (i.e.,  $\mathbf{H} = \mathbf{h}_m$ ). The decoder is another RNN. At the first step, it takes  $\mathbf{H}$  from the encoder. After representing  $(\mathbf{y}_0 \dots \mathbf{h}_i^e, \mathbf{H})$  as  $s_i$  at position  $i$ , a softmax layer is built to predict the next word  $y_{i+1}$ .

words  $y_1 \dots y_i$  and the source-side sequence  $x_1 \dots x_m$ . See Figure 5.2 (c-d) for an illustration of the word predictions of a decoder.

Armed with this model of word prediction, we turn to a form that is frequently used in papers on NMT, like this

$$\begin{aligned}
 \Pr(\mathbf{y}|\mathbf{x}) &= \Pr(y_0\mathbf{y}|\mathbf{x}) \\
 &= \Pr(y_0y_1\dots y_n|x_1\dots x_m) \\
 &= \Pr(y_0|x_1\dots x_m)\Pr(y_1\dots y_n|y_0, x_1\dots x_m) \\
 &= \prod_{i=0}^{n-1} \Pr(y_{i+1}|y_0\dots y_i, x_1\dots x_m)
 \end{aligned} \tag{5.13}$$

Sometimes, this equation is also written in an equivalent form

$$\Pr(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^n \Pr(y_i|y_0 \dots y_{i-1}, x_1 \dots x_m) \quad (5.14)$$

Here we assume that  $\mathbf{y}$  always starts with  $y_0$  (i.e.,  $\langle \text{SOS} \rangle$ ) and so  $\Pr(y_0|x_1 \dots x_m) = 1$ . In many practical systems, it is also common to assume that  $\mathbf{y}$  ends with a special symbol  $\langle \text{EOS} \rangle$ . Therefore, we can modify this equation to involve  $\langle \text{SOS} \rangle$  and  $\langle \text{EOS} \rangle$  on both the source and target-sides, as follows

$$\begin{aligned} \Pr(y_0 \mathbf{y} y_{n+1} | x_0 \mathbf{x} x_{m+1}) &= \Pr(y_0 y_1 \dots y_n y_{n+1} | x_0 x_1 \dots x_m x_{m+1}) \\ &= \Pr(y_0 | x_0 \dots x_{m+1}) \cdot \\ &\quad \Pr(y_1 \dots y_n y_{n+1} | y_0, x_0 \dots x_{m+1}) \\ &= \prod_{i=0}^n \Pr(y_{i+1} | y_0 \dots y_i, x_0 \dots x_{m+1}) \end{aligned} \quad (5.15)$$

where  $x_0 = y_0 = \langle \text{SOS} \rangle$ ,  $x_{m+1} = y_{n+1} = \langle \text{EOS} \rangle$ , and  $\Pr(y_0 | x_0 x_1 \dots x_m x_{m+1}) = 1$ .

Since  $\Pr(\mathbf{y}|\mathbf{x})$  can be expressed as a neural network, training this model is straightforward. As described in Chapter 4, RNN-based language models are trained by using the cross-entropy loss and gradient descent. NMT can use this same method for training model parameters. Once we have obtained the optimized model, we can then use it to translate new sentences. Finding the best translation for any given source-side sentence is a standard search problem. We will discuss it in Section 5.4.

## 5.3 The Attention Mechanism

The NMT model discussed in the previous section was based on a fixed-length representation of the source-side sequence. While this model is easy to implement, in many practical applications it is unsatisfactory because a fixed-length vector might not be sufficient for representing a variable-length sequence, especially when the sequence is long. This system will therefore need some mechanism to couple the encoder and the decoder in a fine-grained manner. In this section we discuss the attention mechanism by which a system can learn, for each word of the target-side sequence, an adaptive representation that focuses more on important parts of the source-side sequence.

In fact, the discussion here is related to the attention models in psychology because translation is itself a cognitive process [Sternberg, 1996; Neisser, 2014]. The key idea behind this type of model is natural: attention is generally concentrated on specific parts of the data when we process something. This forms the basis of many state-of-the-art sequence-to-sequence models, and the attention mechanism has been the de facto standard for the development of these systems.

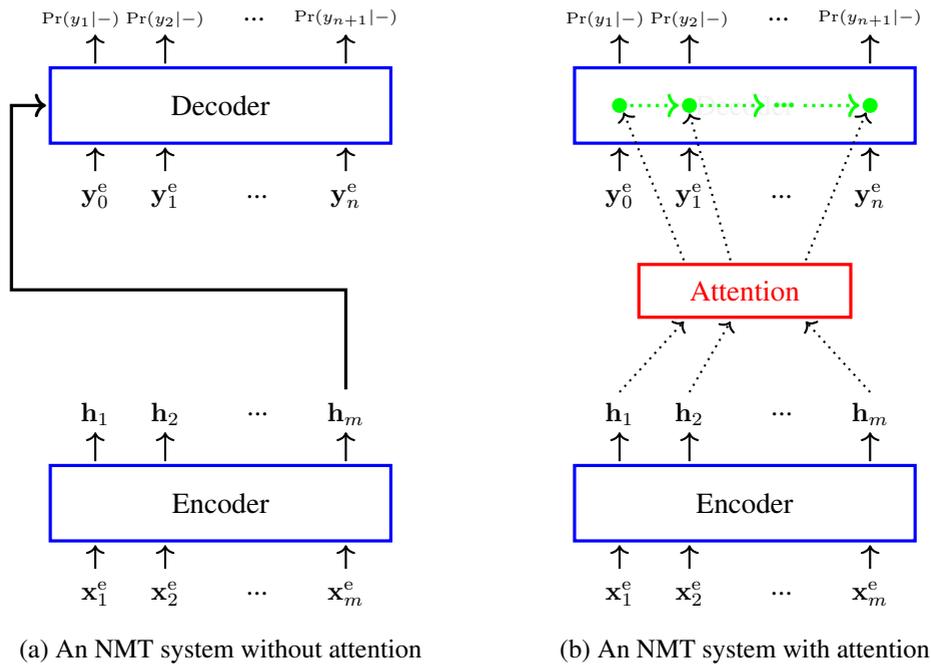


Figure 5.3: NMT architectures without (left) and with (right) the attention model. When the attention model is not involved, a fixed-length representation is considered for generating the entire target-side sequence. By contrast, when the attention model is involved, a new representation is computed specifically for each target-side state so that the decoder can learn to concentrate on different parts of the source-side sequence for predicting a target-side word.

### 5.3.1 A Basic Model

Recall that in the NMT model of the previous section, the encoder represents a source-side word sequence as  $\mathbf{h}_1 \dots \mathbf{h}_m$ , and the decoder represents a target-side word sequence as  $\mathbf{s}_1 \dots \mathbf{s}_n$ . The attention mechanism addresses the question of how a representation can be learned from  $\mathbf{h}_1 \dots \mathbf{h}_m$  so that this representation can explain the source-side sequence well for a given target state  $\mathbf{s}_i$ <sup>2</sup>. From an information processing perspective, so long as we ignore the meanings of  $\mathbf{h}_1 \dots \mathbf{h}_m$  and  $\mathbf{s}_i$  in NMT, attention can be thought of as a generic process of processing the input information  $\mathbf{h}_1 \dots \mathbf{h}_m$  by considering how each  $\mathbf{h}_j$  is related to the interest  $\mathbf{s}_i$ . Figure 5.3 compares NMT architectures with and without the attention mechanism.

More formally, an attention model produces a linear combination of  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  in the form

$$\mathbf{c}_i = \sum_{j=1}^m \alpha_{i,j} \cdot \mathbf{h}_j \quad (5.16)$$

where  $\alpha_{i,j}$  is the **attention weight** that describes how much the model should rely on  $\mathbf{h}_j$  when

<sup>2</sup>Following the convention in machine translation [Brown et al., 1993], we use  $j$  to represent a position in the source-side sequence, and use  $i$  to represent a position in the target-side sequence.

computing  $\mathbf{c}_i$  for  $\mathbf{s}_i$ . Sometimes  $\mathbf{c}_i$  is also called a **context vector**.

A common approach to computing attention weights is to normalize **alignment scores** in the following form

$$\begin{aligned}\alpha_{i,j} &= \text{Softmax}(a(\mathbf{s}_i, \mathbf{h}_j)) \\ &= \frac{\exp(a(\mathbf{s}_i, \mathbf{h}_j))}{\sum_{j'=1}^m \exp(a(\mathbf{s}_i, \mathbf{h}_{j'}))}\end{aligned}\quad (5.17)$$

Here the alignment score  $a(\mathbf{s}_i, \mathbf{h}_j)$  measures how strong  $\mathbf{h}_j$  is related to  $\mathbf{s}_i$ . In general,  $a(\mathbf{s}_i, \mathbf{h}_j)$  can be defined in several different ways [Graves et al., 2014; Bahdanau et al., 2014; Luong et al., 2015]. A comprehensive list of these functions can be found in survey papers on this subject [Chaudhari et al., 2021]. Here we introduce some of the common ones.

- **Dot-product Attention.** One of the simplest methods is to measure the similarity between  $\mathbf{h}_j$  and  $\mathbf{s}_i$ . Thus, we can calculate the dot-product of the two vectors, as follows

$$\begin{aligned}a(\mathbf{s}_i, \mathbf{h}_j) &= \mathbf{s}_i \mathbf{h}_j^T \\ &= \sum_{k=1}^{d_h} s_i(k) \cdot h_j(k)\end{aligned}\quad (5.18)$$

A variant of this model, called **scaled dot-product attention**, adds a scalar factor  $\frac{1}{\beta}$  to the right-hand side of Eq. (5.18), as follows

$$a(\mathbf{s}_i, \mathbf{h}_j) = \frac{\mathbf{s}_i \mathbf{h}_j^T}{\beta}\quad (5.19)$$

We will see an example of this model later in this section.

- **Cosine Attention.** Another commonly used similarity measure in vector algebra is the cosine of the angle between two vectors, given by

$$\begin{aligned}a(\mathbf{s}_i, \mathbf{h}_j) &= \cos(\mathbf{s}_i, \mathbf{h}_j) \\ &= \frac{\mathbf{s}_i \mathbf{h}_j^T}{\|\mathbf{s}_i\|_2 \cdot \|\mathbf{h}_j\|_2}\end{aligned}\quad (5.20)$$

where  $\|\mathbf{a}\|_2 = (\mathbf{a} \cdot \mathbf{a})^{\frac{1}{2}}$  is the Euclidean norm of the vector  $\mathbf{a}$ .

- **Weighted Dot-product Attention.** This attention model involves a linear mapping of the input vectors before performing the dot-product operation, given by

$$a(\mathbf{s}_i, \mathbf{h}_j) = \mathbf{s}_i \mathbf{W}_a \mathbf{h}_j^T\quad (5.21)$$

where  $\mathbf{W}_a \in \mathbb{R}^{d_h \times d_h}$  is the parameter matrix of the linear mapping. Both this approach and the dot-product attention approach are also called **multiplicative attention** [Ruder, 2017].

- **Additive Attention.** In additive attention, the entries of the two vectors are summed in some way. A widely-used form is given by Bahdanau et al. [2014]

$$a(\mathbf{s}_i, \mathbf{h}_j) = \mathbf{v}_a^T \text{TanH}(\mathbf{s}_i \mathbf{W}_s + \mathbf{h}_j \mathbf{W}_h) \quad (5.22)$$

where  $\mathbf{W}_h, \mathbf{W}_s \in \mathbb{R}^{d_h \times d_a}$  and  $\mathbf{v}_a \in \mathbb{R}^{d_a}$  are parameters.  $\text{TanH}(\mathbf{s}_i \mathbf{W}_s + \mathbf{h}_j \mathbf{W}_h)$  produces a  $d_a$ -dimensional vector where each entry is a transformed weighted sum of the entries of  $\mathbf{h}_j$  and  $\mathbf{s}_i$ . It is followed by a dot-product with another weight vector  $\mathbf{v}_a$ .

Now let us return to Eqs. (5.16-5.17) and rethink the role of attention weights. Eq. (5.17) informally defines a “distribution” over  $\mathbf{h}_1 \dots \mathbf{h}_m$ , written as

$$\Pr(\mathbf{h}_j | \mathbf{s}_i) = \alpha_{i,j} \quad (5.23)$$

If we consider  $\mathbf{h}$  a random variable that takes a value from  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ , then  $\alpha_{i,j}$  can be thought of as the probability of  $\mathbf{h} = \mathbf{h}_j$ , conditioned on  $\mathbf{s}_i$ , and Eq. (5.16) can be rewritten as

$$\begin{aligned} \mathbf{c}_i &= \sum_{j=1}^m \Pr(\mathbf{h}_j | \mathbf{s}_i) \cdot \mathbf{h}_j \\ &= \mathbb{E}_{\mathbf{h} \sim \Pr(\mathbf{h} | \mathbf{s}_i)}(\mathbf{h}) \end{aligned} \quad (5.24)$$

In other words,  $\mathbf{c}_i$  can be viewed as an **expected representation** of the source-side sequence given the target-side state  $\mathbf{s}_i$ , that is, the expectation of  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  under the distribution  $\Pr(\mathbf{h}_j | \mathbf{s}_i)$ . This provides a general framework for describing the way the decoder receives the information from the encoder: the decoder is a **receiver** that determines how much information is accepted from each **sender**. For example, in the NMT model of the previous section, there is only one sender  $\mathbf{h}_m$ , and so the receiver receives all the information the sender sends. By contrast, in the NMT model armed with the attention mechanism, there are  $m$  senders  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  and the receiver receives information according to a distribution of preferences for the senders.

It is straightforward to introduce the attention model into the process of word prediction. We modify our treatment of  $\mathbf{s}_i$  so as to make use of both the source-side and target-side information at each decoding step. We slightly modify the definition of  $\mathbf{s}_i$  to include the context vector corresponding to the previous state  $\mathbf{s}_{i-1}$ , as follows

$$\mathbf{s}_i = \text{RNN}(\mathbf{s}_{i-1}, \mathbf{c}_{i-1}, \mathbf{y}_i^e) \quad (5.25)$$

Compared with the model of Eq. (5.11), the model of Eq. (5.25) takes  $\mathbf{c}_{i-1}$  as an additional input. Therefore, this model considers both the representation of the target-side words  $y_1 \dots y_{i-1}$  (as encoded in  $\mathbf{s}_{i-1}$  and  $\mathbf{y}_i^e$ ) and the representation of the entire source-side sequence  $x_1 \dots x_m$  (as encoded in  $\mathbf{c}_{i-1}$ ). Then, the distribution of target words at position  $i$  can be conditioned on  $\mathbf{s}_i$  as usual

$$\Pr(\cdot | y_1 \dots y_i, x_1 \dots x_m) = \Pr(\cdot | \mathbf{s}_i) \quad (5.26)$$

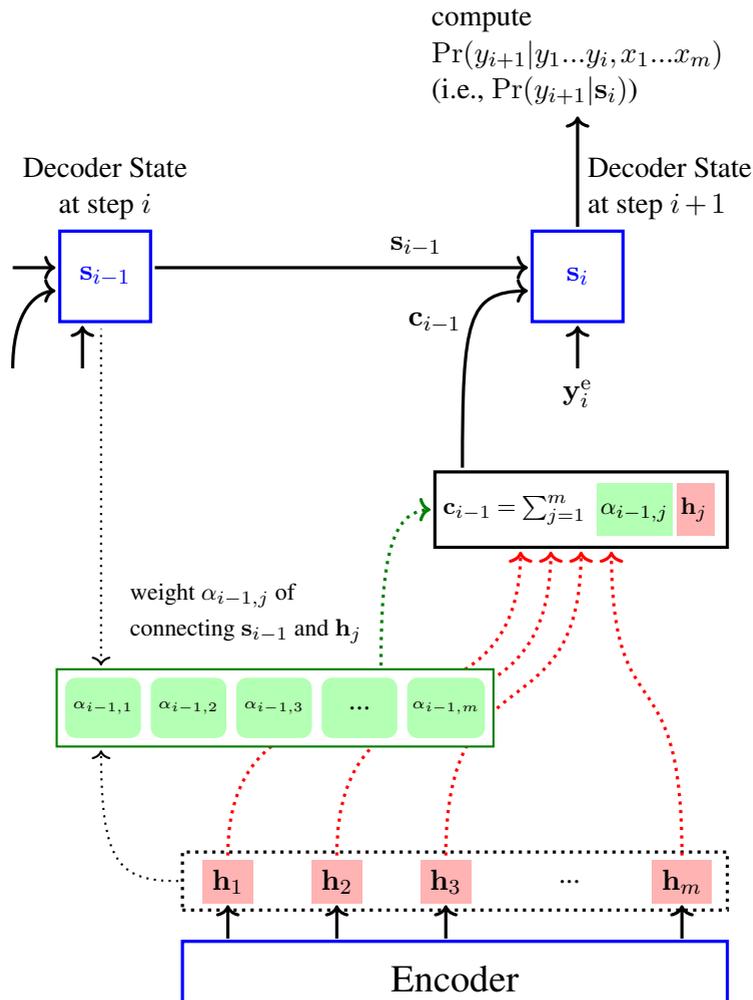


Figure 5.4: An attention model for NMT. Suppose we have obtained the representations  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  and the decoder state  $\mathbf{s}_{i-1}$  up to this point. We wish to obtain the decoder state at the next step. To this end, we first compute attention weights by normalizing some attention scores between  $\mathbf{s}_{i-1}$  and  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ , and then compute a context vector  $\mathbf{c}_{i-1}$  by summing over  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  with the attention weights. A new decoder state  $\mathbf{s}_i$  is created by taking the context vector  $\mathbf{c}_{i-1}$ , the previous state  $\mathbf{s}_{i-1}$ , and the word representation  $\mathbf{y}_i^e$ .  $\mathbf{s}_i$  will be used as a condition for predicting a distribution of words at step  $i + 1$ .

where  $\Pr(\cdot | \mathbf{s}_i)$  is generally a Softmax layer. This process is illustrated in Figure 5.4.

We now have a model for computing  $\Pr(y_{i+1} | y_1 \dots y_i, x_1 \dots x_m)$ . A brief outline of the key steps of this model is given by

1. Encode the source-side sequence as  $\mathbf{h}_1 \dots \mathbf{h}_m$  where  $\mathbf{h}_j = \text{RNN}(\mathbf{h}_{j-1}, \mathbf{x}_j^e)$ .
2. Repeat the following procedure from  $i = 1$  to  $n - 1$ .

- a. Compute the alignment score  $a(\mathbf{s}_{i-1}, \mathbf{h}_j)$  for each  $j$ .
- b. Compute the attention weights  $\{\alpha_{i-1,1}, \dots, \alpha_{i-1,m}\}$  where  $\alpha_{i-1,j} = \frac{\exp(a(\mathbf{s}_{i-1}, \mathbf{h}_j))}{\sum_{j'=1}^m \exp(a(\mathbf{s}_{i-1}, \mathbf{h}_{j'}))}$ .
- c. Compute the context vector  $\mathbf{c}_{i-1} = \sum_{j=1}^m \alpha_{i-1,j} \cdot \mathbf{h}_j$ .
- d. Compute the target-side state  $\mathbf{s}_i = \text{RNN}(\mathbf{s}_{i-1}, \mathbf{c}_{i-1}, \mathbf{y}_i^e)$ .
- e. Compute the distribution of target-side words  $\Pr(\cdot | \mathbf{s}_i)$ .
- f. Compute  $\Pr(y_{i+1} | y_1 \dots y_i, x_1 \dots x_m) = \Pr(y_{i+1} | \mathbf{s}_i)$  for a given word  $y_{i+1}$  (as in training), or select the most likely word  $\hat{y}_{i+1} = \arg \max_{y_{i+1}} \Pr(y_{i+1} | y_1 \dots y_i, x_1 \dots x_m)$  (as in testing).

In real-world systems, this basic model can be modified to better predict the target-side words. For example, we can introduce fusion layers to combine  $\mathbf{s}_i$ ,  $\mathbf{c}_{i-1}$ , and  $\mathbf{y}_i^e$  before the Softmax layer so that we have a deeper model for prediction [Bahdanau et al., 2014]. Another commonly used approach is to stack multiple RNN layers on the target-side. In this case, one can perform attention in either each layer of the stack [Wu et al., 2016] or the top-most layer of the stack [Luong et al., 2015]. See Section 5.3.5 for more information about multi-layer approaches to attention.

### 5.3.2 The QKV Attention

Because the attention mechanism is such a powerful approach, many variants have been developed. Perhaps the most widely used approach is to reframe the attention problem as one of matching a query in a set of key-value pairs. It lays the foundation for the well-known sequence model — Transformer [Vaswani et al., 2017].

Here we assume that there are a number of key-value pairs  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)\}$  and a query  $\mathbf{q}$ . The goal of the **query-key-value attention** (or **QKV attention**) model is to obtain a value by considering the correspondence between the query and the keys. This is a standard searching problem in database systems in which information is returned in its original form or a new form when it matches the query. In the QKV attention, the result of searching is not a single value in  $\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$  but instead a combination of these values. This is the key difference of this attention model compared with the conventional models of searching.

Formally, the result of the QKV attention is defined to be

$$\mathbf{c} = \sum_{j=1}^m \alpha_j \mathbf{v}_j \quad (5.27)$$

where

$$\alpha_j = \text{Softmax}\left(\frac{\mathbf{q}\mathbf{k}_j^T}{\beta}\right) \quad (5.28)$$

is the attention weight. It turns out that the above model has precisely the same general form as the model described in the previous subsection, and  $\mathbf{c}$  can be simply viewed as a context

vector.

While the basic form of the QKV attention is not something “new”, it can handle a variety of problems by giving  $\mathbf{q}$ ,  $\mathbf{k}_j$  and  $\mathbf{v}_j$  appropriate meanings. Here we consider a more general case where there are  $n$  queries  $\{\mathbf{q}_1, \dots, \mathbf{q}_n\}$  and  $n$  output vectors  $\{\mathbf{c}_1, \dots, \mathbf{c}_n\}$ . To simplify notation, we use  $\mathbf{Q}$  to denote a matrix where the  $i$ -th row vector is  $\mathbf{q}_i$ , like this

$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_n \end{bmatrix} \quad (5.29)$$

Likewise, we can define  $\mathbf{K} = \begin{bmatrix} \mathbf{k}_1 \\ \vdots \\ \mathbf{k}_m \end{bmatrix}$ ,  $\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_m \end{bmatrix}$ , and  $\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_n \end{bmatrix}$ . Then, the attention model can be formulated as

$$\mathbf{C} = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\beta}\right)\mathbf{V} \quad (5.30)$$

Figure 5.5 shows an illustration of this equation. Note that  $\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\beta}\right)$  computes a matrix of attention weights

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\beta}\right) = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & & \vdots \\ \alpha_{n,1} & \dots & \alpha_{n,m} \end{bmatrix} \quad (5.31)$$

where a row vector  $[\alpha_{i,1} \dots \alpha_{i,m}]$  represents a distribution over  $\{\mathbf{v}_1, \dots, \mathbf{v}_m\}$ . We can then expand Eq. (5.30) for easy understanding of the model

$$\begin{aligned} \mathbf{C} &= \begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_n \end{bmatrix} \\ &= \begin{bmatrix} \sum_{j=1}^m \alpha_{1,j} \mathbf{v}_j \\ \vdots \\ \sum_{j=1}^m \alpha_{n,j} \mathbf{v}_j \end{bmatrix} \\ &= \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & & \vdots \\ \alpha_{n,1} & \dots & \alpha_{n,m} \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_m \end{bmatrix} \end{aligned} \quad (5.32)$$

In sequence-to-sequence modeling,  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$  can be defined in several different ways. To describe the correspondence between the source-side and target-side sequences, one

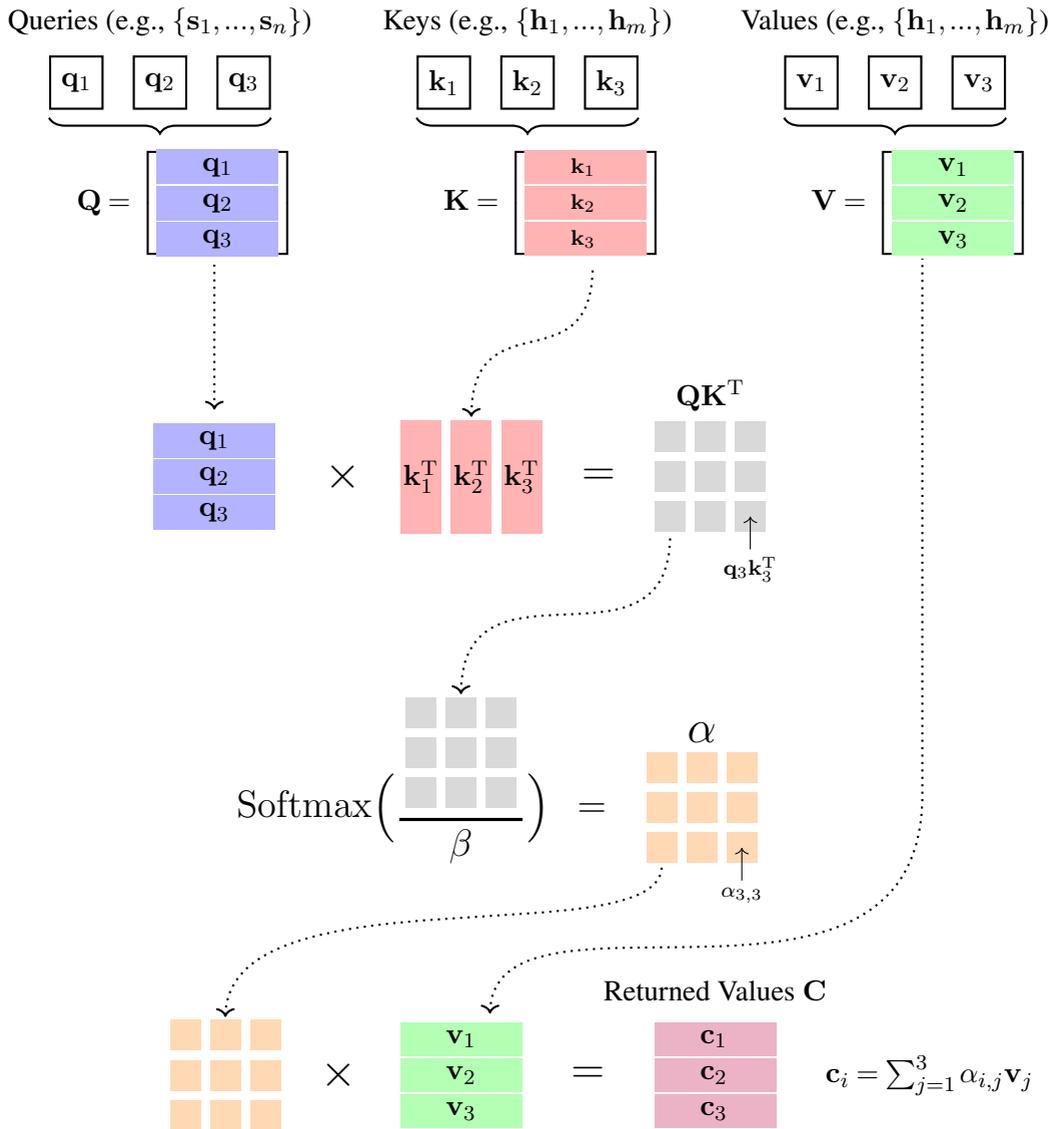


Figure 5.5: The QKV attention model for batches of queries ( $\mathbf{Q}$ ), keys ( $\mathbf{K}$ ), and values ( $\mathbf{V}$ ). The figure shows a direct implementation of the formula  $\mathbf{C} = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\beta}\right)\mathbf{V}$ .  $\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\beta}\right)$  computes the attention weights by normalizing a scaled dot-product of  $\mathbf{Q}$  and  $\mathbf{K}^T$ . This results in a matrix  $\alpha$  in which a row vector describes weights of different values. By multiplying  $\alpha$  with  $\mathbf{V}$ , we obtain a sequence of new values, each expressing a weighted sum of the original values.

approach, called **encoder-decoder attention**, is to simply assume that

$$\mathbf{Q} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} \quad (5.33)$$

and

$$\mathbf{K} = \mathbf{V} = \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \quad (5.34)$$

In this case,  $\mathbf{C}$  is a sequence of new representations of the source-side sequence given the representations of the target-side sequence. As with the model described in the previous subsection, each  $\mathbf{c}_i \in \mathbf{C}$  can be used to predict the word  $y_{i+1}$ .

In addition to applying the model to sequence-to-sequence problems, another type of approach is to regard it as a sequence model, that is, we use the QKV attention to represent a sequence in one language. In this case, the QKV attention is also called **self-attention** which forms the basis of the well-known Transformer model [Vaswani et al., 2017]. Consider, for example, the sequence of states  $\mathbf{h}_1 \dots \mathbf{h}_m$ . The self-attention model assumes that

$$\mathbf{Q} = \mathbf{K} = \mathbf{V} = \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \quad (5.35)$$

Then, the output of the model is a sequence of representations  $\mathbf{c}_1 \dots \mathbf{c}_m$ .  $\mathbf{c}_j$  is a representation which considers the correlations between  $\mathbf{h}_j$  and any other element of the input sequence. We will see a more detailed discussion on this model in Chapter 6.

### 5.3.3 Multi-head Attention

**Multi-head attention** is an interesting extension to the above models. The key idea is to perform attention in different sub-spaces of representations simultaneously rather than in a single space of representations. To illustrate, consider a standard attention model that takes sequences of source-side and target-side states and outputs a sequence of new states, written as

$$\mathbf{c}_1 \dots \mathbf{c}_n = \text{Att}(\mathbf{h}_1 \dots \mathbf{h}_m, \mathbf{s}_1 \dots \mathbf{s}_n) \quad (5.36)$$

where  $\mathbf{h}_j, \mathbf{s}_i, \mathbf{c}_i \in \mathbb{R}^{d_h}$ , and  $\text{Att}(\cdot)$  is the attention function. We can map  $\mathbf{h}_j$  into  $\tau$  vectors  $\{\mathbf{h}_j^{[1]}, \dots, \mathbf{h}_j^{[\tau]}\}$  via the following linear transformations

$$\mathbf{h}_j^{[1]} = \mathbf{h}_j \mathbf{W}_h^{[1]} \quad (5.37)$$

$$\vdots$$

$$\mathbf{h}_j^{[\tau]} = \mathbf{h}_j \mathbf{W}_h^{[\tau]} \quad (5.38)$$

where  $\mathbf{h}_j^{[1]}, \dots, \mathbf{h}_j^{[\tau]} \in \mathbb{R}^{\frac{d_h}{\tau}}$ , and  $\mathbf{W}_h^{[1]}, \dots, \mathbf{W}_h^{[\tau]} \in \mathbb{R}^{d_h \times \frac{d_h}{\tau}}$ .

Similarly, we can map  $\mathbf{s}_i$  into  $\tau$  vectors  $\{\mathbf{s}_i^{[1]}, \dots, \mathbf{s}_i^{[\tau]}\}$ . We then define  $\tau$  **feature sub-spaces** in which the attention function is performed independently. For the  $k$ -th feature sub-space, we

have

$$\mathbf{c}_1^{[k]} \dots \mathbf{c}_n^{[k]} = \text{Att}(\mathbf{h}_1^{[k]} \dots \mathbf{h}_m^{[k]}, \mathbf{s}_1^{[k]} \dots \mathbf{s}_n^{[k]}) \quad (5.39)$$

The output of the model is a sequence of  $d_h$ -dimensional vectors, each of which is obtained by concatenating the vectors that are produced in all these feature sub-spaces, followed by a linear transformation. This procedure is given by

$$\mathbf{c}_1 = [\mathbf{c}_1^{[1]}, \dots, \mathbf{c}_1^{[\tau]}] \mathbf{W}_c \quad (5.40)$$

...

$$\mathbf{c}_n = [\mathbf{c}_n^{[1]}, \dots, \mathbf{c}_n^{[\tau]}] \mathbf{W}_c \quad (5.41)$$

where  $\mathbf{W}_c \in \mathbb{R}^{d_h \times d_h}$ .

Following the notation used in the previous subsection, we can express a sequence of vectors as a matrix, say,  $\mathbf{H} = \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \in \mathbb{R}^{m \times d_h}$ ,  $\mathbf{S} = \begin{bmatrix} \mathbf{s}_1 \\ \vdots \\ \mathbf{s}_n \end{bmatrix} \in \mathbb{R}^{n \times d_h}$ , and  $\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_n \end{bmatrix} \in \mathbb{R}^{n \times d_h}$ .

Using this notation, we rewrite Eq. (5.36) as

$$\mathbf{C} = \text{Att}(\mathbf{H}, \mathbf{S}) \quad (5.42)$$

To give a formal definition of multi-head attention, we first introduce the split and merge functions. The split function divides each row vector of a matrix into a number of sub-vectors, resulting in a 3D tensor. For example, splitting a  $m \times d_h$  matrix  $\mathbf{A}$  with  $\tau$  produces a  $\tau \times m \times \frac{d_h}{\tau}$  tensor<sup>3</sup>

$$\mathbf{A}_{\text{heads}} = \text{Split}(\mathbf{A}, \tau) \quad (5.43)$$

The merge function has a reverse form of the split function. Given a  $\tau \times n \times \frac{d_h}{\tau}$  tensor (say  $\mathbf{A}_{\text{heads}}$ ), it merges each group of  $\tau \frac{d_h}{\tau}$ -dimensional sub-arrays in the form

$$\mathbf{A}_{\text{merge}} = \text{Merge}(\mathbf{A}_{\text{heads}}, \tau) \quad (5.44)$$

Thus the form of multi-head attention is given by

$$\begin{aligned} \mathbf{C} &= \mathbf{C}_{\text{merge}} \mathbf{W}_c \\ &= \text{Merge}(\mathbf{C}_{\text{heads}}, \tau) \mathbf{W}_c \\ &= \text{Merge}(\text{Att}(\mathbf{H}_{\text{heads}}, \mathbf{S}_{\text{heads}}), \tau) \mathbf{W}_c \end{aligned} \quad (5.45)$$

$$\mathbf{H}_{\text{heads}} = \text{Split}(\mathbf{H} \mathbf{W}_h, \tau) \quad (5.46)$$

$$\mathbf{S}_{\text{heads}} = \text{Split}(\mathbf{S} \mathbf{W}_s, \tau) \quad (5.47)$$

<sup>3</sup>A  $a \times b \times c$  tensor can be treated as an array of  $a$  matrices whose shapes are  $b \times c$ .

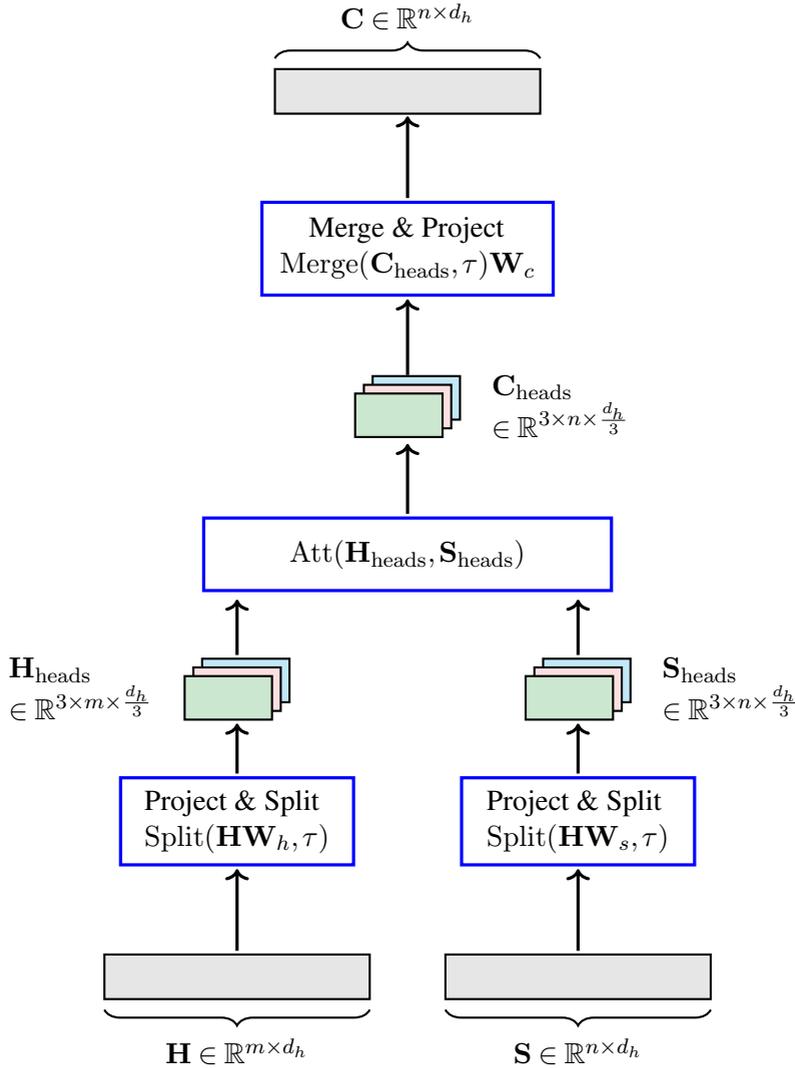


Figure 5.6: An attention model with  $\tau = 3$  heads. First, we transform the input matrices into multi-head representations, i.e., 3D tensors  $\mathbf{H}_{\text{heads}} \in \mathbb{R}^{3 \times m \times \frac{d_h}{3}}$  and  $\mathbf{S}_{\text{heads}} \in \mathbb{R}^{3 \times n \times \frac{d_h}{3}}$ . These tensors are then taken by an attention model. The output of this model is a tensor  $\mathbf{C}_{\text{heads}} \in \mathbb{R}^{3 \times n \times \frac{d_h}{3}}$ . We then merge the heads of  $\mathbf{C}_{\text{heads}}$ , followed by a linear transformation. Finally, we obtain  $n$  vectors of size  $d_h$ , represented by an  $n \times d_h$  matrix.

where  $\mathbf{W}_h, \mathbf{W}_s \in \mathbb{R}^{d_h \times d_h}$  are the parameters.  $\text{Split}(\mathbf{H}\mathbf{W}_h, \tau)$  implements the projections of Eqs. (5.37-5.38) for all  $\mathbf{h}_j$ . Likewise, we can have the meaning of  $\text{Split}(\mathbf{H}\mathbf{W}_s, \tau)$ . Note that here  $\text{Att}(\cdot)$  is extended to deal with multi-head inputs. See Figure 5.6 for an illustration of this model.

Multi-head attention is a very general approach that can be extended to many models. As a simple example of this extension, consider the QKV attention model discussed in the previous subsection. Let  $\text{Att}_{\text{QKV}}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$  be the attention function, and  $\mathbf{Q} \in \mathbb{R}^{d_k}$ ,  $\mathbf{K} \in \mathbb{R}^{d_k}$ ,  $\mathbf{V} \in \mathbb{R}^{d_v}$

be the inputs. The multi-head QKV attention model is given by

$$\mathbf{C} = \text{Merge}(\text{Att}_{\text{QKV}}(\mathbf{Q}_{\text{heads}}, \mathbf{K}_{\text{heads}}, \mathbf{V}_{\text{heads}}))\mathbf{W}_c \quad (5.48)$$

$$\mathbf{Q}_{\text{heads}} = \text{Split}(\mathbf{Q}\mathbf{W}_q, \tau) \quad (5.49)$$

$$\mathbf{K}_{\text{heads}} = \text{Split}(\mathbf{K}\mathbf{W}_k, \tau) \quad (5.50)$$

$$\mathbf{V}_{\text{heads}} = \text{Split}(\mathbf{V}\mathbf{W}_v, \tau) \quad (5.51)$$

where  $\mathbf{W}_q \in \mathbb{R}^{d_k \times d_k}$ ,  $\mathbf{W}_k \in \mathbb{R}^{d_k \times d_k}$ ,  $\mathbf{W}_v \in \mathbb{R}^{d_v \times d_v}$ ,  $\mathbf{W}_c \in \mathbb{R}^{d_v \times d_v}$  are the model parameters.

One advantage of multi-head attention is that the feature sub-spaces will each describe a different perspective of attention (call it an **attention head** or **head** for short). Therefore, the concatenation of the outputs over these heads represents an ensemble of attention models that deal with different parts of the data. This is similar to learning a group of models independently and combining them to form a stronger model. This type of machine learning approach has been proven to be useful in many problems [Opitz and Maclin, 1999; Zhou, 2012b]. Note that the multi-head attention models discussed here are parameterized by the linear projections on the input and output spaces. The use of these linear projections is generally helpful as the models become deeper and can describe more complex problems.

From an architecture design perspective, multi-head attention falls into a broad class of neural networks — those involving a number of branches of layer stacks for dealing with the same input (call them **multi-branch neural networks**). However, unlike conventional approaches, which require different model architectures for different branches, the multi-head attention approach is based on a single model for all the heads. As a result, such systems are very efficient in practice because the attention procedure can run in parallel over these heads.

### 5.3.4 Hierarchical Attention

In many cases the underlying structure of an NLP problem is hierarchical. For example, documents may have a multi-level structure: a document is made up of sentences, a sentence is made up of words, and a word is made up of characters. It is therefore desirable to modify the attention models to take into account the hierarchical nature of this data [Yang et al., 2016].

To illustrate, we consider a simple problem where the source-side has a 2-level tree structure. Suppose the source-side sequence is a concatenation of a number of sub-sequences  $\{\bar{\mathbf{u}}_1, \dots, \bar{\mathbf{u}}_T\}$ . Each  $\bar{\mathbf{u}}_t$  yields a sequence of words

$$\bar{\mathbf{u}}_t = x_{p(t,1)} \dots x_{p(t,|\bar{\mathbf{u}}_t|)} \quad (5.52)$$

where  $p(t, i)$  is the position of the  $i$ -th word of  $\bar{\mathbf{u}}_t$  in the entire source-side sequence  $x_1 \dots x_m$ . Then, the sequence  $x_1 \dots x_m$  can be written as a composition of  $T$  sub-sequences:

$$x_1 \dots x_m = \underbrace{x_{p(1,1)} \dots x_{p(1,|\bar{\mathbf{u}}_1|)}}_{\bar{\mathbf{u}}_1} \underbrace{x_{p(2,1)} \dots x_{p(2,|\bar{\mathbf{u}}_2|)}}_{\bar{\mathbf{u}}_2} \dots \underbrace{x_{p(T,1)} \dots x_{p(T,|\bar{\mathbf{u}}_T|)}}_{\bar{\mathbf{u}}_T} \quad (5.53)$$

Similarly, the encoder output  $\mathbf{h}_1 \dots \mathbf{h}_m$  can be written as

$$\mathbf{h}_1 \dots \mathbf{h}_m = \mathbf{h}_{p(1,1)} \dots \mathbf{h}_{p(1,|\bar{\mathbf{u}}_1|)} \mathbf{h}_{p(2,1)} \dots \mathbf{h}_{p(2,|\bar{\mathbf{u}}_2|)} \dots \mathbf{h}_{p(T,1)} \dots \mathbf{h}_{p(T,|\bar{\mathbf{u}}_T|)} \quad (5.54)$$

On the target-side, we assume that there are two sequences of state vectors: one for placing the standard representations of the target-side sequence (i.e.,  $\mathbf{s}_1 \dots \mathbf{s}_n$ ) and one for placing higher-level representations of  $\mathbf{s}_1 \dots \mathbf{s}_n$ . Let  $\phi(i)$  denote the position in the higher-level sequence of  $\mathbf{s}_i$ , and  $\bar{\mathbf{s}}_{\phi(i)}$  denote the corresponding state vector. For each  $i$ , we thus have a pair of state vectors  $\mathbf{s}_i$  and  $\bar{\mathbf{s}}_{\phi(i)}$ . In general, the relationship between  $\mathbf{s}_i$  and  $\bar{\mathbf{s}}_{\phi(i)}$  comes from the hierarchical structure of the problem. For example,  $\mathbf{s}_i$  is the representation of a word, and  $\bar{\mathbf{s}}_{\phi(i)}$  is the representation of the sentence the word belongs to<sup>4</sup>.

As before, our goal is to obtain a context vector  $\mathbf{c}_i$  for each target-side position  $i$ . Here we still take  $\mathbf{c}_i$  to be a weighted sum of  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ , as in Eq. (5.16). All that remains is to specify the attention weight for each  $\mathbf{h}_j$ . As a first step we attend  $\mathbf{s}_i$  to each  $\mathbf{u}_t$ . This is a standard procedure. We just need to run the attention model on  $\mathbf{h}_{p(t,1)} \dots \mathbf{h}_{p(t,|\bar{\mathbf{u}}_t|)}$  instead of  $\mathbf{h}_1 \dots \mathbf{h}_m$ , given by

$$\begin{aligned} \bar{\mathbf{h}}_t &= \text{Att}(\mathbf{h}_{p(t,1)} \dots \mathbf{h}_{p(t,|\bar{\mathbf{u}}_t|)}, \mathbf{s}_i) \\ &= \sum_{k=1}^{|\bar{\mathbf{u}}_t|} \pi_{i,k,t} \mathbf{h}_{p(t,k)} \end{aligned} \quad (5.55)$$

where  $\pi_{i,k,t}$  is the attention weight restricted to  $\mathbf{u}_t$ .  $\bar{\mathbf{h}}_t$  is a representation of  $\mathbf{u}_t$ , and so we have a new sequence of representations  $\bar{\mathbf{h}}_1 \dots \bar{\mathbf{h}}_T$ .

Then, we run the attention model on  $\bar{\mathbf{h}}_1 \dots \bar{\mathbf{h}}_T$  to perform a second round of attention. This is done by attending  $\mathbf{s}_{\phi(i)}$  to  $\bar{\mathbf{h}}_1 \dots \bar{\mathbf{h}}_T$ . The output is a context vector for the hierarchical attention model, given by

$$\begin{aligned} \mathbf{c}_i &= \text{Att}(\bar{\mathbf{h}}_1 \dots \bar{\mathbf{h}}_T, \mathbf{s}_{\phi(i)}) \\ &= \sum_{t=1}^T \gamma_{i,t} \bar{\mathbf{h}}_t \end{aligned} \quad (5.56)$$

where  $\gamma_{i,t}$  is the weight of attending  $\mathbf{s}_{\phi(i)}$  to  $\bar{\mathbf{h}}_t$ . Substituting Eq. (5.55) into Eq. (5.56), we can write  $\mathbf{c}_i$  as

$$\begin{aligned} \mathbf{c}_i &= \sum_{t=1}^T \sum_{k=1}^{|\bar{\mathbf{u}}_t|} \gamma_{i,t} \pi_{i,k,t} \mathbf{h}_{p(t,k)} \\ &= \sum_{j=1}^m \alpha_{i,j} \mathbf{h}_j \end{aligned} \quad (5.57)$$

While the notation in this subsection is a bit complicated, the form of the resulting model

<sup>4</sup>If the  $a$ -th sentence covers words from position  $b$  to  $c$ , then  $\phi(b) = \phi(b+1) = \dots = \phi(c) = a$ .

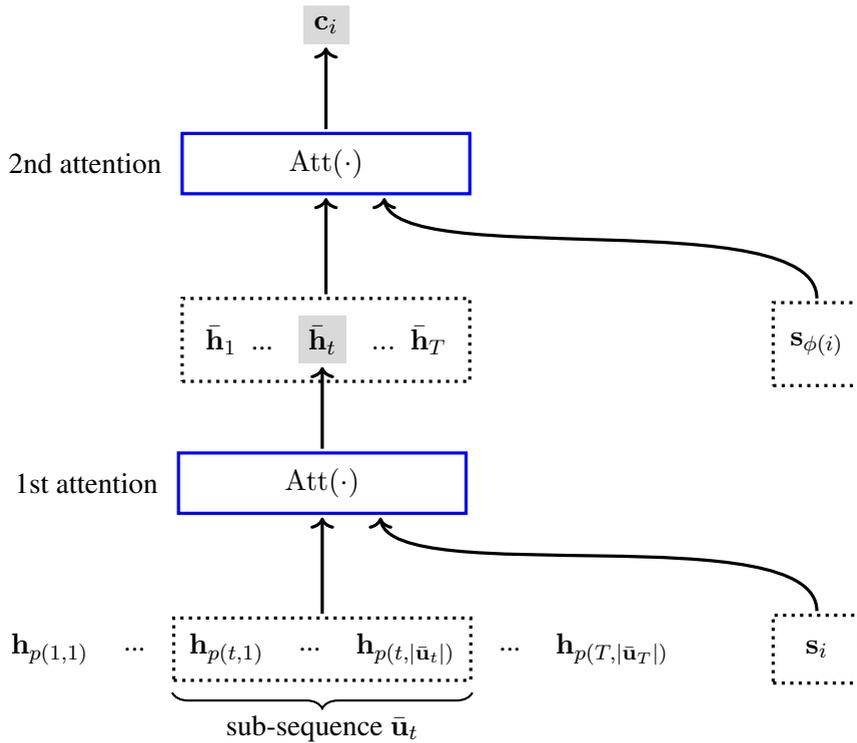


Figure 5.7: A 2-level hierarchical attention model. The input sequence  $\mathbf{h}_1 \dots \mathbf{h}_m$  is made up of  $T$  sub-sequences. For each sub-sequence  $\bar{\mathbf{u}}_t$ , an attention model is used to produce a context vector  $\bar{\mathbf{h}}_t$  by considering the target-side state (i.e.,  $\mathbf{s}_i$ ) and the representations of the sub-sequence (i.e.,  $\mathbf{h}_{p(t,1)} \dots \mathbf{h}_{p(t,|\bar{\mathbf{u}}_t|)}$ ). The result of running this procedure on the  $T$  sub-sequences is  $T$  level-1 representations  $\bar{\mathbf{h}}_1 \dots \bar{\mathbf{h}}_T$ . They are then taken by a second attention model to consider the attention between these representations and a higher-level target-side state  $\mathbf{s}_{\phi(i)}$ . This results in the context vector  $\mathbf{c}_i$  which describes the attention between the target-side state  $\mathbf{s}_i$  and the entire source-side sequence  $\mathbf{h}_1 \dots \mathbf{h}_m$ .

is simple. We still combine  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  in a linear manner but with new weights [Maruf et al., 2019]. Computing  $\alpha_{i,j}$  describes a generative process in which we first determine the weight of each sub-sequence and then determine the weight of each word in a sub-sequence, as illustrated in Figure 5.7. See below for an alignment among different types of attention weight.

sequence	$\mathbf{h}_1$	$\dots$	$\mathbf{h}_{ \mathbf{u}_1 }$	$\mathbf{h}_{ \mathbf{u}_1 +1}$	$\dots$	$\mathbf{h}_{ \mathbf{u}_1 + \mathbf{u}_2 }$	$\dots$	$\mathbf{h}_{\sum_{t=1}^{T-1}  \mathbf{u}_t +1}$	$\dots$	$\mathbf{h}_m$
weight ( $\alpha$ )	$\alpha_{i,1}$	$\dots$	$\alpha_{i, \mathbf{u}_1 }$	$\alpha_{i, \mathbf{u}_1 +1}$	$\dots$	$\alpha_{i, \mathbf{u}_1 + \mathbf{u}_2 }$	$\dots$	$\alpha_{i,\sum_{t=1}^{T-1}  \mathbf{u}_t +1}$	$\dots$	$\alpha_{i,m}$
sequence	$\mathbf{h}_{p(1,1)}$	$\dots$	$\mathbf{h}_{p(1, \bar{\mathbf{u}}_1 )}$	$\mathbf{h}_{p(2,1)}$	$\dots$	$\mathbf{h}_{p(2, \bar{\mathbf{u}}_2 )}$	$\dots$	$\mathbf{h}_{p(T,1)}$	$\dots$	$\mathbf{h}_{p(T, \bar{\mathbf{u}}_T )}$
weight ( $\gamma$ )	$\gamma_{i,1}$	$\dots$	$\gamma_{i,1}$	$\gamma_{i,2}$	$\dots$	$\gamma_{i,2}$	$\dots$	$\gamma_{i,T}$	$\dots$	$\gamma_{i,T}$
weight ( $\pi$ )	$\pi_{i,1,1}$	$\dots$	$\pi_{i, \bar{\mathbf{u}}_1 ,1}$	$\pi_{i,1,2}$	$\dots$	$\pi_{i, \bar{\mathbf{u}}_2 ,2}$	$\dots$	$\pi_{i,1,T}$	$\dots$	$\pi_{i, \bar{\mathbf{u}}_T ,T}$

### 5.3.5 Multi-layer Attention

So far we have considered the case of **single-layer attention** — the output of the attention models is written as a linear combination of the source-side representations. Now we extend it in a natural way to **multi-layer attention** in which the single-layer attention procedure runs a number of times for forming a “deeper” attention model.

To do this, a multi-layer neural network is created on the target-side. The model architecture is regular. We stack a number of attention layers, each interacting with the source-side sequence and feeding its output to the next layer. In an attention layer, we perform attention as usual. For the  $l$ -th layer in the stack, this step takes the source-side sequence (denoted by  $\mathbf{h}_1 \dots \mathbf{h}_m$ ) as well as the output of the previous layer (denoted by  $\mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1}$ ), and produces a sequence of vectors by

$$\mathbf{c}_1^l \dots \mathbf{c}_n^l = \text{Att}(\mathbf{h}_1 \dots \mathbf{h}_m, \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1}) \quad (5.58)$$

where  $\text{Att}(\cdot)$  could be any attention function described in this chapter.

Then, we create another neural network  $f(\cdot)$  to give more modeling power to the model. The output of the attention layer is thus defined to be

$$\mathbf{s}_1^l \dots \mathbf{s}_n^l = f(\mathbf{c}_1^l \dots \mathbf{c}_n^l, \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1}) \quad (5.59)$$

$f(\cdot)$  can be designed in many ways [Sukhbaatar et al., 2015; Wu et al., 2016; Vaswani et al., 2017]. A popular choice is to define  $f(\cdot)$  as a feed-forward neural network with a residual connection, given by

$$f(\mathbf{c}_1^l \dots \mathbf{c}_n^l, \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1}) = \text{FFN}(\mathbf{c}_1^l \dots \mathbf{c}_n^l) + \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1} \quad (5.60)$$

Substituting for the vectors  $\mathbf{c}_1^l \dots \mathbf{c}_n^l$ , using Eq. (5.58), the output of layer  $i$  is written in the form

$$\mathbf{s}_1^l \dots \mathbf{s}_n^l = \text{FFN}(\text{Att}(\mathbf{h}_1 \dots \mathbf{h}_m, \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1})) + \mathbf{s}_1^{l-1} \dots \mathbf{s}_n^{l-1} \quad (5.61)$$

As with the models in the previous subsections, it is convenient to use a more compact notation by expressing a sequence of vectors as a matrix. Thus this model can be given in another form

$$\mathbf{S}^l = \text{FFN}(\text{Att}(\mathbf{H}, \mathbf{S}^{l-1})) + \mathbf{S}^{l-1} \quad (5.62)$$

Here  $\text{FFN}(\cdot)$  is generally a multi-layer neural network with non-linear activation functions. The identity mapping (i.e.,  $+\mathbf{S}^{l-1}$ ) creates a direct path from the input to the output of the layer, making it easier to train a deep neural network.

Figure 5.8 shows the architecture of this model. The attention model starts with the initial

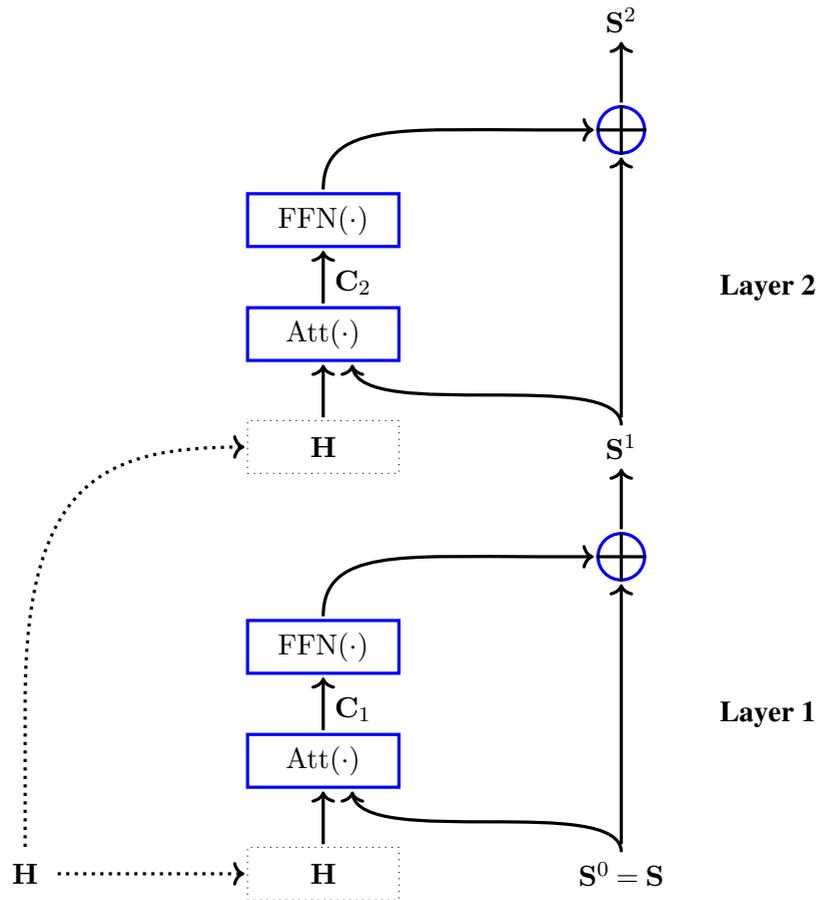


Figure 5.8: A 2-layer attention model. These layers take the same “key-value” pairs (i.e.,  $\mathbf{H}$ ) but each takes a different “query” (i.e.,  $\mathbf{S}^l$ ). The attention model is standard: context vectors  $\mathbf{C}^l$  are generated by taking both  $\mathbf{H}$  and  $\mathbf{S}^l$ . A feed-forward neural network is built to transform  $\mathbf{C}^l$ , followed by an addition of  $\mathbf{S}^l$ . So this model can be formulated as  $\mathbf{S}^l = \text{FFN}(\text{Att}(\mathbf{H}, \mathbf{S}^{l-1})) + \mathbf{S}^{l-1}$ .  $\mathbf{S}^l$  is then used in the next layer as the query, that is, layer  $l + 1$  takes  $\mathbf{H}$  and  $\mathbf{S}^l$ , and repeats the above process. The output of the last layer can be viewed as a deeper representation of  $\mathbf{H}$  for  $\mathbf{S}$ .

representation of the target-side sequence, that is,  $\mathbf{S}^0 = \mathbf{S} = \begin{bmatrix} \mathbf{s}_1 \\ \vdots \\ \mathbf{s}_n \end{bmatrix}$ . If there are  $L$  attention

layers, then the final output will be  $\mathbf{S}^L$ .

### 5.3.6 Remarks

Above we considered a basic attention model and a series of refinements. The literature on attention and related topics contains a wide range of methods for modeling how a system concentrates on different parts of the input, as well as a wide range of applications of such

models. This subsection provides discussions on some of the interesting issues.

## 1. Alignment vs Attention

Attention is related to a long line of research on alignment approaches to modeling the correspondence between two groups of language units. In NLP, alignment is a very general concept that is used to refer to several problems. For example, most statistical machine translation systems are trained on bilingual texts which are annotated with word-to-word alignment [Koehn et al., 2003; Chiang, 2005]. **Word alignment** models are thus developed to generate links between words in two sentences [Vogel et al., 1996; Och and Ney, 2003; Taskar et al., 2005; Dyer et al., 2013]. While the outputs of these systems are discrete variables, the underlying models are mostly probabilistic and continuous. Therefore, the correspondence between word alignment and the attention models discussed here is apparent because they are both learned to assign a weight to each pair of words.

Note that despite the similarity between alignment and attention problems, their goals are different. In most cases word alignment models are used as individual systems to produce alignment results for downstream systems, whereas attention models are typically treated as components of bigger systems and do not work alone (see Figure 5.9 for a comparison of these models). This makes them fit different types of sequence-to-sequence systems in practice: word alignment is one step of a pipelined system, and attention is some intermediate states of a neural network.

Nevertheless, word alignment and attention are two related problems, and can help each other in some cases. For example, one way to see how an attention model behaves is to induce word alignments from it and measure the quality of these word alignments [Tu et al., 2016; Li et al., 2019; Garg et al., 2019]. Also, systems equipped with the attention mechanism can be guided by external word alignment resources [Mi et al., 2016b; Liu et al., 2016b].

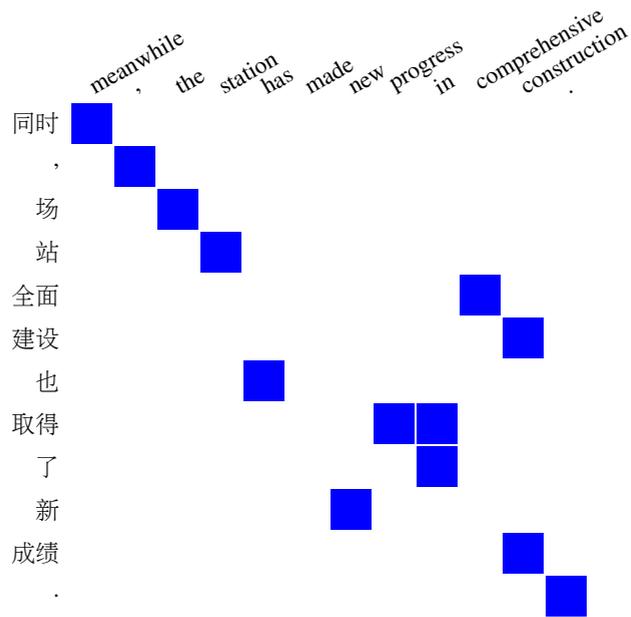
## 2. Introducing Priors

As discussed in Section 5.3.1, the attention models implicitly define an attention distribution over  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$  by which we can compute a weighted sum of these representations. This distribution is expressed in terms of the alignment weights and is learned as part of a model. In addition to learning the attention distribution in an end-to-end fashion, we can also define it based on our knowledge about how we concentrate on different parts of a sequence when reading it.

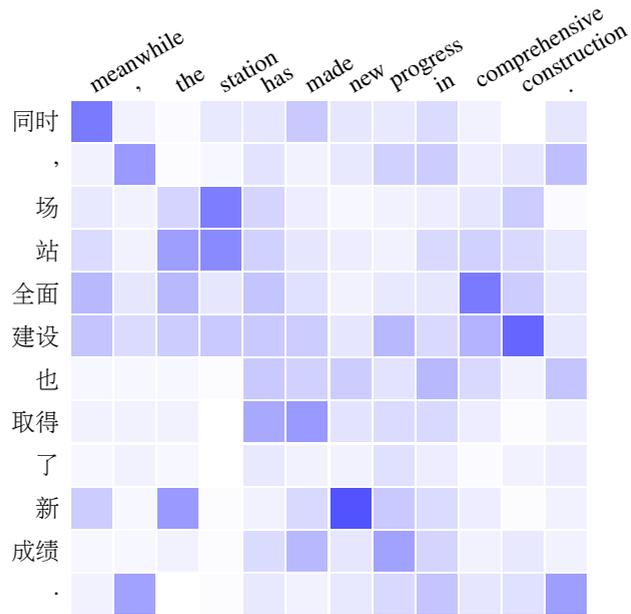
One approach is to directly impose some structure on the distribution. A simple example is that we consider only a span of the sequence for attention and discard the rest. Let  $[\rho_i - D, \rho_i + D]$  be a  $2D + 1$  word window centered at position  $\rho_i$  of the source-side sequence. We can connect  $\mathbf{s}_i$  only to  $\mathbf{h}_{\rho_i - D} \dots \mathbf{h}_{\rho_i + D}$  and obtain a local context vector in the following form

$$\mathbf{c}_i = \text{Att}(\mathbf{h}_{\rho_i - D} \dots \mathbf{h}_{\rho_i + D}, \mathbf{s}_i) \quad (5.63)$$

This approach is also called **local attention**. The problem of determining  $\rho_i$  is similar to the **reordering problem** in machine translation. For translation between languages with



(a) A heat map of word alignments



(b) A heat map of attention weights

Figure 5.9: Heat maps of a word alignment model and an attention model for a pair of Chinese and English sentences. The heat maps are represented as shaded grids in which each cell describes the correspondence of a pair of Chinese and English words. This correspondence is shown on a color scale ranging from white denoting a weight of 0 to pure blue denoting a weight of 1.

similar word orders, it is common to assume that the translation is monotonic and  $\rho_i$  is linear with respect to  $i$  [Koehn, 2004], e.g.,  $\rho_i = \lfloor m \frac{i}{n} \rfloor$  or  $\lceil m \frac{i}{n} \rceil$ . An alternative method is to use a neural network to predict a relative position in the source-side sequence (denoted by  $r_i$ ) [Luong et al., 2015].  $\rho_i$  can then be defined to be  $\lfloor mr_i \rfloor$  or  $\lceil mr_i \rceil$ .

In another thread of research, a new distribution is derived by combining the original attention distribution and some prior distribution. The simplest such distribution takes the form of linear interpolation

$$\widetilde{\text{Pr}}(\mathbf{h}_j|\mathbf{s}_i) = \eta \cdot \text{Pr}(\mathbf{h}_j|\mathbf{s}_i) + (1 - \eta) \cdot \text{Prior} \quad (5.64)$$

where Prior is the prior, and  $\eta$  is the interpolation coefficient. When  $\eta = 1$ , it is a standard attention model. By contrast, when  $\eta = 0$ , the attention is completely dependent on the prior [You et al., 2020].

The prior can be chosen in many ways. A simple choice is to assume Prior to be a Gaussian distribution  $\text{Gaussian}(\mu, \sigma^2)$ . This makes the model well explained: the attention is concentrated on some point of the sequence and decreases its strength as we spread the attention from this point. To determine the mean and variance of the Gaussian distribution, we can use the same technique described above, say, we develop two neural networks to compute them respectively.

The interpolation can also be considered an intermediate step of computing the attention distribution. For example, consider the QKV attention discussed in Section 5.3.2. The interpolation can be placed on the query-key dot-product [Yang et al., 2018a; Guo et al., 2019]. To this end, we can modify Eq.(5.28) in the following form

$$\begin{aligned} \alpha_j &= \text{Softmax}\left(\frac{\mathbf{qk}_j^T}{\beta} + \eta \text{Prior}\right) \\ &= \text{Softmax}\left(\frac{\mathbf{s}_i \mathbf{h}_j^T}{\beta} + \eta \text{Prior}\right) \end{aligned} \quad (5.65)$$

As  $\frac{\mathbf{qk}_j^T}{\beta}$  (or  $\frac{\mathbf{s}_i \mathbf{h}_j^T}{\beta}$ ) is not constrained in  $[0, 1]$ , Prior is re-scaled by a hyper-parameter  $\eta$ .

Sometimes, priors arise in the context where the knowledge of attention comes from external sources. As discussed above, incorporating word alignments into attention models is one of the simplest ways to do this. The idea can be extended to make use of more structural information, such as fertility and coverage [Cohn et al., 2016; Feng et al., 2016; Tu et al., 2016], or more task-specific constraints, such as monotonic alignments between input and output sequences [Raffel et al., 2017; Chiu and Raffel, 2018]. Also, as with syntactic machine translation systems, parse trees can be used to bias the process of attention as an auxiliary input. For example, dependency trees are a widely used source of information in modeling word correspondence for either sequence-to-sequence [Chen et al., 2018a] or sequence modeling problems [Zhang et al., 2020c; Nguyen et al., 2020; Xu et al., 2021b].

Since attention models can be computationally expensive in large-scale applications, researchers have also developed efficient attention models by introducing more inductive

biases into model design [Tay et al., 2020b]. This line of research can broadly be categorized into efficient methods for NLP. In Chapter 6 we will present a discussion.

### 3. Attention in Memory Networks

As well as being of great interest in sequence-to-sequence systems, the attention mechanism is extensively used in memory-based neural models [Sukhbaatar et al., 2015; Graves et al., 2014; Kumar et al., 2016]. As discussed in Chapter 4, a memory system maintains a collection of data items and allows users to retain and retrieve information. Given a query, it computes, in some way, the match between the query and the key of each data item. This procedure is also called **addressing** [Graves et al., 2014]. Such addressing is typically implemented by first representing the query and the data item as real-valued vectors, and then calculating a weight by considering some similarity between the two vectors. The result of the retrieval is a weighted sum of all the data items. This formalism fits perfectly with the model of the QKV attention discussed in Section 5.3.2.

Provided the attention mechanism and the memory mechanism are correlated, though not from a psychology perspective, we can view attention as a process of retrieving information in a memory (i.e.,  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ ) for a given query (i.e.,  $s_i$ ). Thus we can interpret a sequence-to-sequence system with the attention mechanism as follows. On the source-side, we store information in a memory represented as a sequence of vectors  $\mathbf{h}_1 \dots \mathbf{h}_m$ . Then, we retrieve from this memory to recover step by step the source-side information on the target-side.

### 4. Beyond Sequence-to-Sequence Problems

While we restrict our discussion to the problem of transformation from one sequence to another sequence in this section, the general approach of attention can be used to deal with other problems. As mentioned in Section 5.3.2, and will be discussed in Chapter 6, a well-known variant of this approach is self-attention. In self-attention, the QKV attention model is used as a sequence model, and we have only one sequence of variables as input. As a result, the outputs of this attention model can be treated as new representations of the input sequence. Self-attention provides a general approach to modeling the interactions and dependencies between input variables, and so can be applied to a variety of problems. For example, we can concatenate  $\mathbf{h}_1 \dots \mathbf{h}_m$  and  $s_1 \dots s_n$  as a new sequence  $\mathbf{h}_1 \dots \mathbf{h}_m s_1 \dots s_n$ , and run this model on the sequence. In this way, self-attention is easily extended to a sequence-to-sequence model [Lample and Conneau, 2019; Raffel et al., 2020]. Such an approach even works when  $\mathbf{h}_1 \dots \mathbf{h}_m$  and  $s_1 \dots s_n$  represent different types of data. For example, we can use  $\mathbf{h}_1 \dots \mathbf{h}_m$  to represent a text and use  $s_1 \dots s_n$  to represent an image. Then, we have a multi-modal model that fuses textual and visual representations by performing self-attention on them [Chen et al., 2020c].

Another approach to joint representation learning of sequences is to build multiple attention models so that each sequence can learn from other sequences. An example of such models is **co-attention**, which has been widely used in multi-modal language processing [Lu et al., 2016]. For example, for the purposes of **visual question answering (VQA)**, we wish to figure out which parts of the image are related to a word of the question and to figure out which words of the question are related to a given part of the image. To do this we will build two

attention models: one for image-to-text attention, and one for text-to-image attention. The outputs of both models can be thought of as joint representations for the image and text, and thus can be used as features for answer prediction.

The attention models discussed in this section are order-independent for input. This is an issue for dealing with sequential data, and can be addressed by encoding order information in the inputs themselves (see Chapters 4 and 6). On the other hand, the simplicity of this formulation makes these models general: the input data of the models needs not to be sequential. As a result, the attention models can be directly applied to more complex data, such as graphs [Veličković et al., 2018; Lee et al., 2019].

## 5.4 Search

Search is a fundamental issue in artificial intelligence, and plays an important role in many NLP systems. The search problem is a computational challenge here because the number of hypotheses in the search space increases exponentially with the length of the sequence and the size of the vocabulary on the target-side. Exhaustive search in this case is simply slow. Therefore, real-world systems often involve search algorithms or heuristics to ensure that optimal or sub-optimal solutions can be found in an acceptable time.

For many practical sequence-to-sequence applications, the search problem, also called the inference problem sometimes, can be formulated as the following equation

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \Omega} \text{Score}(\mathbf{x}, \mathbf{y}) \quad (5.66)$$

where  $\text{Score}(\mathbf{x}, \mathbf{y})$  is a model that measures the goodness of  $\mathbf{y}$  given  $\mathbf{x}$ .

This equation takes a slightly different form from that of Eq. (5.2). First, we use  $\text{Score}(\mathbf{x}, \mathbf{y})$  instead of  $\Pr(\mathbf{y}|\mathbf{x})$  as the goodness function. While a typical approach to training sequence-to-sequence models is to maximize  $\Pr(\mathbf{y}|\mathbf{x})$  (or  $\Pr(\mathbf{x}, \mathbf{y})$ ), we often need to consider task-specific problems when performing inference on test data, for example the problem of length bias. It is therefore common to involve other terms, as well as  $\Pr(\mathbf{y}|\mathbf{x})$ , to define the objective function for search (see Section 5.4.1). A second difference between Eq. (5.66) and Eq. (5.2) arises from the form of the search space which is constrained to  $\Omega$ . In general,  $\Omega$  is a pruned search space and contains a relatively small number of hypotheses. A common way to achieve this is through the use of pruning techniques and advanced search algorithms (see Section 5.4.2). In this section we consider solutions to these problems and some of the refinements. These methods are largely motivated by the development of machine translation, but the discussions here are general and can be considered in most text generation problems.

### 5.4.1 The Length Problem

Recall from Section 5.2.2 that the probability of the target-side sequence  $\mathbf{y}$  given the source-side sequence  $\mathbf{x}$  can be written as a product of probabilities of each word  $y_i$  given both the generated words  $y_0 \dots y_{i-1}$  and  $\mathbf{x}$ . Here we re-express Eq. (5.14) using simpler notation, as

follows

$$\Pr(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^n \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}) \quad (5.67)$$

where  $\mathbf{y}_{<i}$  denotes the sequence  $y_0 \dots y_{i-1}$ . This can be equivalently expressed in terms of log probabilities

$$\log \Pr(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}) \quad (5.68)$$

Such a simple form of modeling has clear advantages as practical models for NLP, but unfortunately, this leads to a preference for shorter target-side sequences over longer target-side sequences. So it seems implausible to simply take  $\text{Score}(\mathbf{x}, \mathbf{y}) = \Pr(\mathbf{y}|\mathbf{x})$  or  $\log \Pr(\mathbf{y}|\mathbf{x})$  in search because the result is very probably a short sequence, say, a sequence of one or two words. This problem is a direct consequence of the inductive bias of the above model. From a supervised learning perspective, another reason for this is that **teacher forcing** is used to train the model: only a ground-truth target-side sequence is considered in training, and the model is forced to output this ground-truth. By contrast, when applying this model to test data, we need to explore a big set of  $\mathbf{y}$  of different lengths, and to compare different  $\mathbf{y}$  in terms of a function that is different from the one learned on the training data.

This problem can be addressed through a technique called **length reward**, which gives bonuses to longer sequences by adding a term to  $\text{Score}(\mathbf{x}, \mathbf{y})$  [He et al., 2016c]. As discussed in Chapter 3, a commonly used form of length reward is given by

$$\text{Score}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y}|\mathbf{x}) + \lambda \cdot n \quad (5.69)$$

Here the length reward term is the length of  $\mathbf{y}$  (i.e.,  $n = |\mathbf{y}|$ ), weighted by the parameter  $\lambda > 0$ . Improvements on this approach involve replacing  $n$  with an estimated sequence length by using a length prediction model. For example, we can bound the reward in the following form [Huang et al., 2017b; Yang et al., 2018b]

$$\text{Score}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y}|\mathbf{x}) + \lambda \cdot \max(l_p, n) \quad (5.70)$$

where  $l_p$  is a predicted length, and is generally defined to be a scaled length of  $\mathbf{x}$ , that is,  $l_p = \text{scalar}_p \cdot m$ .

If we substitute the log probability  $\log \Pr(\mathbf{y}|\mathbf{x})$  given by Eq. (5.68) into Eq. (5.69), we obtain

$$\begin{aligned} \text{Score}(\mathbf{x}, \mathbf{y}) &= \sum_{i=1}^n \log \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}) + \lambda \cdot n \\ &= \sum_{i=1}^n [\log \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}) + \lambda] \end{aligned} \quad (5.71)$$

Thus, we can interpret the length reward term as a reward to each word  $y_i$ . Such a method has been widely used in **statistical machine translation (SMT)** systems in which the rewards are treated as features of a log-linear model [Koehn et al., 2003; Chiang, 2007]. To find an appropriate value of  $\lambda$ , we can either use minimum error rate training [Och, 2003], following the convention in SMT, or use gradient-based methods as in neural network-based systems [Murray and Chiang, 2018].

A second approach to biasing search towards longer sequences, called **length normalization**, is to divide  $\log \Pr(\mathbf{y}|\mathbf{x})$  by a length correction term, written in the following form

$$\text{Score}(\mathbf{x}, \mathbf{y}) = \frac{\log \Pr(\mathbf{y}|\mathbf{x})}{n_{\text{correct}}} \quad (5.72)$$

A simple example of this model is to define the length correction term as the sequence length [Jean et al., 2015], like this

$$\begin{aligned} n_{\text{correct}} &= n \\ &= |\mathbf{y}| \end{aligned} \quad (5.73)$$

In this case,  $\frac{\log \Pr(\mathbf{y}|\mathbf{x})}{n} = \frac{\sum_{i=1}^n \log \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x})}{n}$  can be viewed as the log-scale geometric mean of the probabilities  $\{\Pr(y_i | \mathbf{y}_{<i}, \mathbf{x})\}$ <sup>5</sup>.

Another example is the one used in the GNMT system [Wu et al., 2016]. It takes the exponential of the shifted, re-scaled  $n$ , as follows

$$n_{\text{correct}} = \frac{(5+n)^\alpha}{(5+1)^\alpha} \quad (5.76)$$

where the power  $\alpha$  is a hyper-parameter and can be determined empirically on a tuning set. To compare different methods, Table 5.2 shows a list of scoring functions for length reward and length normalization.

In machine translation, the length problem is also closely related to the **coverage** problem which has been discussed extensively in SMT. When translating a source-side sequence, we wish to know how many times each word is translated. Then, we will say that **over-translation** occurs (i.e., a longer translation) if some words are translated too many times, and that **under-translation** occurs (i.e., a shorter translation) if some words are not sufficiently translated. Traditionally, the coverage of a source-side sequence is described in terms of an  $m$ -dimensional

<sup>5</sup>Suppose  $\{a_1, \dots, a_n\}$  are  $n$  variables. Since

$$\exp\left(\frac{\sum_{i=1}^n \log a_i}{n}\right) = \left(\prod_{i=1}^n a_i\right)^{\frac{1}{n}} \quad (5.74)$$

we have

$$\frac{\sum_{i=1}^n \log a_i}{n} = \log\left(\prod_{i=1}^n a_i\right)^{\frac{1}{n}} \quad (5.75)$$

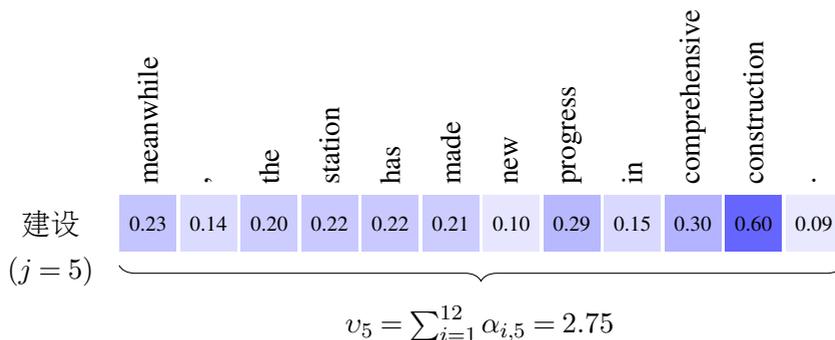
Method	Form of Score( $\mathbf{x}, \mathbf{y}$ )
No Reward/Normalization	Score( $\mathbf{x}, \mathbf{y}$ ) = $\log \Pr(\mathbf{y} \mathbf{x})$
Length Reward	Score( $\mathbf{x}, \mathbf{y}$ ) = $\log \Pr(\mathbf{y} \mathbf{x}) + \lambda \cdot n$
Bounded Length Reward	Score( $\mathbf{x}, \mathbf{y}$ ) = $\log \Pr(\mathbf{y} \mathbf{x}) + \lambda \cdot \max(l_p, n)$
Length Normalization (Basic)	Score( $\mathbf{x}, \mathbf{y}$ ) = $\frac{\log \Pr(\mathbf{y} \mathbf{x})}{n}$
Length Normalization (GNMT)	Score( $\mathbf{x}, \mathbf{y}$ ) = $\frac{\log \Pr(\mathbf{y} \mathbf{x})}{(5+n)^\alpha / (5+1)^\alpha}$

Table 5.2: Scoring functions for length reward and length normalization.  $m = |\mathbf{x}|$ ,  $n = |\mathbf{y}|$ , and  $l_p = \text{scalar}_p \cdot m$ .  $\lambda$  and  $\alpha$  are parameters.

vector  $[v_1 \dots v_m]$ , called the **coverage vector**.  $v_j$  describes to what extent the source-side word  $x_j$  is translated. In SMT systems  $v_j$  is a binary variable: 0 denotes untranslated, and 1 denotes translated. However, NMT systems have no such symbolic coverage mechanism. Instead, they have models that compute the attention weights between  $x_j$  and all the target-side words. Therefore, one way to define what we mean by the coverage of a word is to consider how strong  $x_j$  connects to the target-side words. To do this, we extend  $v_j$  to be a continuous variable, given by

$$v_j = \sum_{i=1}^n \alpha_{i,j} \quad (5.77)$$

$v_j$  can thus be viewed as the “number of times”  $x_j$  is translated, say,  $v_j = 0$  means that  $x_j$  is not translated at all, and  $v_j = 1$  means that  $x_j$  is counted only once in translation. Consider the example in Figure 5.9. For the source-side word 建设, the corresponding attention weights are shown below.



We will say that 建设 is translated 2.75 times. It is possible to make use of  $\{v_1, \dots, v_m\}$  to define how much the source-side sequence is covered in translation. A simple way to do this is to develop a coverage score  $\text{cp}(\mathbf{x}, \mathbf{y})$  by combining  $\{v_1, \dots, v_m\}$ . For example, the GNMT system defines  $\text{cp}(\mathbf{x}, \mathbf{y})$  in the following form

$$\text{cp}(\mathbf{x}, \mathbf{y}) = \beta \sum_{j=1}^m \log(\min(v_j, 1)) \quad (5.78)$$

where  $\beta$  is a weight for the coverage model. The underlying idea is that when  $v_j \geq 1$  the word  $x_j$  is assumed to be adequately translated; when  $v_j < 1$  the word  $x_j$  is assumed to be lack of translation. Thus  $\text{cp}(\mathbf{x}, \mathbf{y})$  penalizes hypotheses in which some of the source-side words miss parts of the translations. An improvement to this form is given by Li et al. [2018]

$$\text{cp}(\mathbf{x}, \mathbf{y}) = \beta \sum_{j=1}^m \log(\max(v_j, \gamma)) \quad (5.79)$$

where  $\gamma$  is the hyper-parameter for truncation, giving a tolerance for under-translation. A similar form was proposed in [Chorowski and Jaitly, 2017]

$$\text{cp}(\mathbf{x}, \mathbf{y}) = \beta \sum_{j=1}^m \mathbb{1}(v_j > \gamma) \quad (5.80)$$

It just counts the number of times  $v_j$  is greater than  $\gamma$ .

$\text{cp}(\mathbf{x}, \mathbf{y})$  can be easily introduced into search by adding it to  $\text{Score}(\mathbf{x}, \mathbf{y})$ . For example, the GHKM-style scoring function is defined to be

$$\text{Score}(\mathbf{x}, \mathbf{y}) = \frac{\log \Pr(\mathbf{y}|\mathbf{x})}{(5+n)^\alpha / (5+1)^\alpha} + \text{cp}(\mathbf{x}, \mathbf{y}) \quad (5.81)$$

In practice, modifying  $\text{Score}(\mathbf{x}, \mathbf{y})$  is not the only way to address the length problem in search. An alternative approach is to have architecture changes for modeling the problem [Tu et al., 2016; Mi et al., 2016a; Sankaran et al., 2016; See et al., 2017; Malaviya et al., 2018]. Note that, sometimes the length of the target-side sequence has been specified or predicted in some way. In these cases, we can either develop models not dependent on the auto-regressive assumption [Gu et al., 2018], or develop length-controllable text generation systems for interesting applications [Rush et al., 2015; Kikuchi et al., 2016].

## 5.4.2 Pruning and Beam Search

There are many ways to define a search space. As a general concept in computer science, a search space is often referred to as the domain of the problem that is searched. For sequence-to-sequence problems, we can think of a hypothesis as a mapping from a source-side sequence  $\mathbf{x}$  to a target-side sequence  $\mathbf{y}$ , and can think of a search space as a collection of such hypotheses<sup>6</sup>.

We can implement a search program by organizing hypotheses in an understandable way so that we can look at the search space for the problem. Recall that in Eqs. (5.67-5.68) we assign a probability of  $\mathbf{y}$  given  $\mathbf{x}$  by using a left-to-right factorization. A typical search system maintains a set of hypotheses (or partial hypotheses) and builds up these hypotheses from left to right<sup>7</sup>. The search procedure begins with an initial hypothesis set  $Z_0$  containing

<sup>6</sup>Here we use  $(\mathbf{x}, \mathbf{y})$  to denote a hypothesis. When there are multiple mappings from  $\mathbf{x}$  to  $\mathbf{y}$ , a hypothesis can be represented as  $(\mathbf{x}, \mathbf{y}, d)$  where  $d$  denotes the mapping. For example, if we transform  $\mathbf{x}$  to  $\mathbf{y}$  with a synchronous grammar, there might be multiple derivations of grammar rules to do this.

<sup>7</sup>A hypothesis is called partial when the corresponding target-side sequence does not end with  $\langle \text{EOS} \rangle$ , i.e., an incomplete target-side sequence. In this section we use the terms *hypothesis* and *partial hypothesis* interchangeably

only one hypothesis  $z_0$  whose target-side is  $y_0$  by considering  $y_0 = \langle \text{SOS} \rangle$  is the start symbol for all target-side sequences. Then, we extend this hypothesis set over a number of search steps. Suppose we have a sequence of hypothesis sets  $Z_0 \dots Z_{n_{\max}}$  where  $n_{\max}$  is the maximum number of search steps. At step  $i$ , we wish to extend each hypothesis by adding a new word  $v_k$  drawn from the vocabulary  $V_y$ . Let  $z.src$  be the source-side of  $z$  and  $z.tgt$  be the target-side of  $z$ . Clearly, we have  $z.src = \mathbf{x}$  for any  $z$ . Given a hypothesis  $z_{\text{cur}} \in Z_{i-1}$ , we can extend it to  $|V_y|$  hypotheses  $\{z_{\text{next}}^1, \dots, z_{\text{next}}^{|V_y|}\}$ , given by

$$\begin{aligned} \{z_{\text{next}}^1, \dots, z_{\text{next}}^{|V_y|}\} &= \text{Extend}(z_{\text{cur}}, V_y) \\ &= \bigcup_{v_k \in V_y} \text{Extend}(z_{\text{cur}}, v_k) \end{aligned} \quad (5.82)$$

Here  $\text{Extend}(z_{\text{cur}}, v_k)$  is a function that extends the input hypothesis  $z_{\text{cur}}$  with a word  $v_k \in V_y$ . The target-side of a resulting hypothesis is the concatenation of  $z_{\text{cur}}.tgt$  and  $v_k$ , written as<sup>8</sup>,

$$z_{\text{next}}^k.tgt = z_{\text{cur}}.tgt \circ v_k \quad (5.83)$$

These new hypotheses  $\{z_{\text{next}}^1, \dots, z_{\text{next}}^{|V_y|}\}$  are then added to  $Z_i$ . Figure 5.10 illustrates a few steps in this hypothesis extension process. We see that all the hypotheses can easily be represented as a tree structure. Here  $Z_i$  corresponds to a set of the nodes at level  $i$  of the search tree, and we simply have

$$|Z_i| = |V| \cdot |Z_{i-1}| \quad (5.84)$$

In other words, the size of  $Z_i$  grows exponentially with the number of steps, say,  $|Z_i| = |V|^i$ .

Each hypothesis  $z$  is associated with a log probability  $\log \Pr(z.tgt|z.src)$ .  $\log \Pr(z.tgt|z.src)$  simply takes the form of Eq. (5.68), and can be defined in a recursive fashion

$$\begin{aligned} \log \Pr(z_{\text{next}}^k.tgt|z_{\text{next}}^k.src) &= \log \Pr(z_{\text{cur}}.tgt|z_{\text{cur}}.src) + \\ &\quad \log \Pr(v_k|z_{\text{cur}}.tgt, z_{\text{cur}}.src) \end{aligned} \quad (5.85)$$

As an example, suppose  $z_{\text{next}}^k.tgt = y_0 \dots y_{i+1}$ . The form of Eq. (5.85) becomes clear from the following rewriting

$$\begin{aligned} \underbrace{\log \Pr(y_0 \dots y_{i+1} | \mathbf{x})}_{\log \Pr(z_{\text{next}}^k.tgt|z_{\text{next}}^k.src)} &= \underbrace{\log \Pr(y_0 \dots y_i | \mathbf{x})}_{\log \Pr(z_{\text{cur}}.tgt|z_{\text{cur}}.src)} + \underbrace{\log \Pr(y_{i+1} | y_0 \dots y_i, \mathbf{x})}_{\log \Pr(v_k|z_{\text{cur}}.tgt, z_{\text{cur}}.src)} \\ &= \sum_{k=1}^i \log \Pr(y_k | \mathbf{y}_{<k}, \mathbf{x}) + \log \Pr(y_{i+1} | y_0 \dots y_i, \mathbf{x}) \\ &= \sum_{k=1}^{i+1} \log \Pr(y_k | \mathbf{y}_{<k}, \mathbf{x}) \end{aligned} \quad (5.86)$$

because their forms are the same.

<sup>8</sup>We use  $a \circ b$  to denote the concatenation of two strings  $a$  and  $b$ .

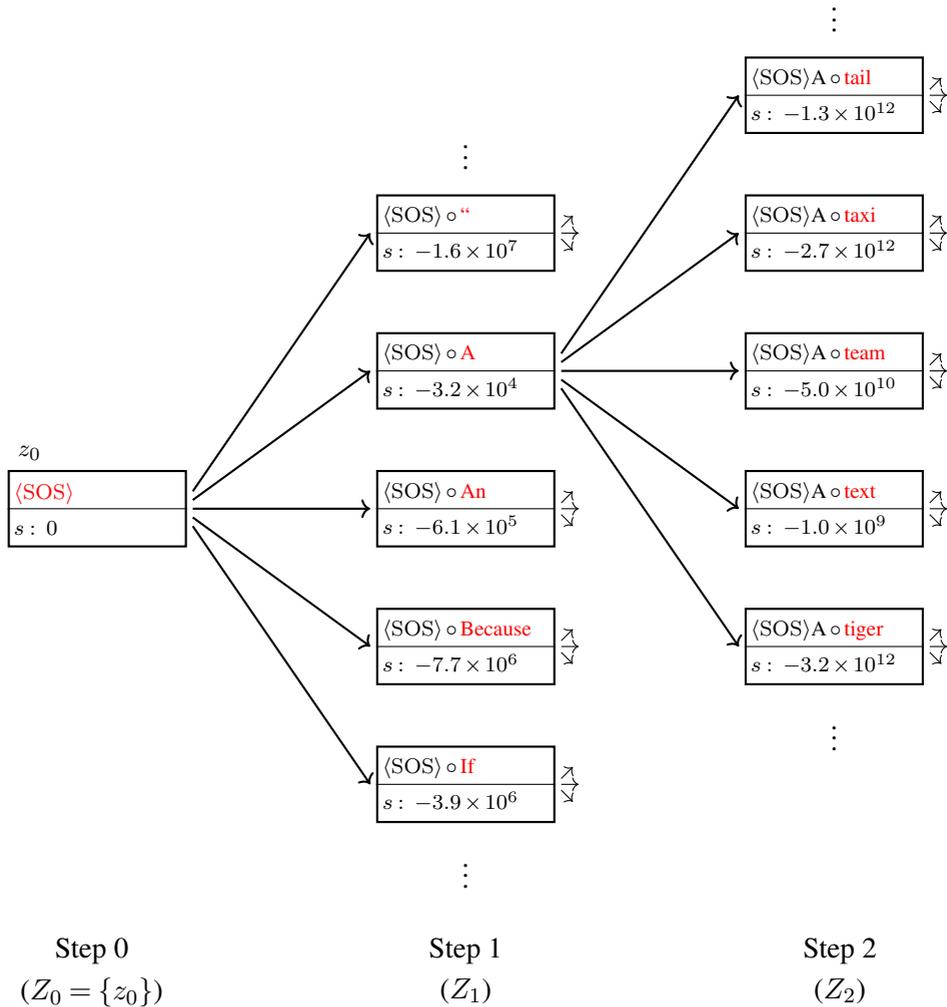


Figure 5.10: Illustration of hypothesis extension in first 3 steps. Each (partial) hypothesis is represented as a box in which we show the corresponding target-side sequence and model score. Each search step is associated with a hypothesis set  $Z_i$ . We start with a hypothesis  $z_0 \in Z_0$  denoting the start symbol  $\langle \text{SOS} \rangle$ . In step  $i$ , we extend every hypothesis in  $Z_{i-1}$  by trying to append every word from a vocabulary  $V$  (see words in red). This operation will result in  $|V| \cdot |Z_{i-1}|$  hypotheses, forming the hypothesis set  $Z_i$ . The hypothesis extension procedure represents a breadth-first search algorithm: we create all the nodes (or search states) at depth  $i - 1$  before moving to depth  $i$ . A tree structure is created along with this procedure, and a leaf node of the tree can trace the search path back to the root node.

Given this probability, we can then compute  $z.score = \text{Score}(z.src, z.tgt)$ , as in Section 5.4.1. This enables us to compare different hypotheses in terms of  $z.score$ . If a hypothesis ends with the symbol  $\langle \text{EOS} \rangle$ , it is called complete and is not extended anymore. Once a hypothesis

is complete, it is added to a max-heap<sup>9</sup>. We can dump the hypotheses with maximum model scores from the heap. In general, the search procedure will stop if we find a certain number of complete hypotheses. For example, we can stop searching when the heap is full (see more discussions later in this subsection). The resulting search algorithm is described below.

**Algorithm:** A Simple Breadth-first Search Algorithm

`SimpleSearch(x)`

// Search for the best hypothesis given the source-side sequence  $\mathbf{x}$

1. Create a Heap with  $size_{\text{heap}}$  elements
2.  $Z_0 = \{z_0\}$  where  $z_0.src = \mathbf{x}$  and  $z_0.tgt = y_0$
3. For each step  $i = 1$  to  $n_{\text{max}}$
4.     For each hypothesis  $z_{\text{cur}} \in Z_{i-1}$
5.         For each word  $v_k \in V_y$
6.              $z_{\text{next}} = \text{Extend}(z_{\text{cur}}, v_k, \mathbf{x})$
7.             If  $z_{\text{next}.tgt}$  ends with  $\langle \text{EOS} \rangle$ , then
8.                 Add  $z_{\text{next}}$  to Heap
9.             Else
10.                 Add  $z_{\text{next}}$  to  $Z_i$
11.             If Heap is full and/or other stopping criteria are met, then
12.                 Break all the loops
13. return Heap.Pop()

`Extend( $z_{\text{cur}}, v_k, src$ )`

// Create a new hypothesis by appending a new word  $v_k$  to the target-side of  $z_{\text{cur}}$

1. Create a new hypothesis  $z_{\text{next}}$
2.  $z_{\text{next}.src} = src$
3.  $z_{\text{next}.tgt} = z_{\text{cur}.tgt} \circ v_k$
4.  $z_{\text{next}.prob} = z_{\text{cur}.prob} + \log \Pr(v_k | z_{\text{cur}.tgt}, z_{\text{cur}.src})$      // see Eq. (5.85)
5.  $z_{\text{next}.score} = \text{score}(z_{\text{next}.src}, z_{\text{next}.tgt})$      // see Section 5.4.1
6. Return  $z_{\text{next}}$

If the hypothesis heap has an infinite capacity ( $size_{\text{heap}} = \infty$ ), this algorithm will perform an exhaustive search over a space of all hypotheses whose target-side lengths are up to  $n_{\text{max}}$ , resulting in at most  $1 + |V_y| + |V_y|^2 + \dots + |V_y|^{n_{\text{max}}} = \frac{|V_y|^{n_{\text{max}}+1} - 1}{|V_y| - 1}$  complete hypotheses. This is an extremely huge search space which is computationally intractable in practice<sup>10</sup>. Therefore, in practical systems it is common to prune the search space in order to make the search tractable. In later parts of this subsection we will introduce two popular search algorithms, both adopting pruning for efficient search.

<sup>9</sup>Given a max-heap  $a$ , we use  $a.Pop()$  to denote a function popping the top-1 item of  $a$ , and use  $a.PopAll()$  to denote a function popping all the items of  $a$ .

<sup>10</sup>Consider, for example, a vocabulary size of 20,000 ( $|V_y| = 20,000$ ) and a length limit of 20 ( $n_{\text{max}} = 20$ ).  $\frac{|V_y|^{n_{\text{max}}+1} - 1}{|V_y| - 1}$  would be  $1.05 \times 10^{86}$ .

## 1. Greedy Search

The **greedy strategy** is one of the most common concepts that one learns in textbooks on algorithms. It is based on a heuristic that the globally optimal solution can be approximated by making locally optimal decisions. Although such an approximation can only obtain a locally optimal solution, this is sufficient for many practical applications and its low computational complexity is a clear advantage.

Applying the greedy strategy to the search problem here is straightforward. In each extension given step  $i$ , we only consider the best hypothesis up to  $i$ . To be more precise, for any  $Z_i$ , we only keep the hypothesis with the highest model score and discard the rest. The output of each step of the greedy search is given by

$$z_{\text{best}} = \underset{z_{\text{next}} \in \text{Extend}(Z_{i-1}, V_y)}{\text{arg max}} \quad z_{\text{next}}.\text{score} \quad (5.87)$$

Here the function  $\text{Extend}(Z_{i-1}, V_y)$  has the same meaning as that in Eq. (5.82), but operates on a set of hypotheses, that is,

$$\text{Extend}(Z_{i-1}, V_y) = \bigcup_{z \in Z_{i-1}} \text{Extend}(z, V_y) \quad (5.88)$$

Then,  $Z_i$  is defined to be

$$Z_i = \{z_{\text{best}}\} \quad (5.89)$$

A greedy search algorithm for sequence-to-sequence problems is described below.

### Algorithm: A Greedy Search Algorithm

#### GreedySearch( $\mathbf{x}$ )

// Search for the “best” hypothesis in a greedy manner

1. Create a hypothesis  $z_{\text{best}}$
2.  $Z_0 = \{z_0\}$  where  $z_0.\text{src} = \mathbf{x}$  and  $z_0.\text{tgt} = y_0$
3. For each step  $i = 1$  to  $n_{\text{max}}$
4.      $z_{\text{best}}.\text{score} = -\infty$
5.     For each hypothesis  $z_{\text{cur}} \in Z_{i-1}$
6.         For each word  $v_k \in V_y$
7.              $z_{\text{next}} = \text{Extend}(z_{\text{cur}}, v_k, \mathbf{x})$
8.             If  $z_{\text{best}}.\text{score} < z_{\text{next}}.\text{score}$ , then
9.                  $z_{\text{best}} = z_{\text{next}}$
10.     If  $z_{\text{best}}.\text{tgt}$  ends with  $\langle \text{EOS} \rangle$  and/or other stopping criteria are met, then
11.         Break the loop
12.      $Z_i = \{z_{\text{best}}\}$
13. Return  $z_{\text{best}}$

In each step of search, we have only one active hypothesis to extend (i.e.,  $|Z_{i-1}| = 1$ ) and

therefore need  $|V|$  extensions from which we select the best one for the next step of search. The total number of times  $\text{Extend}(z_{\text{cur}}, v_k)$  is called is  $|V| \cdot n_{\text{max}}$ . Provided  $\text{Extend}(z_{\text{cur}}, v_k)$  is a fixed-cost function, the time complexity of the algorithm is linear with respect to  $|V|$  and  $n_{\text{max}}$ .

## 2. Beam Search

**Beam search** is a natural extension of the above 1-best greedy search algorithm. It is based on the greedy heuristics as well, and is thus a type of greedy algorithm. The idea of beam search is to keep at each step a number of the most promising hypotheses rather than the 1-best hypothesis. A beam is a data structure that stores the best hypotheses we have generated so far. The number of hypotheses in a beam is a predetermined parameter, called **beam width** or **beam size**. Here we can simply view  $Z_i$  as a beam, written as

$$Z_i = \{z_{\text{best}}^1, \dots, z_{\text{best}}^{\text{size}_{\text{beam}}}\} \quad (5.90)$$

where  $\text{size}_{\text{beam}}$  is the beam size.  $z_{\text{best}}^1$  is the best hypothesis in the extension  $\text{Extend}(Z_{i-1}, V_y)$  (see Eq. (5.87)),  $z_{\text{best}}^2$  is the 2nd best hypothesis in  $\text{Extend}(Z_{i-1}, V_y)$ , and so on.

The following pseudo-code describes a beam search algorithm for sequence-to-sequence problems.

### Algorithm: A Beam Search Algorithm

#### BeamSearch( $\mathbf{x}$ )

// Search for the “best” hypothesis by considering a number of best candidates

// in each step

1. Create a Heap with  $\text{size}_{\text{heap}}$  elements
2.  $Z_0 = \{z_0\}$  where  $z_0.\text{src} = \mathbf{x}$  and  $z_0.\text{tgt} = y_0$
3. For each step  $i = 1$  to  $n_{\text{max}}$
4. Create a heap Beam with  $\text{size}_{\text{beam}}$  elements
5. For each hypothesis  $z_{\text{cur}} \in Z_{i-1}$
6. For each word  $v_k \in V_y$
7.  $z_{\text{next}} = \text{Extend}(z_{\text{cur}}, v_k, \mathbf{x})$
8. If  $z_{\text{next}}.\text{tgt}$  ends with  $\langle \text{EOS} \rangle$ , then
9. Add  $z_{\text{next}}$  to Heap
10. Else
11. UpdateBeam(Beam,  $z_{\text{next}}$ )
12. If Heap is full and/or other stopping criteria are met, then
13. Break all the loops
14.  $Z_i = \text{Beam.PopAll}()$
15. Return Heap.Pop()

#### UpdateBeam(Beam, $z_{\text{next}}$ )

// Update Beam with a newly-generated hypothesis  $z_{\text{next}}$

1. Add  $z_{\text{next}}$  to Beam <sup>a</sup>

<sup>a</sup>Beam is a max-heap with  $size_{\text{beam}}$  elements. So, if  $z_{\text{next}}.score$  is lower than all the elements in the heap, the heap will be left unchanged. In other words, Beam only stores top- $size_{\text{beam}}$  best hypotheses and ignores the rest.

The function  $\text{UpdateBeam}(\text{Beam}, z_{\text{next}})$  is a direct implementation of **histogram pruning**. Note that this general-purpose framework provides a simple way to implement other pruning methods, and one can modify  $\text{UpdateBeam}(\text{Beam}, z_{\text{next}})$  as needed. For example, an alternative method, called **threshold pruning**, retains the hypotheses whose differences in model scores with the best hypothesis in Beam are below a threshold  $\theta_{\text{beam}}$ , say, we discard  $z_{\text{next}}$  in  $\text{UpdateBeam}(\text{Beam}, z_{\text{next}})$  if

$$z_{\text{next}}.score < z_{\text{best}}.score - \theta_{\text{beam}} \quad (5.91)$$

where  $z_{\text{best}}$  is the best hypothesis in Beam. Alternatively, we can consider a relative threshold method [Freitag and Al-Onaizan, 2017], given by

$$z_{\text{next}}.score < z_{\text{best}}.score \cdot \theta_{\text{beam}} \quad (5.92)$$

Figure 5.11 shows a comparison of exhaustive search, (1-best) greedy search and beam search. At one extreme, the optimal solution is guaranteed, but an exponentially large number of search states are visited. At the other extreme, only the minimum number of search states are visited, but the solution is sub-optimal. By contrast, beam search makes a trade-off between the two methods. A larger beam size means more search effort and a higher possibility of finding the optimum, while a smaller beam size means faster search and a higher risk of missing the optimum. It is also possible to use a variable beam size to make a better trade-off during search [Buckman et al., 2016; Post and Vilar, 2018; Kulikov et al., 2019].

An important problem related to these search algorithms is the problem of **search errors**. In general, search errors can be defined in several different ways. Here we say that a search error occurs if the search result is not the same as that of exhaustive search. Common sense tells us that fewer search errors are helpful for finding “better” results. Thus, we often wish to have a more desirable target-side sequence by enlarging the beam size. However, this is not the case for some sequence-to-sequence systems. For example, a search with a larger beam size may lead to a lower translation quality for neural machine translation systems [Koehn and Knowles, 2017]. This inspires very interesting studies on the deterioration issue of large beam search [Ott et al., 2018b; Yang et al., 2018b; Stahlberg and Byrne, 2019].

### 3. Stopping Criteria

Although the time complexities of the above algorithms are bounded by the maximum number of search steps (i.e.,  $n_{\text{max}}$ ), it is important to have more efficient algorithms to stop searching as early as possible, especially for latency-sensitive applications. This typically requires heuristics to design additional criteria for stopping the search procedure at the appropriate point. Some of these stopping criteria are:

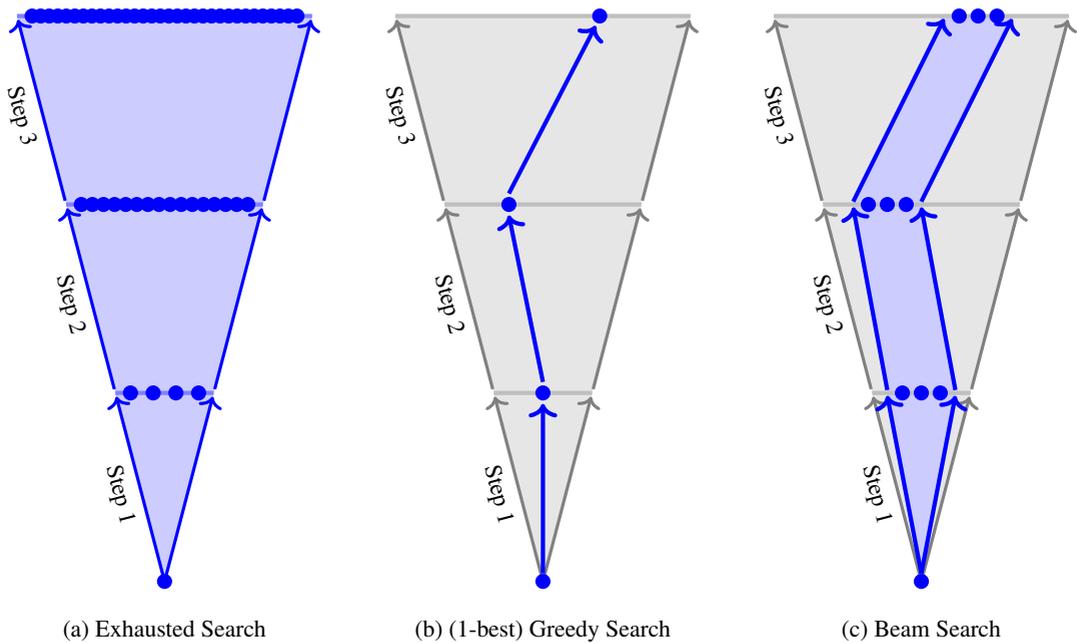


Figure 5.11: A comparison of exhaustive search, (1-best) greedy search and beam search. Balls represent search states or partial hypotheses. Exhausted search explores all search states in the search space. By contrast, greedy search keeps only the 1-best path of search states and prunes away the rest. Beam search is a trade-off between them and keeps the most promising search states in a beam in each step.

- If a given number of complete hypotheses are created, then we stop searching. For example, in the beam search algorithm described in this subsection, the search program terminates when we have  $size_{heap}$  complete hypotheses. Another way to implement this idea is to shrink the beam as the number of complete hypotheses increases. In Bahdanau et al. [2014]’s system, once a new complete hypothesis is created, the beam size decreases by 1. Therefore, the search program will terminate if the beam size is reduced to 0.
- If every hypothesis at step  $i$  has a score lower than that of the best complete hypothesis in Heap by some margin, then we stop searching. Suppose  $z_{bestinall}$  is the best hypothesis we have generated so far (i.e.,  $z_{bestinall} = Heap.Pop()$ ). If every hypothesis  $z_{next}$  at step  $i$  satisfies

$$z_{bestinall}.score - z_{next}.score \geq \theta_{all} \quad (5.93)$$

then we will finish the search process at this step. Here  $\theta_{all}$  is a parameter. One can specify it with an appropriate value through multiple tries. A simple choice is  $\theta_{all} = 0$ , which is employed in some of the popular sequence-to-sequence systems [Ott et al., 2019]. Under some circumstances, such an **early-stop** strategy can guarantee the

optimality of search [Huang et al., 2017b; Yang et al., 2018b].

- If every hypothesis at step  $i$  has a score lower than that of the last complete hypothesis in Heap by some margin, then we stop searching. This is a weak condition for early-stop.
- If the top ranked hypotheses at step  $i$  are all complete hypotheses, then we stop searching. This is a more aggressive version of early-stop. For example, in Klein et al. [2017]’s system, the search program terminates at step  $i$  if the top-1 hypothesis is a complete hypothesis.
- If the search program consumes a certain amount of computing resources, such as a certain number of floating-point instructions and a certain amount of wall clock time, then we stop searching. In applications where computer performance is limited and latency plays an important role, we will often be interested in this kind of stopping criterion.

Sometimes, the search algorithm will not find any complete hypothesis until hitting the length limit  $n_{\max}$ . As a practical matter it might be easy in this case to force the best partial hypothesis to be complete by adding  $\langle \text{EOS} \rangle$  to its end.

Note that choosing appropriate stopping criteria reflects a trade-off between fast computation and accurate prediction at inference time (call it the **speed-accuracy trade-off**). While it is not always the case that more time a search program takes could result in better results for a sequence-to-sequence system, we would always want to know how close we can get to a better solution to the problem by searching through a larger region of the search space. A discussion of accurate search algorithms can be found in Section 5.4.4.

### 5.4.3 Online Search

So far in our general discussion of sequence-to-sequence problems, we have assumed that all the source-side words come together as a whole and can be accessed in the entire search process. However, in some practical applications, the inputs are received in order, and we wish to make predictions conditioned on some of the observed inputs. An example of this is online automatic speech recognition in which the system continually takes new acoustic signals and at the same time outputs the corresponding transcription units.

Intuitively, we might think of the generation of the  $i$ -th target-side word as a problem of mapping a prefix of the source-side sequence to the target-side vocabulary. We can formulate this by introducing a function  $g(i)$  which denotes the maximum length of the prefix of  $\mathbf{x}$  we use in generating  $y_i$ . Thus, the probability of  $y_i$  given the entire sequence  $\mathbf{x}$  and the previously generated words  $\mathbf{y}_{<i}$  can be approximated by

$$\Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}) \approx \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}_{\leq g(i)}) \quad (5.94)$$

where  $\mathbf{x}_{\leq g(i)}$  denotes the sub-sequence  $x_1 \dots x_{g(i)}$ . Then, the log probability of the target-side

sequence  $\mathbf{y}$  given the source-side sequence  $\mathbf{x}$  is written as

$$\begin{aligned} \log \Pr(\mathbf{y}|\mathbf{x}) &= \sum_{i=1}^n \log \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}) \\ &\approx \sum_{i=1}^n \log \Pr(y_i|\mathbf{y}_{<i}, \mathbf{x}_{\leq g(i)}) \end{aligned} \quad (5.95)$$

This equation frames a sequence-to-sequence problem as a prefix-to-prefix problem, that is, the prefix  $\mathbf{y}_{\leq i}$  is only dependent on the prefix  $\mathbf{x}_{\leq g(i)}$ . Inference for this model is simple. For each  $i$ , the search system waits until all  $g(i)$  source-side words are received, and then extends the hypotheses as usual. This can be done by reusing the algorithms described in the previous subsection. For example, we can modify the beam search algorithm and obtain the following online search algorithm.

**Algorithm:** An Online Beam Search Algorithm

`OnlineBeamSearch`( $\mathbf{x}, g(\cdot)$ )

// Online search in which the search is operated once an adequate number of input  
// words are received. In each search step, a number of the most promising candidates  
// are considered.

1. Create a Heap with  $size_{\text{heap}}$  elements
2.  $Z_0 = \{z_0\}$  where  $z_0.tgt = y_0$
3.  $j = 0$
4.  $i = 1$
5.  $input = \phi$
6. While  $i \leq n_{\text{max}}$  do
  7. If  $j < g(i)$ , then // read a word from the input stream
  8.  $input = input \circ x_j$
  9. Else // make a prediction at step  $i$
  10. // when  $g(i)$  input words are observed (stored in  $input$ )
  11. Create a heap Beam with  $size_{\text{beam}}$  elements
  12. For each hypothesis  $z_{\text{cur}} \in Z_{i-1}$
  13. For each word  $v_k \in V_y$
  14.  $z_{\text{next}} = \text{Extend}(z_{\text{cur}}, v_k, input)$
  15. If  $input$  equals  $\mathbf{x}$  and  $z_{\text{next}.tgt}$  ends with  $\langle \text{EOS} \rangle$ , then
  16. Add  $z_{\text{next}}$  to Heap
  17. Else
  18.  $\text{UpdateBeam}(\text{Beam}, z_{\text{next}})$
  19. If Heap is full and/or other stopping criteria are met, then
  20. Break all the loops
  21.  $\text{OutputPartial}(\text{Beam})$
  22.  $Z_i = \text{Beam.PopAll}()$

```

23.     i++
24. Return Heap.Pop()
OutputPartial(Beam)
// Output a partial result
1.  Display the best hypothesis in Beam

```

An advantage of this system is that the output at step  $i$  is immediate once we have seen  $\mathbf{x}_{\leq g(i)}$ . This results in an **online sequence-to-sequence system** in which input words arrive in a continuous stream and predictions can be made just after a “sufficient” number of input words are seen.

While the search problem here seems simple, much remains to be done to define  $g(i)$ . Clearly,  $g(i)$  is a monotonically non-decreasing function. As a simple example, we can define  $g(i) = m$  for any  $i$ . This will make the above algorithm precisely the same as the standard beam search algorithm that works with a complete input sequence. By contrast, in online sequence-to-sequence tasks, we want  $g(i)$  to be as small as possible, and so we can start computation as early as possible in inference. The simplest case of these is that the input and output sequences are synchronous in some way. For example, an automatic speech recognition system assigns each spectral frame a transcription unit. In this case, we have a simple correspondence between inputs and outputs:  $m = n$  (i.e.,  $|\mathbf{x}| = |\mathbf{y}|$ ), and  $x_i$  corresponds to  $y_i$ . Then, we can simply define  $g(i) = i$ , in other words, each time a new input arrives, we make a prediction.

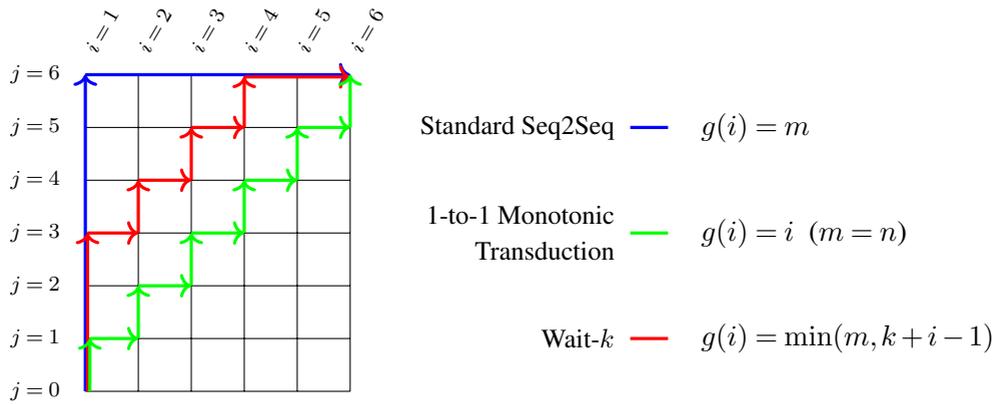
A more complicated case is online sequence-to-sequence problems with reordering, such as **simultaneous translation**, in which a target-side word may depend on source-side words with long-range dependencies. A simple way to address this is to delay the predictions for a number of steps. For example, the wait- $k$  method forces each prediction to lag behind the inputs by  $k$  words [Ma et al., 2019]. More formally, the wait- $k$  version of the function  $g(i)$  is defined to be

$$g(i) = \min(m, k + i - 1) \quad (5.96)$$

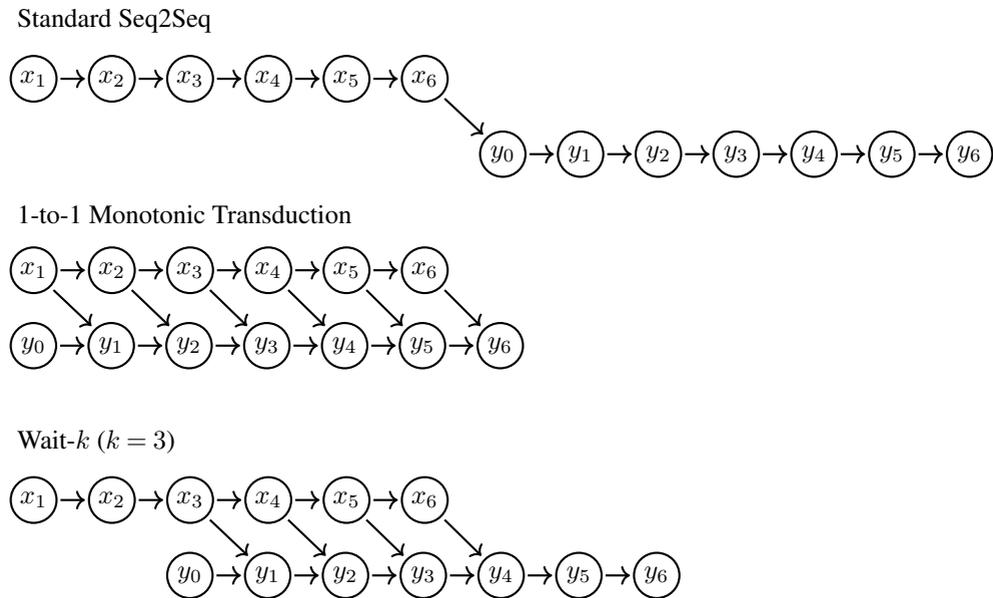
Here  $k$  is a hyper-parameter that controls how large a source-side context is considered in predicting target-side words. When  $k = \infty$ , it is the same as the standard search methods for sequence-to-sequence inference. In simultaneous translation and related tasks, results are in general satisfactory by using a small value of  $k$ . A comparison of different  $g(i)$  is shown in Figure 5.12.

In some applications of online sequence-to-sequence problems, we may know when to perform search and when to read inputs. For example, in **interactive machine translation** [Casacuberta et al., 2009], the translation of a partial input sequence is triggered by some behaviors of users (such as the action of pressing buttons). In this case, we do not need to define  $g(i)$ , but view it as an input variable of the model.

Note that while one can directly employ pre-trained sequence-level models for online inference, developing such systems often requires additional training effort. A more principled approach to online sequence-to-sequence modeling is to model the transformation from  $\mathbf{x}$  to  $\mathbf{y}$  as a sequence of actions [Grissom II et al., 2014; Cho and Esipova, 2016; Gu et al., 2017;



(a) Visualization of  $g(i)$ .



(b) Action sequences.

Figure 5.12: Visualization (top) and action sequences (bottom) of different  $g(i)$  for a pair of sequences ( $\mathbf{x} = x_1 \dots x_6, \mathbf{y} = y_1 \dots y_6$ ). In an action sequence, a circled  $x_j$  stands for the action of reading a source-side word ( $x_j$ ), and a circled  $y_i$  stands for the action of predicting the probability of  $y_i$  given  $\mathbf{x}_{\leq g(i)}$  and  $\mathbf{y}_{< i}$ . Arrows here stand for dependencies between words. Because  $y_0$  denotes the start symbol  $\langle \text{SOS} \rangle$ , it could be generated without dependencies on any words.

Zheng et al., 2019]. For example, an action can be either a predict operation that performs search at the current step, or a read operation that accepts a new input word. Then, we can frame the task of designing the function  $g(i)$  as learning a policy to determine which action is

taken given a source-side prefix  $\mathbf{x}_{\leq j}$  and a target-side prefix  $\mathbf{y}_{< i}$ . And sequence-to-sequence models can be trained on the states of these action sequences so that they can make better predictions conditioned on part of the input. However, a discussion of training online sequence-to-sequence models lies outside the scope of this section. We refer the reader to the above papers for more details on these methods.

#### 5.4.4 Exact Search

From a formal point of view, we would ideally like to develop a system with no search errors. Although approximate search algorithms have been used successfully in many applications, it is important to study model errors of these systems, and thus to focus on the problem in principle, not just in practice. So developing exact search algorithms for sequence-to-sequence models has long been an interesting topic in NLP research. However, the search problem for a simple word-based machine translation system with  $n$ -gram language models has been found to be an NP-hard problem [Knight, 1999]. Much of earlier research formulated the search problem as classical **combinatorial optimization problems**, such as the linear programming problem and the traveling salesman problem, and employed the corresponding solvers [Germann et al., 2004; Zaslavskiy et al., 2009]. Additional research efforts explored exact search algorithms for statistical machine translation systems by using the Lagrangian relaxation technique [Chang and Collins, 2011; Rush and Collins, 2012] and finite-state automata [de Gispert et al., 2010; Allauzen et al., 2014].

Unlike these methods, which are more or less dependent on the integration of  $n$ -gram language models into sequence-to-sequence models, the models described in this chapter take a simpler form. We begin with a basic model in which the scoring function  $\text{score}(\mathbf{x}, \mathbf{y})$  is the log probability  $\log \Pr(\mathbf{y}|\mathbf{x})$ . Eq. (5.68) tells us that  $\log \Pr(\mathbf{y}|\mathbf{x})$  can be written as a sum of word-level log probabilities, and  $\log \Pr(\mathbf{y}|\mathbf{x})$  becomes smaller as more target-side words are generated (i.e., a larger  $n$ )<sup>11</sup>. In other words,  $\log \Pr(\mathbf{y}|\mathbf{x})$  is a monotonic decreasing function with respect to the target-side length  $n$ : for any  $i$ , we have

$$\begin{aligned} \log \Pr(\mathbf{y}_{\leq i}|\mathbf{x}) &= \log \Pr(\mathbf{y}_{< i}|\mathbf{x}) + \log \Pr(y_i|\mathbf{y}_{< i}, \mathbf{x}) \\ &\leq \log \Pr(\mathbf{y}_{< i}|\mathbf{x}) \end{aligned} \tag{5.97}$$

This is also called the **monotonicity** of the scoring function.

Then, by making use of the monotonic nature of model scores, we can develop a heuristic to rule out hypotheses that would never be the best. Let  $z_{\text{bestinall}}$  be the global best complete hypothesis we have found. If a new hypothesis has a model score lower than  $z_{\text{bestinall}}.\text{score}$ , then we will not need to extend it. Thus we can explore a region that is significantly smaller than the original search space, without loss of optimality. Note that  $z_{\text{bestinall}}.\text{score}$  continues to become larger in search. It will be more difficult to find a better hypothesis and more hypotheses will be pruned away as the search process proceeds. See the pseudo-code below for an exact search algorithm of the sequence-to-sequence model of Eq. (5.68).

<sup>11</sup>Consider  $\log \Pr(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr(y_i|\mathbf{y}_{< i}, \mathbf{x})$ . Since  $\log \Pr(y_i|\mathbf{y}_{< i}, \mathbf{x})$  has a non-positive value,  $\log \Pr(\mathbf{y}|\mathbf{x})$  will be smaller or unchanged if  $n$  grows.

**Algorithm:** An Exact Search Algorithm

`ExactSearch(x)`

// Search for the “best” hypothesis by making use of the monotonicity of the  
// scoring function ( $\text{score}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y}|\mathbf{x})$ ).

1. Create a priority queue (max-heap) Queue
2. Create a hypothesis  $z_{\text{best}}$  with  $z_{\text{best}}.\text{score} = -\infty$
3. While Queue is not empty do
4.      $z_{\text{cur}} = \text{Queue.Pop}()$
5.     If  $|z_{\text{cur}}.\text{tgt}| > n_{\text{max}}$ , then
6.         skip  $z_{\text{cur}}$  and continue the loop
7.     For each word  $v_k \in V_y$
8.          $z_{\text{next}} = \text{Extend}(z_{\text{cur}}, v_k, \mathbf{x})$
9.          $\text{bound} = z_{\text{best}}.\text{score}$      // a lower bound on model scores
10.        If  $\text{bound} < z_{\text{next}}.\text{score}$ , then // admissible pruning
11.            If  $z_{\text{next}}.\text{tgt}$  ends with  $\langle \text{EOS} \rangle$ , then
12.                 $z_{\text{best}} = z_{\text{next}}$
13.                 $\text{bound} = z_{\text{next}}.\text{score}$
14.            Else
15.                Add  $z_{\text{next}}$  to Queue
16. Return  $z_{\text{best}}$

This is a general algorithm for exact search, and its search efficiency is greatly influenced by the design of the priority queue [Meister et al., 2020]. For example, we can view  $\text{score}(\mathbf{x}, \mathbf{y})$  as the priority of each hypothesis in the priority queue, as in a max-heap<sup>12</sup>. Then, the resulting algorithm performs a procedure of breadth-first-like search, since a hypothesis with a shorter target-side sequence is more likely to have a higher model score and to be a top-ranked item in the priority queue. For efficient search, however, we wish to find complete hypotheses as early as possible, such that more unpromising hypotheses can be thrown away in the early stage of search. To do this, we can bias the priority of a hypothesis towards a longer target-side sequence. This provides a **depth-first search** algorithm which is more likely to find complete hypotheses in a shorter time [Stahlberg and Byrne, 2019].

While the exact search algorithm becomes apparent by considering the monotonicity of  $\Pr(\mathbf{y}|\mathbf{x})$ , in practical systems, as discussed in Section 5.4.1,  $\text{score}(\mathbf{x}, \mathbf{y})$  often has a more complex form involving length reward or normalization, and so the monotonic property does not hold. Fortunately, the assumption of monotonicity can be dropped at the expense of slightly relaxing the lower bound on model scores for pruning. Here we define *bound* to be the lowest model score that a hypothesis should have so that it can at best be extended to an equally good hypothesis with  $z_{\text{best}}$ . For example, consider a simple word reward model described in Eq. (5.69):  $\text{Score}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y}|\mathbf{x}) + \lambda \cdot n$ . For a hypothesis  $z_{\text{next}}$ , there are at most

<sup>12</sup>We can implement a priority queue using a max-heap.

$n_{\max} - |z_{\text{next}}.tgt|$  words we can predict to obtain a complete hypothesis. Suppose all these  $n_{\max} - |z_{\text{next}}.tgt|$  words are predicted with a probability of 1. Then, the model score of the resulting hypothesis (denoted by  $z_{\text{new}}$ ) will be given by

$$\begin{aligned} z_{\text{new}}.score &= z_{\text{next}}.score + \sum_{i=|z_{\text{next}}.tgt|+1}^{n_{\max}} (\log 1 + \lambda) \\ &= z_{\text{next}}.score + \lambda \cdot (n_{\max} - |z_{\text{next}}.tgt|) \end{aligned} \quad (5.98)$$

Using this result, we can define *bound* as

$$bound = z_{\text{best}}.score - \lambda \cdot (n_{\max} - |z_{\text{next}}.tgt|) \quad (5.99)$$

An alternative way to derive the lower bound is to simply consider  $n_{\max}$  times of word reward, given by

$$bound = z_{\text{best}}.score - \lambda \cdot n_{\max} \quad (5.100)$$

This is a loose lower bound and leads to less pruning.

In the case of length normalization, we can do this in a similar way. For example, consider the length normalization model  $\text{Score}(\mathbf{x}, \mathbf{y}) = \frac{\log \Pr(\mathbf{y}|\mathbf{x})}{n}$ , as in Eqs. (5.72-5.73). A lower bound on admissible model scores is given by

$$bound = \frac{\Pr(z_{\text{next}}.tgt|\mathbf{x})}{n_{\max}} \quad (5.101)$$

In practice, such a lower bound can be defined in several different ways to guarantee the optimality of search, depending on which model and search strategy are used in the sequence-to-sequence systems [Huang et al., 2017b; Stahlberg and Byrne, 2019].

We can easily apply these lower bounds to the above exact search algorithm by replacing line 9 with Eq. (5.99) or (5.101). As a side effect, the search will explore more hypotheses and thus be much slower.

## 5.4.5 Differentiable Search

We have addressed the search problem through the introduction of heuristic search algorithms in which we try to minimize the scoring function on a set of sequences of discrete variables. An alternative possibility is to relax these discrete variables to continuous variables and to formulate the problem using the framework of **continuous optimization** [Hoang et al., 2017; Kumar et al., 2021]. While we try to use a consistent notation throughout this book, it is convenient here to introduce some new notation that is slightly different from that adopted in the previous chapters. We will use a vector  $\mathbf{y}_i^w \in \{0, 1\}^{|V_y|}$  to denote the one-hot representation of  $y_i$ . Suppose the output at step  $i$  is a distribution over  $V_y$ , denoted by  $\Pr(\cdot | \mathbf{y}_{<i}, \mathbf{x})$ . Then, we

can write the log probability of  $y_i$  at step  $i$  as a dot product of two vectors, like this

$$\begin{aligned}\log \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}) &= \mathbf{y}_i^w \cdot \log \Pr(\cdot | \mathbf{y}_{<i}, \mathbf{x}) \\ &= \mathbf{y}_i^w \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})\end{aligned}\quad (5.102)$$

where  $\mathbf{y}_{<i} = y_0 \dots y_{i-1}$  is represented as a sequence of one-hot vectors  $\mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w$ . As discussed in Chapter 3, the right-hand side of the above equation means the selection of the entry  $y_i$  of the vector  $\log \Pr(\cdot | \mathbf{y}_{<i}, \mathbf{x})$  (or  $\log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})$ ).

Using this notation, we can write  $\log \Pr(\mathbf{y} | \mathbf{x})$  as

$$\begin{aligned}\log \Pr(\mathbf{y} | \mathbf{x}) &= \sum_{i=1}^n \log \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}) \\ &= \sum_{i=1}^n \mathbf{y}_i^w \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})\end{aligned}\quad (5.103)$$

Provided we use  $\log \Pr(\mathbf{y} | \mathbf{x})$  as the objective function (i.e.,  $\text{score}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y} | \mathbf{x})$ ), the search problem can be formulated as

$$\hat{\mathbf{y}}_0^w \dots \hat{\mathbf{y}}_n^w = \arg \max_{\mathbf{y}_1^w \dots \mathbf{y}_n^w} \sum_{i=1}^n \mathbf{y}_i^w \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})\quad (5.104)$$

This is equivalent to the standard form for inference of sequence-to-sequence models, given by

$$\begin{aligned}\hat{\mathbf{y}} &= \hat{y}_0 \dots \hat{y}_n \\ &= \arg \max_{y_0 \dots y_n} \Pr(y_0 \dots y_n | \mathbf{x})\end{aligned}\quad (5.105)$$

Given Eq. (5.104), we can now relax each one-hot vector to a real-valued vector with a constraint that the sum of all its entries is equal to 1, that is,

$$\mathbf{y}_i^w \in +\mathbb{R}^{|V_y|}\quad (5.106)$$

$$\text{s.t. } \|\mathbf{y}_i^w\|_1 = 1\quad (5.107)$$

In this way,  $\mathbf{y}_i^w$  can be informally treated as a  $|V_y|$ -dimensional embedding of  $y_i$ , though it has much more dimensions than the usual embeddings used in NLP. Now  $\mathbf{y}_i^w$  does not correspond to a specific word in the vocabulary, but describes a distribution over the vocabulary. In Hoang et al. [2017]’s work,  $\mathbf{y}_i^w \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})$  is called the **expected embedding** under the distribution  $\log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x})$ . What is interesting about this formulation is that Eq. (5.104) in fact defines a “new” task in which we try to maximize a sum of continuous variables (i.e., a sum of  $n$  expected embeddings).

We can solve Eq. (5.104) by using the off-the-shelf toolkits in optimization. Since we have a constraint that  $\mathbf{y}_i^w$  is a variable in a **simplex**<sup>13</sup>, it is straightforward to apply general

<sup>13</sup>Simplex is a term used in geometry. In a Euclidean space, a  $k$ -simplex is a  $k$ -dimensional polytope described

**constrained optimization** algorithms to this problem. An alternative way is to use algorithms that are designed to solve the optimization problem with simplex constraints. The details of these algorithms can be found in many books on optimization.

A third choice of solving Eq. (5.104) is to formulate the constraints in the objective function explicitly and to use gradient descent methods to optimize this function. For example, Hoang et al. [2017] modify Eq. (5.104) and obtain a new form for optimization

$$\hat{\mathbf{y}}_0^w \dots \hat{\mathbf{y}}_n^w = \arg \max_{\mathbf{y}_1^w \dots \mathbf{y}_n^w} \sum_{i=1}^n \text{Softmax}(\mathbf{y}_i^w) \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w, \mathbf{x}) \quad (5.111)$$

Here we remove the simplex constraint from  $\mathbf{y}_i^w$ , and impose it on a new output that is produced by a Softmax function.

Once we have obtained the optimal sequence  $\hat{\mathbf{y}}_0^w \dots \hat{\mathbf{y}}_n^w$ , we need to map each  $\mathbf{y}_i^w$  to a unique word. A simple method is to take the word corresponding to the entry of  $\mathbf{y}_i^w$  with the largest value. However, this may break the optimality of the solution because the condition  $\mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w$  is changed when these variables are discretized. A more practical method is to perform optimization to predict the next word given a prefix, say, we fix  $\mathbf{y}_0^w \dots \mathbf{y}_{i-1}^w$  to the one-hot representations of the optimal prefix, and maximize  $\sum_{k=i}^n \mathbf{y}_k^w \cdot \log \Pr(\cdot | \mathbf{y}_0^w \dots \mathbf{y}_{k-1}^w, \mathbf{x})$ . Then, we select the best word at position  $i$  and move on to the next position.

So far we have assumed that the search objective is derived from the log probability  $\log \Pr(\mathbf{y} | \mathbf{x})$  and the length of the output is given in advance. To have a search over sequences with different lengths, we can repeat the above optimization procedure for every  $n \in [1, n_{\max}]$ , and select the sequence with the maximum score. This also makes it easy to introduce length normalization and reward into search. We can ignore the length bias issue in each search with a fixed  $n$ , and add the length models after optimization, that is, we leave the search objective unchanged, but, in the final step, we select the best sequence in a set of candidates with different  $n$  in terms of  $\text{score}(\mathbf{x}, \mathbf{y})$ .

## 5.4.6 Hypothesis Diversity

Multiple outputs are often required when one wants to rescore these outputs and/or interact with the system. One of the most widely used methods is to use beam search to generate a number of top-ranked hypotheses. For example, we can simply view the elements of Heap as the  $k$ -best hypotheses in beam search (see Section 5.4.2). However, this approach suffers from the problem that there is often little difference among the hypotheses in the beam, and

by a set of  $k+1$  independent points  $\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_k\}$ . This polytope is defined as a set of points

$$P_{k\text{-simplex}} = \{a_0 \cdot \mathbf{p}_0 + a_1 \cdot \mathbf{p}_1 + \dots + a_k \cdot \mathbf{p}_k\} \quad (5.108)$$

where

$$\sum_{i=0}^k a_i = 1 \quad (5.109)$$

$$a_i \geq 0 \text{ for any } i \in [0, k] \quad (5.110)$$

Rank	Output
1	Manuela Arbelaez accidentally revealed the correct answer to a guessing game for a new Hyundai Sonata. Host Drew Carey couldn't stop laughing. It's been a busy week for "The Price Is Right" when Bob Barker, 91, showed up to run his old show.
2	Manuela Arbelaez accidentally revealed the correct answer to a guessing game for a new Hyundai Sonata. Host Drew Carey couldn't stop laughing. It's been a busy week for "The Price Is Right" when Bob Barker showed up to run his old show.
3	Manuela Arbelaez accidentally revealed the correct answer to a guessing game for a new Hyundai Sonata. Host Drew Carey couldn't stop laughing. It's been a busy week for "The Price Is Right" when Bob Barker, 91, showed up to run the show.
4	Manuela Arbelaez accidentally revealed the correct answer to a guessing game for a new Hyundai Sonata. Host Drew Carey couldn't stop laughing. It's been a busy week for "The Price Is Right" when Bob Barker, 91, showed up to run his show.

Table 5.3: 4-best outputs of a text summarization system on a sample in the CNN/Daily Mail dataset (beam size = 4). We see that these texts differ only by a few words.

it is difficult to figure out which one is better though more options are available to users. Table 5.3 shows the 4-best outputs of a text summarization system. We see that these texts are fairly similar to each other. One reason for this phenomenon is that diverse hypotheses, though probably with high model scores when completed, will be pruned away if they are low-ranked in some stages of beam search. From a modeling perspective, we can interpret this as a problem with the locally normalized models that we use here: every prediction is made on an intermediate step of search, and there is no way for the following steps to escape if the prefix is fixed [Murray and Chiang, 2018].

One approach to improving the hypothesis diversity is to give penalties to cases where the hypotheses in the beam are less diverse [Li and Jurafsky, 2016; Vijayakumar et al., 2018]. A simple example of such objective functions is given by

$$\text{score}_d(\mathbf{x}, \mathbf{y}) = \text{score}(\mathbf{x}, \mathbf{y}) - \lambda \cdot dp \quad (5.112)$$

It combines the original model score  $\text{score}(\mathbf{x}, \mathbf{y})$  and a diversity penalty  $dp$ .  $dp$  can be defined in a few different ways. An idea is to penalize hypotheses that are close in the search tree. For example, one can define  $dp$  as the rank of a hypothesis in the set of its siblings that are extended from the same parent hypothesis, and so the beam can spread its members over a larger region of the space of hypotheses [Li and Jurafsky, 2016]. Another way to introduce diversity measures is to consider the differences between the target-side sequences of the hypotheses in the beam. For example, we can define  $dp$  as the average string similarity between a given hypothesis and other hypotheses in the beam [Xiao et al., 2013].

The above idea can also be expressed as constraints imposed on the search procedure. For example, we can constrain the beam to include only the hypotheses that are rooted at

different parents in the last step [Boulanger-Lewandowski et al., 2013]. More precisely, for each hypothesis  $z_{\text{cur}} \in Z_{i-1}$ , we seek the best next-step hypothesis by

$$\hat{z}_{\text{next}} = \underset{z_{\text{next}} \in \text{Extend}(z_{\text{cur}}, V_Y)}{\text{arg max}} \Pr(z_{\text{next}} \cdot \text{tgt} | \mathbf{x}) \quad (5.113)$$

The hypothesis  $\hat{z}_{\text{next}} \in Z_i$  is then added to  $Z_i$ . Note that this is essentially a **sub-space method** that divides a space of hypotheses into sub-spaces of hypotheses, and collects results over these sub-spaces. An intuition behind this method is that different sub-spaces can describe different aspects of the problem, and so we can have diverse solutions.

Another approach to addressing the diversity issue is to perturb beam search by introducing randomly generated hypotheses into the beam [Holtzman et al., 2020a; Wiher et al., 2022]. One common way to do this is to choose some random words for extending a hypothesis, and to add the extended hypotheses to the beam. In general, these words can be sampled from the distribution  $\Pr(\cdot | \mathbf{y}_{<i}, \mathbf{x})$  over the entire vocabulary or its subset. Randomness can also be added to the inputs of a system at test time. For example, one can express an input word as a linear combination of its original embedding and the embedding of a word of a random sequence drawn from the training data [Li et al., 2021b]. In problems having many local minima, adding random “noise” to search procedures is generally helpful, as we can explore more diverse hypotheses and prevent the systems from getting stuck in certain regions of the search space.

Instead of performing search using a single system, we can use multiple systems to obtain diverse hypotheses. These systems can be built on either different architectures or different hidden structures/configurations [He et al., 2018; Shen et al., 2019; Wu et al., 2020a; Sun et al., 2020a]. Although methods of this type do not fall under the search framework that we have been discussing, combining the results from multiple systems is generally helpful. The following section will present a discussion on this issue.

## 5.4.7 Combining Multiple Models

From a machine learning point of view, **ensembling** are methods for addressing modeling issues, not search issues. In this subsection, we discuss these methods because their implementations typically require modifications to the search modules, and we can gain some insight into the resulting system by viewing it from the search perspective.

In machine learning, ensemble methods aim to make better predictions by combining predictions of a number of **constituent systems** or **component systems**. The problem of combining multiple systems has been discussed extensively in times when statistical models emerged in NLP, and is sometimes called **system combination methods** for emphasizing its practical use. For sequence-to-sequence models discussed here, a widely used form of system combination is an average of predictions [Sutskever et al., 2014]. Suppose we have  $K$  sequence-to-sequence models that have been trained. The log probability of the target-side word  $y_i$  given its left context  $\mathbf{y}_{<i}$  and the source-side sequence  $\mathbf{x}$  can be defined by using the

geometric average

$$\log \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}) = \frac{1}{K} \sum_{k=1}^K \log \Pr_k(y_i | \mathbf{y}_{<i}, \mathbf{x}) \quad (5.114)$$

or alternatively by using the arithmetic average

$$\log \Pr(y_i | \mathbf{y}_{<i}, \mathbf{x}) = \log \frac{1}{K} \sum_{k=1}^K \Pr_k(y_i | \mathbf{y}_{<i}, \mathbf{x}) \quad (5.115)$$

where  $\Pr_k(y_i | \mathbf{y}_{<i}, \mathbf{x})$  is the output of the  $k$ -th component system. These forms are so simple that one can implement them for any sequence-to-sequence models without significant modifications to existing systems, and they have been used as the basis of many successful systems in various evaluation tasks [Barrault et al., 2020; Akhbardeh et al., 2021].

A problem with prediction averaging is that all the component systems are required to follow the same basic form of modeling (see Eq. (5.68)) and we need to have access to the probabilities  $\{\Pr_k(y_i | \mathbf{y}_{<i}, \mathbf{x})\}$ . When we have only a set of black-box systems in hand, we need to perform sequence ensembling. A common idea is to vote from the ensemble of the sequences produced by the component systems. For example, one of the simplest ways to do this is **hypothesis selection** [Hildebrand and Vogel, 2008], in which we simply select the “best” sequence from the ensemble using some criterion. An alternative way of sequence ensembling is to regenerate a new sequence differing from any of the original sequences [Matusov et al., 2006; Rosti et al., 2007]. This typically requires a model that represents the sequences into a compact representation (such as a lattice), as well as an additional search pass by which we can find the best output in this new representation of hypotheses (such as lattice search and rescore) [Deoras et al., 2011; Stahlberg et al., 2016; Khayrallah et al., 2017].

Note that the ensembling of sequence-to-sequence models is related to the diversity issue discussed in the previous subsection. It is often thought that component systems need to be diverse for a better ensembling result, and so we need to build these systems in some way that we can make them different [Sutskever et al., 2014; Zhou et al., 2017]. One of the most popular methods is **checkpoint ensembling**. It takes a number of copies of a model at different checkpoints during training, and combines these model copies via prediction averaging. This method can be useful for alleviating the overfitting problem in practice. Also, different models can be created from a base model under different settings. For example, we can build models with different numbers of parameters on the basis of a backbone model. A more general approach is to take models based on different architectures, although this is at the expense of more development effort.

Another way to view sequence ensembling is that it provides a two-pass search scheme. In the first pass of search, multiple systems are used to perform inference individually. Each of these systems has its own bias for modeling and search, and explores different regions of the search space. A hypothesis explored by one system might not be seen and evaluated by

another system. The result of this pass is a diverse ensemble of hypotheses that are “optimal” from some perspectives. In the second pass of search, we use this ensemble to define a new space of hypotheses, and use a fine-grained model to search for the final result.

### 5.4.8 More Search Objectives

In this subsection, we consider more objective functions that can be applied to the search problem.

#### 1. Search with Future Scores

Most of the algorithms described in this subsection can be viewed as some optimizations of best-first search algorithms [Meister et al., 2020]. As another example of best-first search, **A\* search** is widely considered to be a good solution to the general search problem. Vanilla A\* search requires that all states of search are sorted in every search step, which is intractable in our problems. We therefore still consider beam search and greedy search for our discussion, but use an A\* search-like objective function instead. Specifically, given a search state  $(\mathbf{x}, \mathbf{y}_{\leq i})$ , the A\* search-like objective function can be defined as

$$\text{score}_{A^*}(\mathbf{x}, \mathbf{y}_{\leq i}) = g(\mathbf{x}, \mathbf{y}_{\leq i}) + h(\mathbf{x}, \mathbf{y}_{\leq i}) \quad (5.116)$$

Here  $g(\mathbf{x}, \mathbf{y}_{\leq i})$  is the reward of the path from the start state to  $(\mathbf{x}, \mathbf{y}_{\leq i})$ , and  $h(\mathbf{x}, \mathbf{y}_{\leq i})$  is the estimated reward of the “optimal” path from  $(\mathbf{x}, \mathbf{y}_{\leq i})$  to the final goal. Because  $g(\mathbf{x}, \mathbf{y}_{\leq i})$  and  $h(\mathbf{x}, \mathbf{y}_{\leq i})$  can have arbitrary forms, this framework is very general. For example, if we define

$$g(\mathbf{x}, \mathbf{y}_{\leq i}) = \text{score}(\mathbf{x}, \mathbf{y}_{\leq i}) \quad (5.117)$$

$$h(\mathbf{x}, \mathbf{y}_{\leq i}) = 0 \quad (5.118)$$

then  $\text{score}_{A^*}(\mathbf{x}, \mathbf{y})$  is exactly the same as the objective functions discussed previously.

To make full use of this formulation, it seems natural to seek a function of future reward or future cost. Ideally, we would like  $h(\mathbf{x}, \mathbf{y}_{\leq i})$  to be able to compute how much additional reward we can obtain if we extend  $(\mathbf{x}, \mathbf{y}_{\leq i})$  to the best complete hypothesis. This is, however, intractable because we need to explore all the hypotheses extended from  $(\mathbf{x}, \mathbf{y}_{\leq i})$  and find the best one. It is common practice to use a computationally cheaper model analogous to the real future reward model. Conventional approaches rely on heuristics to define  $h(\mathbf{x}, \mathbf{y}_{\leq i})$  [Koehn et al., 2007], such as estimating the weights of the words that could be further generated. These heuristics can be generalized to the knowledge of the model design of sequence-to-sequence systems [He et al., 2017; Zheng et al., 2018]. A more general approach is to use a value-based treatment of the problem [Ren et al., 2017; Li et al., 2017a; Leblond et al., 2021]. We can develop a policy that learns to predict the distribution of  $y_i$  given  $\mathbf{x}$  and  $\mathbf{y}_{< i}$ , and a **value function** for this policy that learns to predict future rewards. Eq. (5.116) can therefore be interpreted as a linear combination of the policy score of  $(\mathbf{x}, \mathbf{y}_{\leq i})$  and the corresponding value. Such a treatment of search objectives falls into the framework of **value-based search**, and has been successfully employed in reinforcement learning [Silver et al., 2017].

## 2. Search with Language Models

For a long time, language models played an important role in text generation tasks. For example, statistical machine translation systems and automatic speech recognition systems typically rely on large  $n$ -gram language models to produce fluent texts. While modern sequence-to-sequence models are not required to have separate language models, applying them to sequence-to-sequence search still makes intuitive sense for machine translation and related problems.

Following the convention that a language model can be treated as a feature of a log-linear (or linear) model [Och and Ney, 2002], the language model-augmented objective can be defined as

$$\text{score}_{\text{lm}}(\mathbf{x}, \mathbf{y}) = \log \Pr(\mathbf{y}|\mathbf{x}) + \lambda \cdot \log \Pr(\mathbf{y}) \quad (5.119)$$

This formulation does not involve length reward and normalization terms, but either of them can be easily used as an additional feature of the model. In general, the language model  $\Pr(\mathbf{y})$  is trained solely on target-side sequences, enabling the use of large-scale monolingual data in sequence-to-sequence models [Gulcehre et al., 2017]. Interestingly, it has been found that current sequence-to-sequence models are strong language models themselves if they are trained sufficiently, and a better way to make use of target-side data might be to use it to create synthetic data, called **data augmentation**. An example of this is **back translation** in which we use a backward translation system to translate target-side sentences to source-side sentences, and then use this synthetic bilingual data as additional data for training a forward translation system [Sennrich et al., 2016a; Edunov et al., 2018]. In many tasks, such a simple method can achieve significant improvements in translation quality, but this result questions the necessity of using additional language models in neural machine translation.

Note that the model of Eq. (5.119) depends on our choice for the coefficient  $\lambda$ . For machine translation, we are usually interested in a positive value of  $\lambda$  so that our system can produce more fluent texts. By contrast, a negative value of  $\lambda$  means that we want some output that is less frequent. For example, if  $\lambda = -1$ , then Eq. (5.119) can be written as the point-wise mutual information of  $\mathbf{x}$  and  $\mathbf{y}$

$$\begin{aligned} \text{score}_{\text{lm}}(\mathbf{x}, \mathbf{y}) &= \log \Pr(\mathbf{y}|\mathbf{x}) - \log \Pr(\mathbf{y}) \\ &= \log \frac{\Pr(\mathbf{x}, \mathbf{y})}{\Pr(\mathbf{x}) \cdot \Pr(\mathbf{y})} \end{aligned} \quad (5.120)$$

This scoring function has been shown to be useful for generating more diverse outputs for neural conversation systems [Li et al., 2016].

## 3. Minimum Bayes Risk Search

So far, our discussion of search objectives has focused on the use of the decision rule of choosing the highest score hypothesis, called **maximum a posteriori (MAP)** search<sup>14</sup>. An

<sup>14</sup>In statistics, MAP is a method for inference of the parameters of a statistical model. Suppose we have a model that describes the distribution of a variable  $x$  and the model is parameterized by  $\theta$ . MAP seeks the optimal value of

assumption behind this method is that the posterior probability  $\Pr(\mathbf{y}|\mathbf{x})$  (or the model score  $\text{score}(\mathbf{x}, \mathbf{y})$ ) correlates with the true quality of outputs. In practice, this assumption leads to several useful properties, e.g., the search system is easy to implement, and the objective of search is consistent with that of training. However, there are some shortcomings with MAP search, which causes researchers to consider more powerful methods. One problem with MAP search is that the objective does not reflect the way one evaluates the system. The metrics used in end-to-end evaluation of a system may have very different forms from  $\Pr(\mathbf{y}|\mathbf{x})$ . A second problem is that MAP is just a special case of the Bayesian treatment of determining posterior probabilities. It provides a point estimate of  $\theta$  with no uncertainty measure, and is sometimes overconfident. In some applications, sequence-to-sequence models spread too much probability mass across many different hypotheses [Ott et al., 2018a], and MAP may not describe the major portion of the distribution.

Here we consider **minimum Bayes risk (MBR)** search that provides ways to introduce evaluation measures into search, as well as ways to make use of the distributions over hypotheses. The MBR method assumes a risk function on a pair of sequences, denoted by  $R(\mathbf{y}, \mathbf{y}_r)$ . It computes the cost of replacing  $\mathbf{y}_r$  with  $\mathbf{y}$  in terms of some evaluation metric. For example, we can define the risk score to be  $1 - \text{BLEU}$  for machine translation. Then, the risk for  $\mathbf{y}$  on a set of sequences  $\Omega$  is given by the expectation of  $R(\mathbf{y}, \mathbf{y}_r)$  with respect to the distribution  $\Pr(\mathbf{y}_r|\mathbf{x})$

$$\begin{aligned} \text{Risk}(\mathbf{y}) &= \mathbb{E}_{\mathbf{y}_r \sim \Pr(\mathbf{y}_r|\mathbf{x})} R(\mathbf{y}, \mathbf{y}_r) \\ &= \sum_{\mathbf{y}_r \in \Omega} R(\mathbf{y}, \mathbf{y}_r) \cdot \Pr(\mathbf{y}_r|\mathbf{x}) \end{aligned} \quad (5.124)$$

However, the summation over all possible target-side sequences is computationally infeasible. We therefore define  $\Omega$  to be the  $k$ -best outputs or sampled outputs of a system [Eikema and Aziz, 2020], denoted by  $\Omega_{\text{system}}$ . Then, we take  $\text{score}(\mathbf{x}, \mathbf{y}) = -\text{Risk}(\mathbf{y})$  and obtain the

---

$\theta$  by maximizing the probability of  $\theta$  given  $x$ , written as

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta} \Pr(\theta|x) \quad (5.121)$$

$\hat{\theta}_{\text{MAP}}$  is also called the **mode** of the posterior distribution of  $\theta$ . For the MAP search problem here, we simply denote  $\theta$  by  $\mathbf{y}$  and seek the mode of  $\Pr(\mathbf{y}|\mathbf{x})$ .

As a Bayesian method, we can re-express the above equation using the Bayes' rule

$$\begin{aligned} \hat{\theta}_{\text{MAP}} &= \arg \max_{\theta} \frac{\Pr(x|\theta) \cdot \Pr(\theta)}{\Pr(x)} \\ &= \arg \max_{\theta} \Pr(x|\theta) \cdot \Pr(\theta) \end{aligned} \quad (5.122)$$

where  $\theta$  is treated as a variable having a prior distribution  $\Pr(\theta)$ .

By contrast, MLE directly maximizes the likelihood function  $\Pr(x|\theta)$

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} \Pr(x|\theta) \quad (5.123)$$

Thus, the MAP result can be viewed as an estimation of  $\theta$  that considers both MLE of  $x$  given  $\theta$  and the prior of  $\theta$ . Note that MAP and MLE will be equivalent if  $\Pr(\theta)$  is a uniform distribution.

following objective for MBR search

$$\begin{aligned}\hat{\mathbf{y}} &= \arg \max_{\mathbf{y}} -\text{Risk}(\mathbf{y}) \\ &= \arg \min_{\mathbf{y}} \sum_{\mathbf{y}_r \in \Omega_{\text{system}}} R(\mathbf{y}, \mathbf{y}_r) \cdot \text{Pr}(\mathbf{y}_r | \mathbf{x})\end{aligned}\quad (5.125)$$

This model is very general and applies to a wide range of NLP problems in which one needs to search for an optimal hypothesis in a large set of candidates [Goodman, 1996b; Goel and Byrne, 2000; Kumar and Byrne, 2004b]. It allows for flexible forms of risk functions, for instance having various factors considered in evaluating hypotheses. MBR search has recently been of interest to NLP researchers as they are found to be effective in eliminating the biases caused by MAP search [Müller and Sennrich, 2021; Freitag et al., 2022]. In addition to providing a formulation of search objectives, MBR methods can be used for training sequence-to-sequence models, and are thought to be solutions to the discrepancy issue between objectives of training and evaluation [Shen et al., 2016].

## 5.5 Summary

In this chapter, we attempted to provide an overview of sequence-to-sequence modeling which can serve as the basis for many NLP systems. Sequence-to-sequence modeling is a very rich area of research, and has been widely discussed in different disciplines, even beyond NLP. This chapter is not a review of all the literature on this subject (this would be a big project), but focuses on some of the core methods and ideas. We started with an introduction of sequence-to-sequence problems, as well as the encoder-decoder architecture which lays the foundations for most of the state-of-the-art sequence-to-sequence systems. As an illustration of the application of this architecture, we considered the problem of neural machine translation, and built a simple neural machine translation model using the basic knowledge we have learned so far.

We also presented the attention mechanism and a series of refinements. If we look back to the past few years, we will find that exploring attention models is the next natural step in developing sequence-to-sequence models. While these models are well known for their application and impressive performance in machine translation, they have dominated the NLP community. There is also great interest in attention models in some other sub-fields of AI, such as computer vision [Borji and Itti, 2012; Xu et al., 2015; Jaderberg et al., 2015] and speech processing [Chorowski et al., 2015; Chan et al., 2016; Bahdanau et al., 2016]. The result is that the past few years were an exciting time for people in these areas.

Sequence-to-sequence models are so successful that we try to put everything in the same pocket. Not only have we developed powerful sequence-to-sequence models to deal with very general problems, but current research is forced to be unifying. An example is that Transformer, a self-attention-based sequence-to-sequence model, has become one of the fundamental models for many tasks ranging over different types of data, from textual to visual and acoustic data. It can even be extended to deal with multimodal problems which are sometimes more challenging.

This makes things more interesting and exciting: an improvement to one model can be used to improve systems in a variety of tasks. And we are seeing a significant change in our research paradigm in which the NLP and machine learning fields are marrying and results in NLP research are becoming more influential. However, on the other side of the coin is that we are making much room for some of the problems but leaving less room for the others. In recent NLP conferences, we can see many, many papers talking about how to train big sequence-to-sequence models and apply them to different text generation tasks, but there are a relatively small number of papers on parsing. There have always been debates on this over the past few decades, for example, what and how much prior knowledge do we need to build an NLP system? [Church, 2011; See, 2018] Getting involved in such debates is simply beyond the discussions in this chapter. Fortunately, NLP research promises to continue to be diverse and active, and we can always hear and learn from both sides of the debates. For example, there are interesting findings that the neural sequence models can learn some linguistic properties from data, and linguistic structures can help system design. In Chapter 6, we will see a few examples.

The “bias” of research focus also exists on the machine learning side of problem-solving. For example, for sequence-to-sequence problems discussed here, recent years have witnessed a drastic increase of interest in model design and training methods, but only a relatively small group of people discuss the search problem. While search is a classical problem in AI and plays an important role in practical systems [Russell and Norvig, 2010], it is even not discussed in recent tutorials and surveys in NLP. This motivates us to write a section on this subject so that we can have a more complete picture of the problem. However, our general discussion does not cover all aspects of the search problem. A topic we left out is efficiency [Birch et al., 2018; Heafield et al., 2021]. While this chapter includes some discussions on the efficiency issue, such as stopping criteria of search algorithms, efficient methods are a wide-ranging topic and are generally dependent on model architectures. A more detailed discussion of them can be found in Chapter 6. Another topic that one may be interested in is **constrained search** in which constraints are imposed on the search process [Hokamp and Liu, 2017; Anderson et al., 2017]. In general, these constraints come from our prior knowledge or interactions with users. For example, constrained search has been used to enforce term translation constraints on machine translation [Hasler et al., 2018; Post and Vilar, 2018].

One last note on limitations of this chapter. The formulation of the general sequence-to-sequence problem described here is based on the left-to-right factorization of  $\Pr(y|x)$ , resulting in an autoregressive model. One limitation of this formulation is that each prediction at some step depends only on the preceding words, and so the model cannot access the right context. To make use of the right context of a word, a simple approach is to build another model that performs right-to-left generation. The left-to-right and right-to-left models can then be combined to generate a better output sequence [Liu et al., 2016a; Hoang et al., 2017; Zhang et al., 2018b; 2020a]. An alternative approach is given by **non-autoregressive generation** or **non-autoregressive decoding** in which the constraint of autoregressive generation is removed and each word prediction is conditioned on the global context [Gu et al., 2018; Ghazvininejad et al., 2019; Lee et al., 2020]. A nice property of non-autoregressive generation is the possibility

---

of system speed-up, since all the words in a sequence can be generated in parallel and we can do this efficiently using GPUs.



# Chapter 6

## Transformers

So far we have discussed several basic models for solving sequence-to-sequence problems. We now explore a new class of models which are based on a powerful architecture, called **Transformer**. Transformers differ in several ways from the models given in Chapters 4 and 5. First, they do not depend on recurrent or convolutional neural networks for modeling sequences of words, but use only attention mechanisms and feed-forward neural networks. Second, the use of self-attention in Transformers makes it easier to deal with global contexts and dependencies among words. Third, Transformers are very flexible architectures and can be easily modified to accommodate different tasks. The past few years have seen the rise of Transformers not only in NLP but also in several other fields. As Transformers and their variants continue to mature, these models are playing an increasingly important role in the research and application of artificial intelligence.

In this chapter, we will discuss the core ideas of Transformers. We will begin our discussion by looking at the standard Transformer architecture. Then we will look at some notable developments, such as improvements to the basic architecture and efficient methods. We will also present several applications in which Transformer models have been extensively used. However, the discussion of Transformer is a wide-ranging topic, and there have many, many related papers. This chapter is not intended to provide a comprehensive survey of the literature but a collection of selected topics that NLP people may be interested in.

### 6.1 The Basic Model

Here we consider the model presented in [Vaswani et al. \[2017\]](#)'s work. We start by considering the Transformer architecture and discuss the details of the sub-models subsequently.

#### 6.1.1 The Transformer Architecture

Figure 6.1 shows the standard Transformer model which follows the general encoder-decoder framework. A Transformer encoder comprises a number of stacked **encoding layers** (or **encoding blocks**). Each encoding layer has two different sub-layers (or sub-blocks), called the self-attention sub-layer and the feed-forward neural network (FFN) sub-layer. Suppose

we have a source-side sequence  $\mathbf{x} = x_1 \dots x_m$  and a target-side sequence  $\mathbf{y} = y_1 \dots y_n$ . The input of an encoding layer is a sequence of  $m$  vectors  $\mathbf{h}_1 \dots \mathbf{h}_m$ , each having  $d_{\text{model}}$  dimensions (or  $d$  dimensions for simplicity). We follow the notation adopted in the previous chapters, using  $\mathbf{H} \in \mathbb{R}^{m \times d}$  to denote these input vectors<sup>1</sup>. The self-attention sub-layer first performs a self-attention operation  $\text{Att}_{\text{self}}(\cdot)$  on  $\mathbf{H}$  to generate an output  $\mathbf{C}$ :

$$\mathbf{C} = \text{Att}_{\text{self}}(\mathbf{H}) \quad (6.1)$$

Here  $\mathbf{C}$  is of the same size as  $\mathbf{H}$ , and can thus be viewed as a new representation of the inputs. Then, a residual connection and a layer normalization unit are added to the output so that the resulting model is easier to optimize.

The original Transformer model employs the **post-norm** structure where a residual connection is created before layer normalization is performed, like this

$$\mathbf{H}_{\text{self}} = \text{LNorm}(\mathbf{C} + \mathbf{H}) \quad (6.2)$$

where the addition of  $\mathbf{H}$  denotes the residual connection, and  $\text{LNorm}(\cdot)$  denotes the layer normalization function. Substituting Eq. (6.1) into Eq. (6.2), we obtain the form of the self-attention sub-layer

$$\begin{aligned} \text{Layer}_{\text{self}}(\mathbf{H}) &= \mathbf{H}_{\text{self}} \\ &= \text{LNorm}(\text{Att}_{\text{self}}(\mathbf{H}) + \mathbf{H}) \end{aligned} \quad (6.3)$$

The definitions of  $\text{LNorm}(\cdot)$  and  $\text{Att}_{\text{self}}(\cdot)$  have been given in Chapters 2 and 5, and we will also discuss them later in the section.

The FFN sub-layer takes  $\mathbf{H}_{\text{self}}$  and outputs a new representation  $\mathbf{H}_{\text{ffn}} \in \mathbb{R}^{m \times d}$ . It has the same form as the self-attention sub-layer, with the attention function replaced by the FFN function, given by

$$\begin{aligned} \text{Layer}_{\text{ffn}}(\mathbf{H}_{\text{self}}) &= \mathbf{H}_{\text{ffn}} \\ &= \text{LNorm}(\text{FFN}(\mathbf{H}_{\text{self}}) + \mathbf{H}_{\text{self}}) \end{aligned} \quad (6.4)$$

Here  $\text{FFN}(\cdot)$  could be any feed-forward neural networks with non-linear activation functions. The most common structure of  $\text{FFN}(\cdot)$  is a two-layer network involving two linear transformations and a ReLU activation function between them.

For deep models, we can stack the above neural networks. Let  $\mathbf{H}^l$  be the output of layer  $l$ . Then, we can express  $\mathbf{H}^l$  as a function of  $\mathbf{H}^{l-1}$ . We write this as a composition of two

---

<sup>1</sup>Provided  $\mathbf{h}_j \in \mathbb{R}^d$  is a row vector, we have  $\mathbf{H} = \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_m \end{bmatrix}$ .

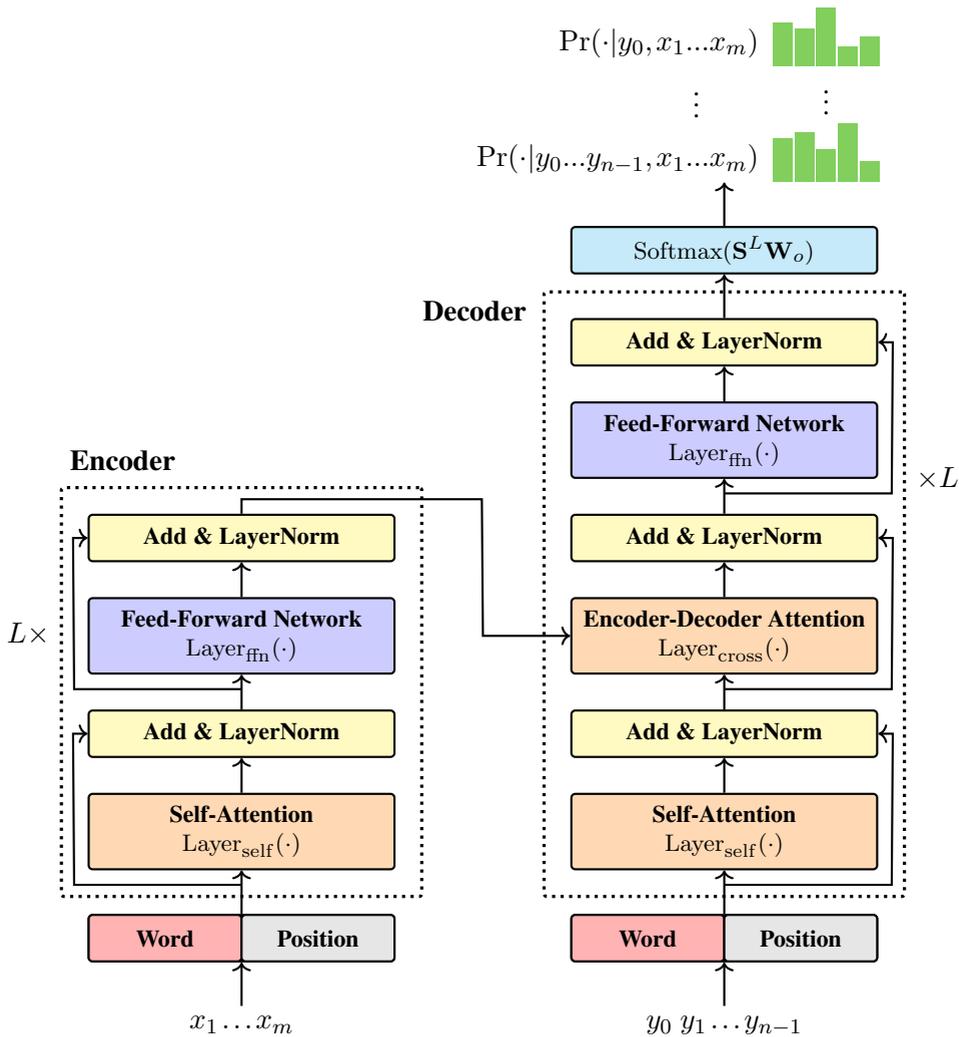


Figure 6.1: The Transformer architecture [Vaswani et al., 2017]. There are  $L$  stacked layers on each of the encoder and decoder sides. An encoding layer comprises a self-attention sub-layer and an FFN sub-layer. Both of these sub-layers share the same structure which involves a core function (either  $\text{Layer}_{\text{self}}(\cdot)$  or  $\text{Layer}_{\text{ffn}}(\cdot)$ ), followed by a residual connection and a layer normalization unit. Each decoding layer has a similar architecture with the encoding layers, but with an additional encoder-decoder attention sub-layer sandwiched between the self-attention and FFN sub-layers. As with most sequence-to-sequence models, Transformer takes  $x_1 \dots x_m$  and  $y_0 \dots y_{i-1}$  for predicting  $y_i$ . The representation of an input word comprises a sum of a word embedding and a positional embedding. The distributions  $\{\text{Pr}(\cdot | y_0 \dots y_{i-1}, x_1 \dots x_m)\}$  are generated in sequence by a Softmax layer, which operates on a linear transformation of the output from the last decoding layer.

sub-layers

$$\mathbf{H}^l = \text{Layer}_{\text{ffn}}(\mathbf{H}_{\text{self}}^l) \quad (6.5)$$

$$\mathbf{H}_{\text{self}}^l = \text{Layer}_{\text{self}}(\mathbf{H}^{l-1}) \quad (6.6)$$

If there are  $L$  encoding layers, then  $\mathbf{H}^L$  will be the output of the encoder. In this case,  $\mathbf{H}^L$  can be viewed as a representation of the input sequence that is learned by the Transformer encoder.  $\mathbf{H}^0$  denotes the input of the encoder. In recurrent and convolutional models,  $\mathbf{H}^0$  can simply be word embeddings of the input sequence. Transformer takes a different way of representing the input words, and encodes the positional information explicitly. In Section 6.1.2 we will discuss the embedding model used in Transformers.

The Transformer decoder has a similar structure as the Transformer encoder. It comprises  $L$  stacked **decoding layers** (or **decoding blocks**). Let  $\mathbf{S}^l$  be the output of the  $l$ -th decoding layer. We can formulate a decoding layer by using the following equations

$$\mathbf{S}^l = \text{Layer}_{\text{ffn}}(\mathbf{S}_{\text{cross}}^l) \quad (6.7)$$

$$\mathbf{S}_{\text{cross}}^l = \text{Layer}_{\text{cross}}(\mathbf{H}^L, \mathbf{S}_{\text{self}}^{l-1}) \quad (6.8)$$

$$\mathbf{S}_{\text{self}}^l = \text{Layer}_{\text{self}}(\mathbf{S}^{l-1}) \quad (6.9)$$

Here there are three decoder sub-layers. The self-attention and FFN sub-layers are the same as those used in the encoder.  $\text{Layer}_{\text{cross}}(\cdot)$  denotes a cross attention sub-layer (or encoder-decoder sub-layer) which models the transformation from the source-side to the target-side. In Section 6.1.6 we will see that  $\text{Layer}_{\text{cross}}(\cdot)$  can be implemented using the same function as  $\text{Layer}_{\text{self}}(\cdot)$ .

The Transformer decoder outputs a distribution over a vocabulary  $V_y$  at each target-side position. This is achieved by using a softmax layer that normalizes a linear transformation of  $\mathbf{S}^L$  to distributions of target-side words. To do this, we map  $\mathbf{S}^L$  to an  $n \times |V_y|$  matrix  $\mathbf{O}$  by

$$\mathbf{O} = \mathbf{S}^L \cdot \mathbf{W}_o \quad (6.10)$$

where  $\mathbf{W}_o \in \mathbb{R}^{d \times |V_y|}$  is the parameter matrix of the linear transformation.

Then, the output of the Transformer decoder is given in the form

$$\begin{aligned} \begin{bmatrix} \Pr(\cdot | y_0, \mathbf{x}) \\ \vdots \\ \Pr(\cdot | y_0 \dots y_{n-1}, \mathbf{x}) \end{bmatrix} &= \text{Softmax}(\mathbf{O}) \\ &= \begin{bmatrix} \text{Softmax}(\mathbf{o}_1) \\ \vdots \\ \text{Softmax}(\mathbf{o}_n) \end{bmatrix} \end{aligned} \quad (6.11)$$

where  $\mathbf{o}_i$  denotes the  $i$ -th row vector of  $\mathbf{O}$ , and  $y_0$  denotes the start symbol  $\langle \text{SOS} \rangle$ . Under this model, the probability of  $\mathbf{y}$  given  $\mathbf{x}$  can be defined as usual,

$$\log \Pr(\mathbf{y} | \mathbf{x}) = \sum_{i=1}^n \log \Pr(y_i | y_0 \dots y_{i-1}, \mathbf{x}) \quad (6.12)$$

This equation resembles the general form of language modeling: we predict the word at

time  $i$  given all of the words up to time  $i - 1$ . Therefore, the input of the Transformer decoder is shifted one word left, that is, the input is  $y_0 \dots y_{n-1}$  and the output is  $y_1 \dots y_n$ .

The Transformer architecture discussed above has several variants which have been successfully used in different fields of NLP. For example, we can use a Transformer encoder to represent texts (call it the **encoder-only architecture**), can use a Transformer decoder to generate texts (call it the **decoder-only architecture**), and can use a standard encoder-decoder Transformer model to transform an input sequence to an output sequence. In the rest of this chapter, most of the discussion is independent of the particular choice of application, and will be mostly focused on the encoder-decoder architecture. In Section 6.5, we will see applications of the encoder-only and decoder-only architectures.

### 6.1.2 Positional Encoding

In their original form, both FFNs and attention models used in Transformer ignore an important property of sequence modeling, which is that the order of the words plays a crucial role in expressing the meaning of a sequence. This means that the encoder and decoder are insensitive to the positional information of the input words. A simple approach to overcoming this problem is to add positional encoding to the representation of each word of the sequence. More formally, a word  $x_j$  can be represented as a  $d$ -dimensional vector

$$\mathbf{x}p_j = \mathbf{x}_j + \text{PE}(j) \quad (6.13)$$

Here  $\mathbf{x}_j \in \mathbb{R}^d$  is the embedding of the word which can be obtained by using the word embedding models, as described Chapter 3.  $\text{PE}(j) \in \mathbb{R}^d$  is the representation of the position  $j$ . Vanilla Transformer employs the sinusoidal positional encoding models which we write in the form

$$\text{PE}(i, 2k) = \sin\left(i \cdot \frac{1}{10000^{2k/d}}\right) \quad (6.14)$$

$$\text{PE}(i, 2k + 1) = \cos\left(i \cdot \frac{1}{10000^{2k/d}}\right) \quad (6.15)$$

where  $\text{PE}(i, k)$  denotes the  $k$ -th entry of  $\text{PE}(i)$ . The idea of positional encoding is to distinguish different positions using continuous systems. Here we use the sine and cosine functions with different frequencies. The interested reader can refer to Chapter 4 to see that such a method can be interpreted as a carrying system. Because the encoding is based on individual positions, it is also called **absolute positional encoding**. In Section 6.3.1 we will see an improvement to this method.

Once we have the above embedding result,  $\mathbf{x}p_1 \dots \mathbf{x}p_m$  is taken as the input to the Transformer encoder, that is,

$$\mathbf{H}_0 = \begin{bmatrix} \mathbf{x}p_1 \\ \vdots \\ \mathbf{x}p_m \end{bmatrix} \quad (6.16)$$

Similarly, we can also define the input on the decoder side.

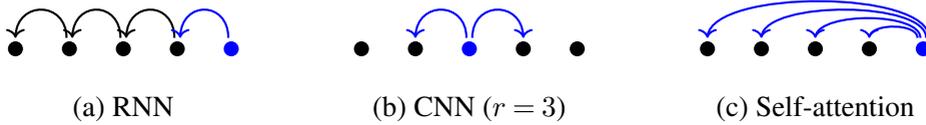


Figure 6.2: Information flows in recurrent, convolutional and self-attention models, shown as arrow lines between positions.

### 6.1.3 Multi-head Self-attention

The use of self-attention is perhaps one of the most significant advances in sequence-to-sequence models. It attempts to learn and make use of direct interactions between each pair of inputs. From a representation learning perspective, self-attention models assume that the learned representation at position  $i$  (denoted by  $\mathbf{c}_i$ ) is a weighted sum of the inputs over the sequence. The output  $\mathbf{c}_i$  is thus given by

$$\mathbf{c}_i = \sum_{j=1}^m \alpha_{i,j} \mathbf{h}_j \quad (6.17)$$

where  $\alpha_{i,j}$  indicates how strong the input  $\mathbf{h}_i$  is correlated with the input  $\mathbf{h}_j$ . We thus can view  $\mathbf{c}_i$  as a representation of the global context at position  $i$ .  $\alpha_{i,j}$  can be defined in different ways if one considers different attention models. Here we use the scaled dot-product attention function to compute  $\alpha_{i,j}$ , as follows

$$\begin{aligned} \alpha_{i,j} &= \text{Softmax}(\mathbf{h}_i \mathbf{h}_j^T / \beta) \\ &= \frac{\exp(\mathbf{h}_i \mathbf{h}_j^T / \beta)}{\sum_{k=1}^m \exp(\mathbf{h}_i \mathbf{h}_k^T / \beta)} \end{aligned} \quad (6.18)$$

where  $\beta$  is a scaling factor and is set to  $\sqrt{d}$ .

Compared with conventional recurrent and convolutional models, an advantage of self-attention models is that they shorten the computational “distance” between two inputs. Figure 6.2 illustrates the information flow in these models. We see that, given the input at position  $i$ , self-attention models can directly access any other input. By contrast, recurrent and convolutional models might need two or more jumps to see the whole sequence.

We can have a more general view of self-attention by using the QKV attention model.

Suppose we have a sequence of  $\kappa$  queries  $\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_\kappa \end{bmatrix}$ , and a sequence of  $\psi$  key-value pairs ( $\mathbf{K} =$

$\begin{bmatrix} \mathbf{k}_1 \\ \vdots \\ \mathbf{k}_\psi \end{bmatrix}$ ,  $\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_\psi \end{bmatrix}$ ). The output of the model is a sequence of vectors, each corresponding to

a query. The form of the QKV attention is given by

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V} \quad (6.19)$$

We can write the output of the QKV attention model as a sequence of row vectors

$$\begin{aligned} \mathbf{C} &= \begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_\kappa \end{bmatrix} \\ &= \text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \end{aligned} \quad (6.20)$$

To apply this equation to self-attention, we simply have

$$\mathbf{H}^q = \mathbf{H}\mathbf{W}^q \quad (6.21)$$

$$\mathbf{H}^k = \mathbf{H}\mathbf{W}^k \quad (6.22)$$

$$\mathbf{H}^v = \mathbf{H}\mathbf{W}^v \quad (6.23)$$

where  $\mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v \in \mathbb{R}^{d \times d}$  represents linear transformations of  $\mathbf{H}$ .

By considering Eq. (6.1), we then obtain

$$\begin{aligned} \mathbf{C} &= \text{Att}_{\text{self}}(\mathbf{H}) \\ &= \text{Att}_{\text{qkv}}(\mathbf{H}^q, \mathbf{H}^k, \mathbf{H}^v) \\ &= \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}}\right)\mathbf{H}^v \end{aligned} \quad (6.24)$$

Here  $\text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}}\right)$  is an  $m \times m$  matrix in which each row represents a distribution over  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$ , that is

$$\text{row } i = \begin{bmatrix} \alpha_{i,1} & \dots & \alpha_{i,m} \end{bmatrix} \quad (6.25)$$

We can improve the above self-attention model by using a technique called **multi-head attention**. This method can be motivated from the perspective of learning from multiple lower-dimensional feature sub-spaces, which projects a feature vector onto multiple sub-spaces and learns feature mappings on individual sub-spaces. Specifically, we project the whole of the input space into  $\tau$  sub-spaces (call them **heads**), for example, we transform  $\mathbf{H} \in \mathbb{R}^{m \times d}$  into  $\tau$  matrices of size  $m \times \frac{d}{\tau}$ , denoted by  $\{\mathbf{H}_1^{\text{head}}, \dots, \mathbf{H}_\tau^{\text{head}}\}$ . The attention model is then run  $\tau$  times, each time on a head. Finally, the outputs of these model runs are concatenated, and transformed by a linear projection. This procedure can be expressed by

$$\mathbf{C} = \text{Merge}(\mathbf{C}_1^{\text{head}}, \dots, \mathbf{C}_\tau^{\text{head}})\mathbf{W}_c \quad (6.26)$$

$$(6.27)$$

For each head  $h$ ,

$$\mathbf{C}_h^{\text{head}} = \text{Softmax}\left(\frac{\mathbf{H}_h^q [\mathbf{H}_h^k]^\top}{\sqrt{d}}\right) \mathbf{H}_h^v \quad (6.28)$$

$$\mathbf{H}_h^q = \mathbf{H} \mathbf{W}_h^q \quad (6.29)$$

$$\mathbf{H}_h^k = \mathbf{H} \mathbf{W}_h^k \quad (6.30)$$

$$\mathbf{H}_h^v = \mathbf{H} \mathbf{W}_h^v \quad (6.31)$$

Here  $\text{Merge}(\cdot)$  is the concatenation function, and  $\text{Att}_{\text{QKV}}(\cdot)$  is the attention function described in Eq. (6.20).  $\mathbf{W}_h^q, \mathbf{W}_h^k, \mathbf{W}_h^v \in \mathbb{R}^{d \times \frac{d}{\tau}}$  are the parameters of the projections from a  $d$ -dimensional space to a  $\frac{d}{\tau}$ -dimensional space for the queries, keys, and values. Thus,  $\mathbf{H}_h^q, \mathbf{H}_h^k, \mathbf{H}_h^v$ , and  $\mathbf{C}_h^{\text{head}}$  are all  $m \times \frac{d}{\tau}$  matrices.  $\text{Merge}(\mathbf{C}_1^{\text{head}}, \dots, \mathbf{C}_\tau^{\text{head}})$  produces an  $m \times d$  matrix. It is then transformed by a linear mapping  $\mathbf{W}_c \in \mathbb{R}^{d \times d}$ , leading to the final result  $\mathbf{C} \in \mathbb{R}^{d \times d}$ .

While the notation here seems somewhat tedious, it is convenient to implement multi-head models using various deep learning toolkits. A common method in Transformer-based systems is to store inputs from all the heads in data structures called tensors, so that we can make use of parallel computing resources to have efficient systems. A more general discussion of the QKV attention and multi-head attention models can be found in Chapter 5.

### 6.1.4 Layer Normalization

Layer normalization provides a simple and effective means to make the training of neural networks more stable by standardizing the activations of the hidden layers in a layer-wise manner. As introduced in Ba et al. [2016]’s work, given a layer’s output  $\mathbf{h} \in \mathbb{R}^d$ , the layer normalization method computes a standardized output  $\text{LNorm}(\mathbf{h}) \in \mathbb{R}^d$  by

$$\text{LNorm}(\mathbf{h}) = \mathbf{g} \odot \frac{\mathbf{h} - \boldsymbol{\mu}}{\boldsymbol{\sigma} + \epsilon} + \mathbf{b} \quad (6.32)$$

Here  $\boldsymbol{\mu} \in \mathbb{R}^d$  and  $\boldsymbol{\sigma} \in \mathbb{R}^d$  are the mean and standard derivation of the activations. Let  $h_k$  be the  $k$ -th dimension of  $\mathbf{h}$ .  $\boldsymbol{\mu}$  and  $\boldsymbol{\sigma}$  are given by

$$\boldsymbol{\mu} = \frac{1}{d} \cdot \sum_{k=1}^d h_k \quad (6.33)$$

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{d} \cdot \sum_{k=1}^d (h_k - \mu)^2} \quad (6.34)$$

Here  $\mathbf{g} \in \mathbb{R}^d$  and  $\mathbf{b} \in \mathbb{R}^d$  are the rescaling and bias terms. They can be treated as parameters of layer normalization, whose values are to be learned together with other parameters of the Transformer model. The addition of  $\epsilon$  to  $\boldsymbol{\sigma}$  is used for the purpose of numerical stability. In general,  $\epsilon$  is chosen to be a small number.

We illustrate the layer normalization method for the hidden states of an encoder in the

following example (assume that  $m = 4$ ,  $d = 3$ ,  $\mathbf{g} = \mathbf{1}$ ,  $\mathbf{b} = \mathbf{0}$ , and  $\epsilon = 0.1$ ).

$$\begin{array}{l} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \\ \mathbf{h}_4 \end{array} \begin{bmatrix} 1 & 1 & 2 \\ 0.9 & 0.9 & 0 \\ 0.7 & 0.8 & 0 \\ 3 & 1 & 7 \end{bmatrix} \begin{array}{l} \mu = 1.3, \sigma = 0.5 \\ \mu = 0.6, \sigma = 0.4 \\ \mu = 0.5, \sigma = 0.4 \\ \mu = 3.7, \sigma = 2.5 \end{array} \implies \begin{bmatrix} \frac{1-1.3}{0.5+0.1} & \frac{1-1.3}{0.5+0.1} & \frac{2-1.3}{0.5+0.1} \\ \frac{0.9-0.6}{0.4+0.1} & \frac{0.9-0.6}{0.4+0.1} & \frac{0-0.6}{0.4+0.1} \\ \frac{0.7-0.5}{0.4+0.1} & \frac{0.8-0.5}{0.4+0.1} & \frac{0-0.5}{0.4+0.1} \\ \frac{3-3.7}{2.5+0.1} & \frac{1-3.7}{2.5+0.1} & \frac{7-3.7}{2.5+0.1} \end{bmatrix}$$

As discussed in Section 6.1.1, the layer normalization unit in each sub-layer is used to standardize the output of a residual block. Here we describe a more general formulation for this structure. Suppose that  $F(\cdot)$  is a neural network we want to run. Then, the post-norm structure of  $F(\cdot)$  is given by

$$\mathbf{H}_{\text{out}} = \text{LNorm}(F(\mathbf{H}_{\text{in}}) + \mathbf{H}_{\text{in}}) \quad (6.35)$$

where  $\mathbf{H}_{\text{in}}$  and  $\mathbf{H}_{\text{output}}$  are the input and output of this model. Clearly, Eq. (6.4) is an instance of this equation.

An alternative approach to introducing layer normalization and residual connections into modeling is to execute the  $\text{LNorm}(\cdot)$  function right after the  $F(\cdot)$  function, and to establish an identity mapping from the input to the output of the entire sub-layer. This structure, known as the **pre-norm** structure, can be expressed in the form

$$\mathbf{H}_{\text{out}} = \text{LNorm}(F(\mathbf{H}_{\text{in}})) + \mathbf{H}_{\text{in}} \quad (6.36)$$

Both post-norm and pre-norm Transformer models are widely used in NLP systems. See Figure 6.3 for a comparison of these two structures. In general, residual connections are considered an effective means to make the training of multi-layer neural networks easier. In this sense, pre-norm Transformer seems promising because it follows the convention that a residual connection is created to bypass the whole network and that the identity mapping from the input to the output leads to easier optimization of deep models. However, by considering the expressive power of a model, there may be modeling advantages in using post-norm Transformer because it does not so much rely on residual connections and enforces more sophisticated modeling for representation learning. In Section 6.3.2, we will see a discussion on this issue.

### 6.1.5 Feed-forward Neural Networks

The use of FFNs in Transformer is inspired in part by the fact that complex outputs can be formed by transforming the inputs through nonlinearities. While the self-attention model itself has some nonlinearity (in  $\text{Softmax}(\cdot)$ ), a more common way to do this is to consider additional layers with non-linear activation functions and linear transformations. Given an input  $\mathbf{H}_{\text{in}} \in \mathbb{R}^{m \times d}$  and an output  $\mathbf{H}_{\text{out}} \in \mathbb{R}^{m \times d}$ , the  $\mathbf{H}_{\text{out}} = \text{FFN}(\mathbf{H}_{\text{in}})$  function in Transformer

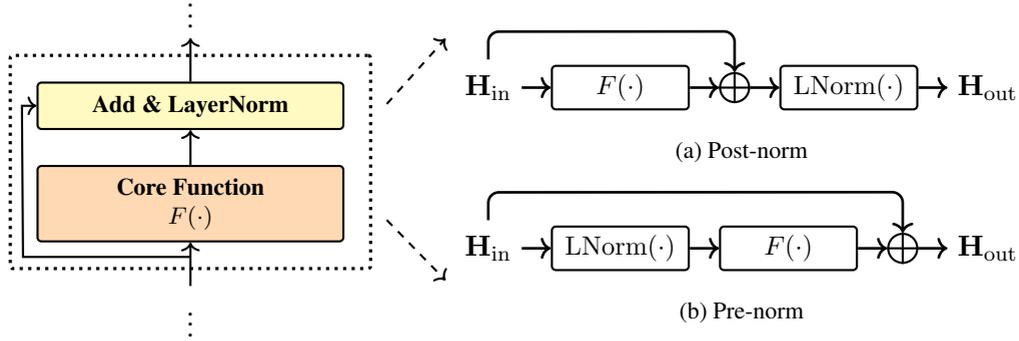


Figure 6.3: The post-norm and pre-norm structures.  $F(\cdot)$  = core function,  $\text{LNorm}(\cdot)$  = layer normalization, and  $\oplus$  = residual connection.

has the following form

$$\mathbf{H}_{\text{out}} = \mathbf{H}_{\text{hidden}} \mathbf{W}_f + \mathbf{b}_f \quad (6.37)$$

$$\mathbf{H}_{\text{hidden}} = \text{ReLU}(\mathbf{H}_{\text{in}} \mathbf{W}_h + \mathbf{b}_h) \quad (6.38)$$

where  $\mathbf{H}_{\text{hidden}} \in \mathbb{R}^{m \times d_{\text{ffn}}}$  is the hidden states, and  $\mathbf{W}_h \in \mathbb{R}^{d \times d_{\text{ffn}}}$ ,  $\mathbf{b}_h \in \mathbb{R}^{d_{\text{ffn}}}$ ,  $\mathbf{W}_f \in \mathbb{R}^{d_{\text{ffn}} \times d}$  and  $\mathbf{b}_f \in \mathbb{R}^d$  are the parameters. This is a two-layer FFN in which the first layer (or hidden layer) introduces a nonlinearity through  $\text{ReLU}(\cdot)$ <sup>2</sup> and the second layer involves only a linear transformation. It is common practice in Transformer to use a larger size of the hidden layer. For example, a common choice is  $d_{\text{ffn}} = 4d$ , that is, the size of each hidden representation is 4 times as large as the input.

Note that using a wide FFN sub-layer has been proven to be of great practical value in many state-of-the-art systems. However, a consequence of this is that the model is occupied by the parameters of the FFN. Table 6.1 shows parameter numbers and time complexities for different modules of a standard Transformer system. We see that FFNs dominate the model size when  $d_{\text{ffn}}$  is large, though they are not the most time consuming components. In the case of very big Transform models, we therefore wish to address this problem for building efficient systems.

### 6.1.6 Attention Models on the Decoder Side

A decoder layer involves two attention sub-layers, the first of which is a self-attention sub-layer, and the second is a cross-attention sub-layer. These sub-layers are based on either the post-norm or the pre-norm structure, but differ by designs of the attention functions. Consider, for example, the post-norm structure, described in Eq. (6.35). We can define the cross-attention

<sup>2</sup> $\text{ReLU}(x) = \max\{0, x\}$ .

	Sub-model	# of Parameters	Time Complexity	×
Encoder	Multi-head Self-attention	$4d^2$	$O(m^2 \cdot d)$	$L$
	Feed-forward Network	$2d \cdot d_{\text{ffn}} + d + d_{\text{ffn}}$	$O(m \cdot d \cdot d_{\text{ffn}})$	$L$
	Layer Normalization	$2d$	$O(d)$	$2L$
Decoder	Multi-head Self-attention	$4d^2$	$O(n^2 \cdot d)$	$L$
	Multi-head Cross-attention	$4d^2$	$O(m \cdot n \cdot d)$	$L$
	Feed-forward Network	$2d \cdot d_{\text{ffn}} + d + d_{\text{ffn}}$	$O(n \cdot d \cdot d_{\text{ffn}})$	$L$
	Layer Normalization	$2d$	$O(d)$	$3L$

Table 6.1: Numbers of parameters and time complexities of different Transformer modules under different setups.  $m$  = source-sequence length,  $n$  = target-sequence length,  $d$  = default number of dimensions of a hidden layer,  $d_{\text{ffn}}$  = number of dimensions of the FFN hidden layer,  $\tau$  = number of heads in the attention models, and  $L$  = number of encoding or decoding layers. The column  $\times$  means the number of times a sub-model is applied on the encoder or decoder side. The time complexities are estimated by counting the number of multiplication of floating-point numbers.

and self-attention sub-layers for a decoding layer to be

$$\begin{aligned} \mathbf{S}_{\text{cross}} &= \text{Layer}_{\text{cross}}(\mathbf{H}_{\text{enc}}, \mathbf{S}_{\text{self}}) \\ &= \text{LNorm}(\text{Att}_{\text{cross}}(\mathbf{H}_{\text{enc}}, \mathbf{S}_{\text{self}}) + \mathbf{S}_{\text{self}}) \end{aligned} \quad (6.39)$$

$$\begin{aligned} \mathbf{S}_{\text{self}} &= \text{Layer}_{\text{self}}(\mathbf{S}) \\ &= \text{LNorm}(\text{Att}_{\text{self}}(\mathbf{S}) + \mathbf{S}) \end{aligned} \quad (6.40)$$

where  $\mathbf{S} \in \mathbb{R}^{n \times d}$  is the input of the self-attention sub-layer,  $\mathbf{S}_{\text{cross}} \in \mathbb{R}^{n \times d}$  and  $\mathbf{S}_{\text{self}} \in \mathbb{R}^{n \times d}$  are the outputs of the sub-layers, and  $\mathbf{H}_{\text{enc}} \in \mathbb{R}^{m \times d}$  is the output of the encoder<sup>3</sup>.

As with conventional attention models, cross-attention is primarily used to model the correspondence between the source-side and target-side sequences. The  $\text{Att}_{\text{cross}}(\cdot)$  function is based on the QKV attention model which generates the result of querying a collection of key-value pairs. More specifically, we define the queries, keys and values as linear mappings of  $\mathbf{S}_{\text{self}}$  and  $\mathbf{H}_{\text{enc}}$ , as follows

$$\mathbf{S}_{\text{self}}^q = \mathbf{S}_{\text{self}} \mathbf{W}_{\text{cross}}^q \quad (6.41)$$

$$\mathbf{H}_{\text{enc}}^k = \mathbf{H}_{\text{enc}} \mathbf{W}_{\text{enc}}^k \quad (6.42)$$

$$\mathbf{H}_{\text{enc}}^v = \mathbf{H}_{\text{enc}} \mathbf{W}_{\text{enc}}^v \quad (6.43)$$

where  $\mathbf{W}_{\text{cross}}^q, \mathbf{W}_{\text{enc}}^k, \mathbf{W}_{\text{enc}}^v \in \mathbb{R}^{d \times d}$  are the parameters of the mappings. In other words, the queries are defined based on  $\mathbf{S}_{\text{self}}$ , and the keys and values are defined based on  $\mathbf{H}_{\text{enc}}$ .

<sup>3</sup>For an encoder having  $L$  encoder layers,  $\mathbf{H}_{\text{enc}} = \mathbf{H}^L$ .

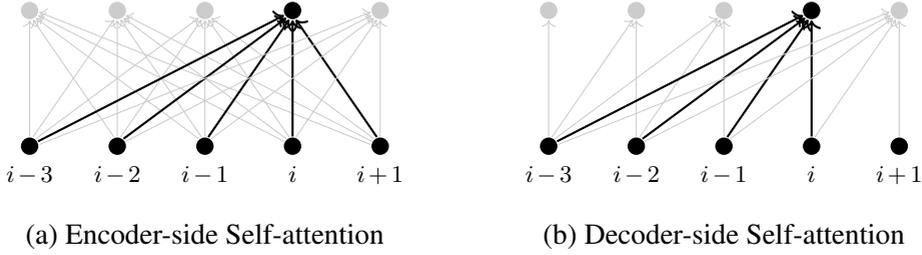


Figure 6.4: Self-attention on the encoder and decoder sides. Each line connects an input and an output of the self-attention model, indicating a dependency of an output state on an input state. For encoder self-attention, the output at any position is computed by having access to the entire sequence. By contrast, for decoder self-attention, the output at position  $i$  is computed by seeing only inputs at positions up to  $i$ .

$\text{Att}_{\text{cross}}(\cdot)$  is then defined as

$$\begin{aligned} \text{Att}_{\text{cross}}(\mathbf{H}_{\text{enc}}, \mathbf{S}_{\text{self}}) &= \text{Att}_{\text{qkv}}(\mathbf{S}_{\text{self}}^q, \mathbf{H}_{\text{enc}}^k, \mathbf{H}_{\text{enc}}^v) \\ &= \text{Softmax}\left(\frac{\mathbf{S}_{\text{self}}^q [\mathbf{H}_{\text{enc}}^k]^T}{\sqrt{d}}\right) \mathbf{H}_{\text{enc}}^v \end{aligned} \quad (6.44)$$

The  $\text{Att}_{\text{self}}(\cdot)$  function has a similar form as  $\text{Att}_{\text{cross}}(\cdot)$ , with linear mappings of  $\mathbf{S}$  taken as the queries, keys, and values, like this

$$\begin{aligned} \text{Att}_{\text{self}}(\mathbf{S}) &= \text{Att}_{\text{qkv}}(\mathbf{S}^q, \mathbf{S}^k, \mathbf{S}^v) \\ &= \text{Softmax}\left(\frac{\mathbf{S}^q [\mathbf{S}^k]^T}{\sqrt{d}} + \mathbf{M}\right) \mathbf{S}^v \end{aligned} \quad (6.45)$$

where  $\mathbf{S}^q = \mathbf{S} \mathbf{W}_{\text{dec}}^q$ ,  $\mathbf{S}^k = \mathbf{S} \mathbf{W}_{\text{dec}}^k$ , and  $\mathbf{S}^v = \mathbf{S} \mathbf{W}_{\text{dec}}^v$  are linear mappings of  $\mathbf{S}$  with parameters  $\mathbf{W}_{\text{dec}}^q, \mathbf{W}_{\text{dec}}^k, \mathbf{W}_{\text{dec}}^v \in \mathbb{R}^{d \times d}$ .

This form is similar to that of Eq. (6.20). A difference compared to self-attention on the encoder side, however, is that the model here needs to follow the rule of left-to-right generation (see Figure 6.4). That is, given a target-side word at the position  $i$ , we can see only the target-side words in the left context  $y_1 \dots y_{i-1}$ . To do this, we add a masking variable  $\mathbf{M}$  to the unnormalized weight matrix  $\frac{\mathbf{S}^q [\mathbf{S}^k]^T}{\sqrt{d}} + \mathbf{M}$ . Both  $\mathbf{M}$  and  $\frac{\mathbf{S}^q [\mathbf{S}^k]^T}{\sqrt{d}} + \mathbf{M}$  are of size  $n \times n$ , and so a lower value of an entry of  $\mathbf{M}$  means a larger bias towards lower alignment scores for the corresponding entry of  $\frac{\mathbf{S}^q [\mathbf{S}^k]^T}{\sqrt{d}} + \mathbf{M}$ . In order to avoid access to the right context given  $i$ ,  $\mathbf{M}$  is defined to be

$$M(i, k) = \begin{cases} 0 & i \leq k \\ -\infty & i > k \end{cases} \quad (6.46)$$

where  $M(i, k)$  indicates a bias term for the alignment score between positions  $i$  and  $k$ . Below

we show an example of how the masking variable is applied (assume  $n = 4$ ).

$$\begin{aligned}
 & \text{Softmax}\left(\frac{\mathbf{S}^q[\mathbf{S}^k]^T}{\sqrt{d}} + \mathbf{M}\right) \\
 = & \text{Softmax}\left(\begin{bmatrix} 2 & 0.1 & 1 & 1 \\ 0 & 0.9 & 0.9 & 0.9 \\ 0.2 & 0.8 & 0.7 & 2 \\ 0.3 & 1 & 0.3 & 3 \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}\right) \\
 = & \text{Softmax}\left(\begin{bmatrix} 2 & -\infty & -\infty & -\infty \\ 0 & 0.9 & -\infty & -\infty \\ 0.2 & 0.8 & 0.7 & -\infty \\ 0.3 & 1 & 0.3 & 3 \end{bmatrix}\right) \\
 = & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.3 & 0.7 & 0 & 0 \\ 0.2 & 0.4 & 0.4 & 0 \\ 0.05 & 0.1 & 0.05 & 0.8 \end{bmatrix} \tag{6.47}
 \end{aligned}$$

As noted in Section 6.1.3, it is easy to improve these models by using the multi-head attention mechanism. Also, since decoders are typically the most time-consuming part of practical systems, the bulk of the computational effort in running these systems is very much concerned with the efficiency of the attention modules on the decoder side.

### 6.1.7 Training and Inference

Transformers can be trained and used in a regular way. For example, we can train a Transformer model by performing gradient descent to minimize some loss function on the training data (see Chapter 2), and test the trained model by performing beam search on the unseen data (see Chapter 5). Below we present some of the techniques that are typically used in the training and inference of Transformer models.

- **Learning Rate Scheduling.** As standard neural networks, Transformers can be directly trained using back-propagation. The training process is generally iterated many times to make the models fit the training data well. In each training step, we update the weights of the neural networks by moving them a small step in the direction of negative gradients of errors. There are many ways to design the update rule of training. A popular choice is to use the Adam optimization method [Kingma and Ba, 2014]. To adjust the learning rate during training, Vaswani et al. [2017] present a learning rate scheduling strategy which increases the learning rate linearly for a number of steps and then decay it gradually. They design a learning rate of the form

$$lr = lr_0 \cdot \min\{n_{\text{step}}^{-0.5}, n_{\text{step}} \cdot (n_{\text{warmup}})^{-1.5}\} \tag{6.48}$$

where  $lr_0$  denotes the initial learning rate, and  $n_{\text{step}}$  denotes the number of training steps we have executed, and  $n_{\text{warmup}}$  denotes the number of warmup steps. In the first

$n_{\text{warmup}}$  steps, the learning rate  $lr$  grows larger as training proceeds. It reaches the highest value at the point of  $n_{\text{step}} = n_{\text{warmup}}$ , and then decreases as an inverse square root function (i.e.,  $lr_0 \cdot n_{\text{step}}^{-0.5}$ ).

- **Batching and Padding.** To make a trade-off between global optimization and training convergency, it is common to update the weights each time on a relatively small collection of samples, called a **minibatch** of samples. Therefore, we can consider a batch version of forward and backward computation processes in which the whole minibatch is used together to obtain the gradient information. One advantage of batching is that it allows the system to make use of efficient tensor operations to deal with multiple sequences in a single run. This requires that all the input sequences in a minibatch are stored in a single memory block, so that they can be read in and processed together. To illustrate this idea, consider a minimatch containing four samples whose source-sides are

A	B	C	D	E	F
M	N				
R	S	T			
W	X	Y	Z		

We can store these sequences in a  $4 \times 6$  continuous block where each “row” represents a sequence, like this

A	B	C	D	E	F
M	N	□	□	□	□
R	S	T	□	□	□
W	X	Y	Z	□	□

Here padding words □ are inserted between sequences, so that these sequences are aligned in the memory. Typically, we do not want padding to affect the operation of the system, and so we can simply define □ as a zero vector (call it **zero padding**). On the other hand, in some cases we are interested in using padding to describe something that is not covered by the input sequences. For example, we can replace padding words with the words in the left (or right) context of a sequence, though this may require modifications to the system to ensure that the newly added context words do not cause additional content to appear in the output.

- **Search and Caching.** At test time, we need to search the space of candidate hypotheses (or candidate target-side sequences) to identify the hypothesis (or target-side sequence) with the highest score.

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}} \operatorname{score}(\mathbf{x}, \mathbf{y}) \quad (6.49)$$

where  $\operatorname{score}(\mathbf{x}, \mathbf{y})$  is the model score of the target-side sequence  $\mathbf{y}$  given the source-side sequence  $\mathbf{x}$ . While there are many search algorithms to achieve this, most of them share a similar structure: the search program operates by extending candidate target-side

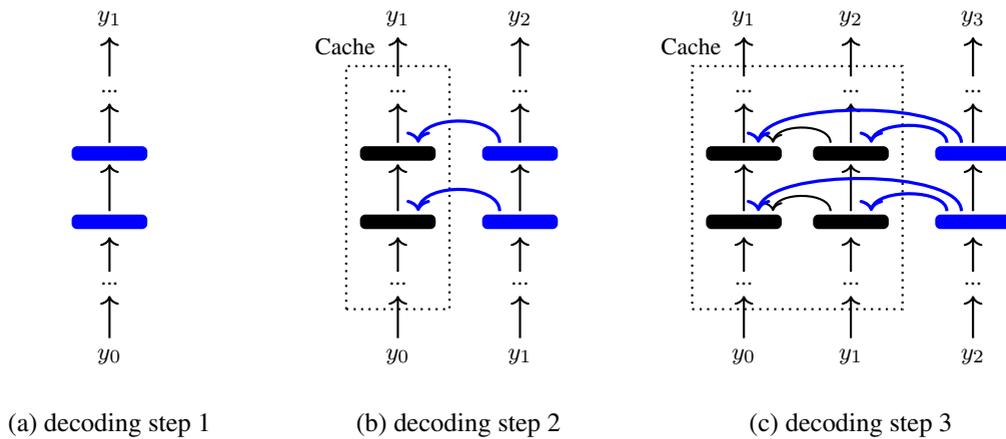


Figure 6.5: Illustration of the caching mechanism in Transformer decoders. Rectangles indicate the states of decoding layers or sub-layers. At step  $i$ , all the states at previous steps are stored in a cache (see dotted boxes), and we only need to compute the states for this step (see blue rectangles and arrows). Then, we add the newly generated states to the cache, and move on to step  $i + 1$ .

sequences in a pool at a time. In this way, the resulting algorithm can be viewed as a left-to-right generation procedure. For a more detailed discussion of search algorithms and model scores of general sequence-to-sequence models, see Chapter 5. Note that all of the designs of  $\text{score}(\mathbf{x}, \mathbf{y})$ , no matter how complex, are based on computing  $\Pr(\mathbf{y}|\mathbf{x})$ . Because the attention models used in Transformer require computing the dot-product of each pair of the input vectors of a layer, the time complexity of the search algorithm is a quadratic function of the length of  $\mathbf{y}$ . It is therefore not efficient to repeatedly compute the outputs of the attention models for positions that have been dealt with. This problem can be addressed by caching the states of each layer for words we have seen. Figure 6.5 illustrates the use of the caching mechanism in a search step. All the states for positions  $< i$  are maintained and easily accessed in a cache. At position  $i$ , all we need is to compute the states for the newly added word, and then to update the cache.

## 6.2 Syntax-aware Models

Although Transformer is simply a deep learning model that does not make use of any linguistic structure or assumption, it may be necessary to incorporate our prior knowledge into such systems. This is in part because NLP researchers have long believed that a higher level of abstraction of data is needed to develop ideal NLP systems, and there have been many systems that use structure as priors. However, structure is a wide-ranging topic and there are several types of structure one may refer to See [2018]’s work. For example, the inductive biases used in our model design can be thought of as some structural prior, while NLP models can also learn the underlying structure of problems by themselves. In this subsection we will discuss

some of these issues. We will focus on the methods of introducing linguistic structure into Transformer models. As Transformer can be applied to many NLP tasks, which differ much in their input and output formats, we will primarily discuss modifications to Transformer encoders (call them **syntax-aware Transformer encoders**). Our discussion, however, is general, and the methods can be easily extended to Transformer decoders.

## 6.2.1 Syntax-aware Input and Output

One of the simplest methods of incorporating structure into NLP systems is to modify the input sequence, leaving the system unchanged. As a simple example, consider a sentence where each word  $x_j$  is assigned a set of  $\kappa$  syntactic labels  $\{\text{tag}_j^1, \dots, \text{tag}_j^\kappa\}$  (e.g., POS labels and dependency labels). We can write these symbols together to define a new “word”

$$x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa$$

Then, the embedding of this word is given by

$$\mathbf{x}p_j = e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa) + \text{PE}(j) \quad (6.50)$$

where  $e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa) \in \mathbb{R}^d$  is the embedding of  $x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa$ . Since  $x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa$  is a complex symbol, we decompose the learning problem of  $e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa)$  into easier problems. For example, we can develop  $\kappa$  embedding models, each producing an embedding given a tag. Then, we write  $e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa)$  as a sum of the word embedding and tag embeddings

$$e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa) = \mathbf{x}_j + e(\text{tag}_j^1) + \dots + e(\text{tag}_j^\kappa) \quad (6.51)$$

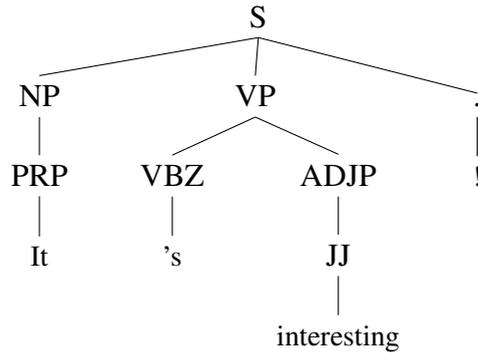
where  $\{e(\text{tag}_j^1), \dots, e(\text{tag}_j^\kappa)\}$  are the embeddings of the tags. Alternatively, we can combine these embeddings via a neural network in the form

$$e(x_j / \text{tag}_j^1 / \dots / \text{tag}_j^\kappa) = \text{FFN}_{\text{embed}}(\mathbf{x}_j, e(\text{tag}_j^1), \dots, e(\text{tag}_j^\kappa)) \quad (6.52)$$

where  $\text{FFN}_{\text{embed}}(\cdot)$  is a feed-forward neural network that has one layer or two.

We can do the same thing for sentences on the decoder side as well, and treat  $y_i / \text{tag}_i^1 / \dots / \text{tag}_i^\kappa$  as a syntax-augmented word. However, this may lead to a much larger target-side vocabulary and poses a computational challenge for training and inference.

Another form that is commonly used to represent a sentence is syntax tree. In linguistics, the syntax of a sentence can be interpreted in many different ways, resulting in various grammars and the corresponding tree (or graph)-based representations. While these representations differ in their syntactic forms, a general approach to use them in sequence modeling is **tree linearization**. Consider the following sentence annotated with a constituency-based parse tree



We can write this tree structure as a sequence of words, syntactic labels and brackets via a tree traversal algorithm, as follows

(S (NP (PRP **It** )PRP )NP (VP (VBZ **'s** )VBZ (ADJP (JJ **interesting** )JJ )ADJP )VP ( . **!** ) . )S

This sequence of syntactic tokens can be used as an input to the system, that is, each token is represented by word and positional embeddings, and then the sum of these embeddings is treated as a regular input of the encoder. An example of the use of linearized trees is tree-to-string machine translation in which a syntax tree in one language is translated into a string in another language [Li et al., 2017b; Currey and Heafield, 2018]. Linearized trees can also be used for tree generation. For example, we can frame parsing tasks as sequence-to-sequence problems to map an input text to a sequential representation of its corresponding syntax tree [Vinyals et al., 2015; Choe and Charniak, 2016]. See Figure 6.6 for illustrations of these models. It should be noted that the methods described here are not specific to Transformer but could be applied to many models, such as RNN-based models.

## 6.2.2 Syntax-aware Attention Models

For Transformer models, it also makes sense to make use of syntax trees to guide the process of learning sequence representations. In the previous section we saw how representations of a sequence can be computed by relating different positions within that sequence. This allows us to impose some structure on these relations which are represented by distributions of attention weights over all the positions. To do this we use the encoder self-attention with an additive mask

$$\text{AttSyn}_{\text{self}}(\mathbf{H}) = \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}} + \mathbf{M}\right)\mathbf{H}^v \quad (6.53)$$

or alternatively with a multiplicative mask

$$\text{AttSyn}_{\text{self}}(\mathbf{H}) = \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}} \odot \mathbf{M}\right)\mathbf{H}^v \quad (6.54)$$

where  $\mathbf{M} \in \mathbb{R}^{m \times m}$  is a matrix of masking variables in which a larger value of  $M(i, j)$  indicates

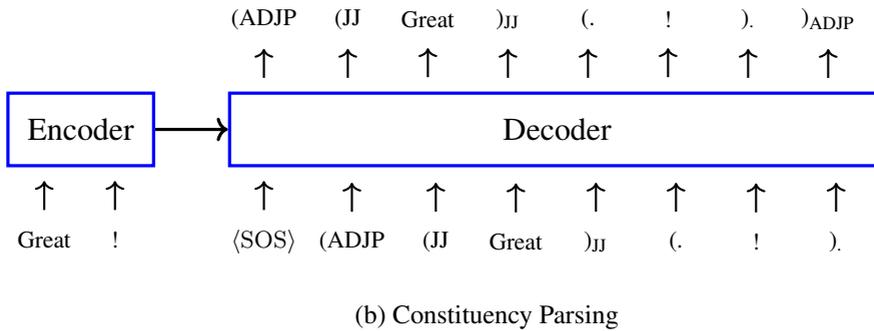
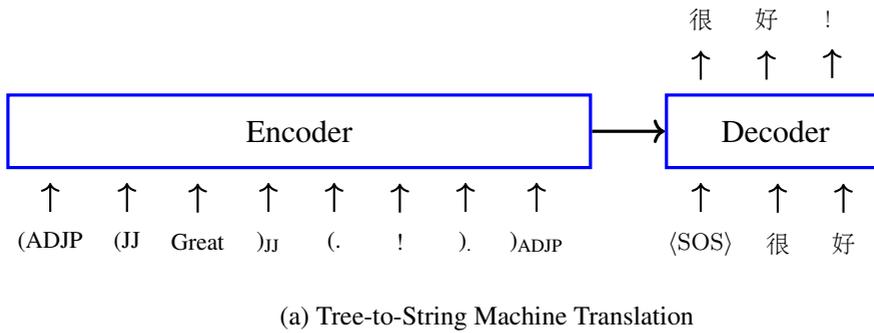


Figure 6.6: Illustration of tree linearization on either the encoder or decoder side. For tree-to-string machine translation, the encoder takes sequential representation of an input parse tree, and the decoder outputs the corresponding translation. For parsing, the encoder takes a sentence, and the decoder outputs the corresponding syntax tree.

a stronger syntactic correlation between positions  $i$  and  $j$ . In the following description we choose Eq. (6.54) as the basic form.

One common way to design  $\mathbf{M}$  is to project syntactic relations of the input tree structure into constraints over the sequence. Here we consider constituency parse trees and dependency parse trees for illustration. Generally, two types of masking methods are employed.

- **0-1 Masking.** This method assigns  $M(i, j)$  a value of 1 if the words at positions  $i$  and  $j$  are considered syntactically correlated and a value of 0 otherwise [Zhang et al., 2020c; Bai et al., 2021]. To model the relation between two words in a syntax tree, we can consider the distance between their corresponding nodes. One of the simplest forms is given by

$$M(i, j) = \begin{cases} 1 & \omega(i, j) \leq \omega_{\max} \\ 0 & \text{otherwise} \end{cases} \quad (6.55)$$

where  $\omega(i, j)$  is the length of the shortest path between the nodes of the words at positions  $i$  and  $j$ . For example, given a dependency parse tree,  $\omega(i, j)$  is the number

of dependency edges in the path between the two words. For a constituency parse tree, all the words are leaf nodes, and so  $\omega(i, j)$  gives a tree distance between the two leaves in the same branch of the tree.  $\omega_{\max}$  is a parameter used to control the maximum distance between two nodes that can be considered syntactically correlated. For example, assuming that there is a dependency parse tree and  $\omega_{\max} = 1$ , Eq. (6.55) enforces a constraint that the attention score between positions  $i$  and  $j$  is computed only if they have a parent-dependent relation<sup>4</sup>.

- **Soft Masking.** Instead of treating  $\mathbf{M}$  as a hard constraint, we can use it as a soft constraint that scales the attention weight between positions  $i$  and  $j$  in terms of the degree to which the corresponding words are correlated. An idea is to reduce the attention weight as  $\omega(i, j)$  becomes larger. A very simple method to do this is to transform  $\omega(i, j)$  in some way that  $M(i, j)$  holds a negative correlation relationship with  $\omega(i, j)$  and its value falls into the interval  $[0, 1]$

$$M(i, j) = \text{DNorm}(\omega(i, j)) \quad (6.56)$$

There are several alternative designs for  $\text{DNorm}(\cdot)$ . For example, one can compute a standardized score of  $-\omega(i, j)$  by subtracting its mean and dividing by its standard deviation [Chen et al., 2018a], or can normalize  $1/\omega(i, j)$  over all possible  $j$  in the sequence [Xu et al., 2021b]. In cases where parsers can output a score between positions  $i$  and  $j$ , it is also possible to use this score to compute  $M(i, j)$ . For example, a dependency parser can produce the probability of the word at position  $i$  being the parent of the word at position  $j$  [Strubell et al., 2018]. We can then write  $M(i, j)$  as

$$M(i, j) = \text{Pr}_{\text{parent}}(i|j) \quad (6.57)$$

or alternatively

$$M(i, j) = \max\{\text{Pr}_{\text{parent}}(i|j), \text{Pr}_{\text{parent}}(j|i)\} \quad (6.58)$$

where  $\text{Pr}_{\text{parent}}(i|j)$  and  $\text{Pr}_{\text{parent}}(j|i)$  are the probabilities given by the parser. See Figure 6.7 for an example of inducing a soft masking variable from a dependency parse tree.

### 6.2.3 Multi-branch Models

Introducing syntax into NLP systems is not easy. This is partially because automatic parse trees may have errors, and partially because the use of syntax may lead to strong assumption of the underlying structure of a sentence. Rather than combining syntactic and word information

<sup>4</sup>For multiplicative masks,  $M(i, j) = 0$  does not mean that the attention weight between  $j$  and  $i$  is zero because the Softmax function does not give a zero output for a dimension whose corresponding input is of a zero value. A method to “mask” an entry of  $\text{Softmax}(\frac{\mathbf{H}\mathbf{H}^T}{\sqrt{d}})$  is to use an additive mask and set  $M(i, j) = -\infty$  if  $\omega(i, j) > \omega_{\max}$ .

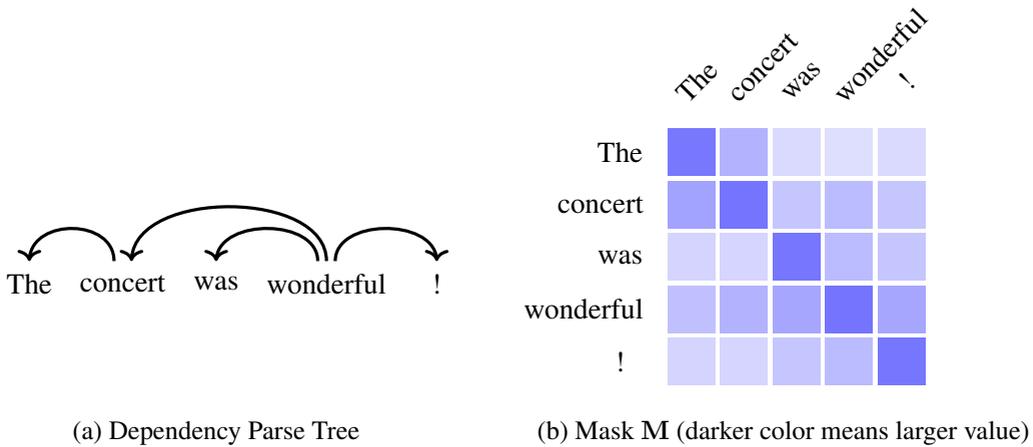


Figure 6.7: Priors induced from a dependency parse tree. The row  $i$  of the matrix  $M$  represents a distribution that describes how much weight we can give to  $M(i, j)$  in terms of the syntactic distance between  $i$  and  $j$ .

into one “big” model, it may be more flexible and effective to build one model to encode syntax and a different one to encode word sequences. One way to achieve this is through the use of multiple neural networks (called **branches** or **paths**), each dealing with one type of input. The outputs of these branches are then combined to produce an output [Xie et al., 2017; Fan et al., 2020; Lin et al., 2022b]. Various methods have therefore been used to combine different types of input for neural models like Transformer.

One commonly-used approach is to build two separate encoders, in which one model is trained to encode the syntactic input (denoted by  $\mathbf{t}$ ), and the other is trained to encode the usual input (denoted by  $\mathbf{x}$ ). Figure 6.8 (a) illustrates this multi-encoder architecture. The syntactic encoder  $\text{Encode}_{\text{syn}}(\mathbf{t})$  is based on models presented in Sections 6.2.1 and 6.2.2, and the text encoder  $\text{Encode}_{\text{text}}(\mathbf{x})$  is a standard Transformer encoder. The representations generated by these encoders are then fed into the combination model as input, and combined into a hybrid representation, given by

$$\begin{aligned} \mathbf{H}_{\text{hybrid}} &= \text{Combine}(\mathbf{H}_{\text{syn}}, \mathbf{H}_{\text{text}}) \\ &= \text{Combine}(\text{Encode}_{\text{syn}}(\mathbf{t}), \text{Encode}_{\text{text}}(\mathbf{x})) \end{aligned} \quad (6.59)$$

There are several designs for  $\text{Combine}(\cdot)$ , depending on what kind of problems we apply the encoders to. For example, if we want to develop a text classifier,  $\text{Combine}(\cdot)$  can be a simple pooling network. For more complicated tasks, such as machine translation,  $\text{Combine}(\cdot)$  can be a Transformer encoder as well, and we can fuse information from different sources by performing self-attention on  $[\mathbf{H}_{\text{syn}}, \mathbf{H}_{\text{text}}]$ .

While we restrict attention to syntactic models in this section, the general multi-encoder architecture can be used in many problems where inputs from additional sources are required. For example, one can use one encoder to represent a sentence, and use another encoder to

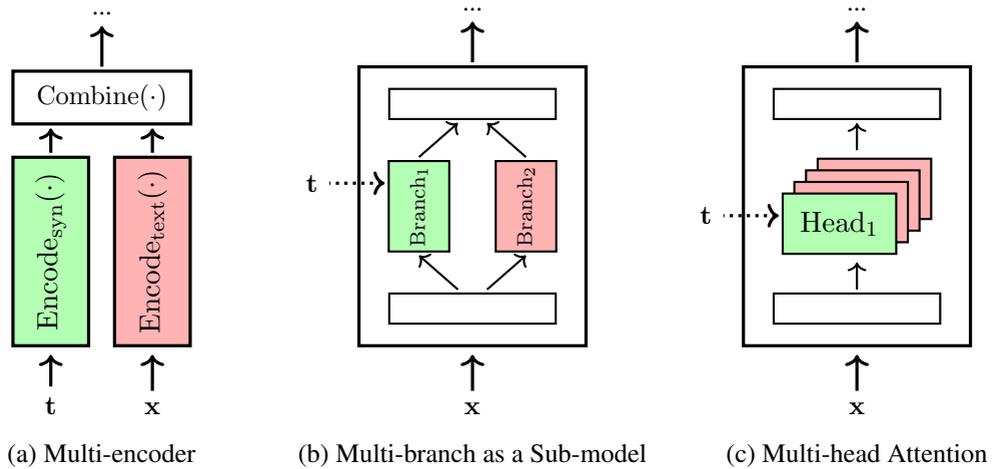


Figure 6.8: Multi-branch architectures. There are two inputs: a sentence (denoted by  $\mathbf{x}$ ) and the syntax tree of the sentence (denoted by  $\mathbf{t}$ ). In the multi-encoder architecture (see sub-figure (a)), two encoders are constructed to encode  $\mathbf{x}$  and  $\mathbf{t}$ , respectively. A combination model then takes the outputs of the encoders and produces a combined representation of  $\mathbf{x}$  and  $\mathbf{t}$ . The idea of multi-branch networks can be used for designing sub-models of the encoder. A simple example is that we create multiple paths in parallel for some layers of the encoder (see sub-figure (b)). Another example is multi-head attention (see sub-figure (c)) where we use different heads to learn different representations.

represent the previous sentence in the same document. We thus have a context-aware model by combining the two encoders [Voita et al., 2018; Li et al., 2020a]. Furthermore, the architectures of the encoders do not need to be restricted to Transformer, and we can choose different models for different branches. For example, as a widely-used 2-branch encoding architecture, we can use a CNN-based encoder to model local context, and a Transformer encoder to model global context [Wu et al., 2020b].

Sub-models of a Transformer model can also be multi-branch neural networks. See Figure 6.8 (b) for an example involving two self-attention branches. One is the standard self-attention network  $\text{Att}_{\text{self}}(\mathbf{H})$ . The other is the syntax-aware self-attention network  $\text{AttSyn}_{\text{self}}(\mathbf{H})$ . The output of the self-attention model is a linear combination of the outputs of these two branches [Xu et al., 2021b], given by

$$\mathbf{H}_{\text{self}} = \alpha \cdot \text{Att}_{\text{self}}(\mathbf{H}) + (1 - \alpha) \cdot \text{AttSyn}_{\text{self}}(\mathbf{H}) \quad (6.60)$$

where  $\alpha$  is a coefficient of combination.  $\mathbf{H}_{\text{self}}$  can be used as usual by taking a layer normalization function and adding a residual connection, and so the overall architecture is the same as standard Transformer models.

Multi-head attention networks can also be viewed as forms of multi-branch models. Therefore, we can provide guidance from syntax to only some of the heads while keeping the rest unchanged [Strubell et al., 2018]. This approach is illustrated in Figure 6.8 (c) where only one

head of the self-attention sub-layer makes use of syntax trees for computing attention weights.

### 6.2.4 Multi-scale Models

In linguistics, syntax studies how sentences are built up by smaller constituents. Different levels of these constituents are in general organized in a hierarchical structure, called **syntactic hierarchy**. It is therefore possible to use multiple levels of syntactic constituents to explain the same sentence, for example, words explain how the sentence is constructed from small meaningful units, and phrases explain how the sentence is constructed from larger linguistic units.

Multi-scale Transformers leverage varying abstraction levels of data to represent a sentence using diverse feature scales. A common approach is to write a sentence in multiple different forms and then to combine them using a multi-branch network [Hao et al., 2019]. For example, consider a sentence

The oldest beer-making facility was discovered in China.

We can tokenize it into a sequence of words, denoted by

$$\mathbf{x}_{\text{words}} = \text{The oldest beer-making facility was discovered in China .}$$

Alternatively, we can write it as a sequence of phrases by using a parser, denoted by

$$\mathbf{x}_{\text{phrases}} = [\text{The oldest beer-making facility}]_{\text{NP}} [\text{was discovered in China}]_{\text{VP}} [.]$$

The simplest way to build a multi-scale model is to encode  $\mathbf{x}_{\text{words}}$  and  $\mathbf{x}_{\text{phrases}}$  using two separate Transformer encoders. Then, the outputs of these encoders are combined in some way. This leads to the same form as Eq. (6.59), and we can view this model as an instance of the general multi-encoder architecture.

Both  $\mathbf{x}_{\text{words}}$  and  $\mathbf{x}_{\text{phrases}}$  can be viewed as sequences of tokens, for example,  $\mathbf{x}_{\text{words}}$  has nine word-based tokens, and  $\mathbf{x}_{\text{phrases}}$  has three phrase-based tokens<sup>5</sup>. However, involving all possible phrases will result in a huge vocabulary. We therefore need some method to represent each phrase as an embedding in a cheap way. By treating phrase embedding as a sequence modeling problem, it is straightforward to learn sub-sequence representations simply by considering the sequence models described in the previous chapters and this chapter. Now we have a two-stage learning process. In the first stage, we learn the embeddings of input units on different scales using separate models. In the second stage, we learn to encode sequences on different scales using a multi-branch model.

More generally, we do not need to restrict ourselves to linguistically meaningful units in multi-scale representation learning. For example, we can learn sub-word segmentations from data and represent an input sentence as a sequence of sub-words. This results in a hierarchical

<sup>5</sup> $\mathbf{x}_{\text{phrases}}$  comprises three tokens *The oldest beer-making facility, was discovered in China, and ..*

representation of the sentence, for example, sub-words  $\rightarrow$  words  $\rightarrow$  phrases. While the learned sub-words may not have linguistic meanings, they provide a new insight into modeling words and phrases, as well as a new scale of features. Also, we do not need to develop multiple encoders for multi-scale modeling. An alternative approach is to take representations on different scales in the multi-head self-attention attention modules, which makes it easier to model the interactions among different scales [Guo et al., 2020; Li et al., 2022b].

A problem with the approaches described above, however, is that the representations (or attention weight matrices) learned on different scales are of different sizes. For example, in the above examples, the representation learned from  $\mathbf{x}_{\text{words}}$  is a  $9 \times d$  matrix, and the representation learned from  $\mathbf{x}_{\text{phrases}}$  is a  $3 \times d$  matrix. A simple solution to this problem is to perform upsampling on the phrase-based representation to expand it to a  $9 \times d$  matrix. Likewise, we can perform downsampling on the word-based representation to shrink it to a  $3 \times d$  matrix. Then, the combination model  $\text{Combine}(\cdot)$  can be the same as those described in Section 6.2.3.

It is worth noting that multi-scale modeling is widely discussed in several fields. For example, in computer vision, multi-scale modeling is often referred to as a process of learning a series of feature maps on the input image [Fan et al., 2021; Li et al., 2022f]. Unlike the multi-branch models presented here, the multi-scale vision Transformer models make use of the hierarchical nature of features in representing images. Systems of this kind are often based on a stack of layers in which each layer learns the features on a larger scale (e.g., a higher channel capacity) from the features produced by the previous layer.

### 6.2.5 Transformers as Syntax Learners

So far we have discussed syntax trees as being constraints or priors on the encoding process so that we can make use of linguistic representations in learning neural networks. It is natural to wonder whether these neural models can learn some knowledge of linguistic structure from data without human design linguistic annotations. This reflects one of the goals of developing NLP systems: linguistic knowledge can be learned from data and encoded in models.

In order to explore the linguistic properties learned by NLP systems, a simple method is to examine the syntactic behaviors of the outputs of the systems. For example, we can examine whether the outputs of language generation systems have grammatical errors. Another example is to ask these systems to accomplish tasks that make sense for linguistics, though they are not trained to do so [Brown et al., 2020]. However, examining and explaining how model predictions exhibit syntactic abilities is not sufficient to answer the question. It is also the case that the neural networks have learned some knowledge about language, but it is not used in prediction [Clark et al., 2019a]. Therefore, we need to see what is modeled and learned inside these neural networks.

One approach to examining the latent linguistic structure in Transformer models is to develop **probes** to see whether and to what extent these models capture notions of linguistics, such as dependency relations and parts-of-speech. A general approach to probing is to extract the internal representations of the models and probe them for linguistic phenomena. For Transformer, it is usually achieved by examining the attention map and/or output of an

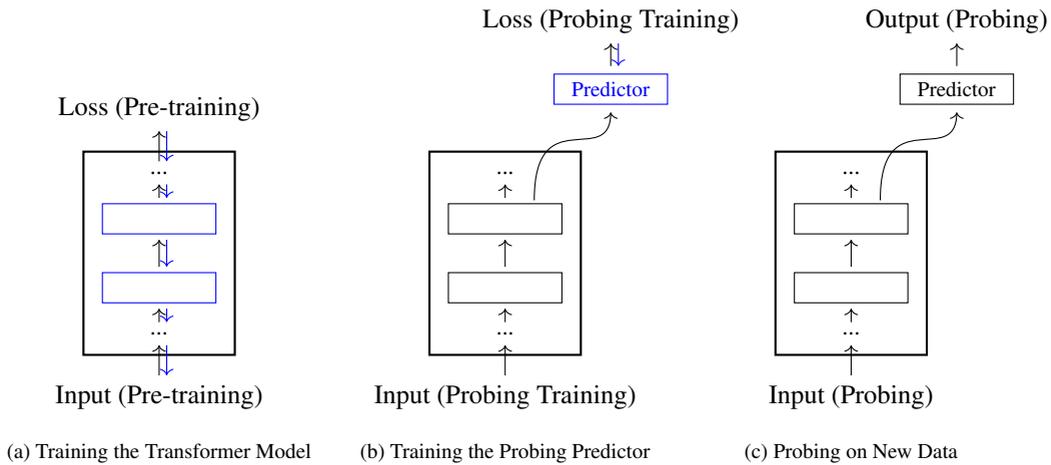


Figure 6.9: An overview of probing for Transformer-based models. Given a Transformer model (e.g., a Transformer-based language model), we first optimize the model parameters on some unlabeled data. Then, we develop a predictor which takes the states of a hidden layer of the Transformer model and generates outputs for a probing task (see sub-figure (a)). The predictor can be trained as usual in which only the parameters of the predictor are optimized and the parameters of the Transformer model are fixed (see sub-figure (b)). The Transformer model and the predictor are used together to make predictions on new data for probing (see sub-figure (c)).

attention layer. Then, we construct a **probing predictor** (or **probing classifier**) that takes these internal representations as input and produces linguistic notions as output [Belinkov, 2022]. The probing predictor can be based on either simple heuristics or parameterized models optimized on the probing task. Recent work shows that large-scale Transformer-based language models exhibit good behaviors, called **emergent abilities**, in various probing tasks. However, we will not discuss details of these language modeling systems in this chapter, but leave them in the following chapters. Nevertheless, we assume here that we have a Transformer encoder that has been well trained on unlabeled data and can be used for probing. Figure 6.9 illustrates the process of probing.

Many probing methods have been used in recent work on analyzing and understanding what is learned in neural encoders. Here we describe some of the popular ones.

- **Trees.** Given a trained Transformer encoder, it is easy to know how “likely” two words of a sentence have some linguistic relationship by computing the attention weight between them. We can use this quantity to define a metric measuring the syntactic distance between the two words at positions  $i$  and  $j$

$$d_s(i, j) = 1 - \alpha(i, j) \quad (6.61)$$

By using this metric it is straightforward to construct the **minimum-spanning tree** for the sentence, that is, we connect all the words to form a tree structure with the minimum

total distance. The tree structure can be seen as a latent tree representation of the sentence that is induced from the neural network. While this dependency-tree-like structure can be used as a source of learned syntactic information in downstream tasks, it says nothing about our knowledge of syntax. An approach to aligning the representations in the encoder with linguistic structure is to learn to produce syntax trees that are consistent with human annotations. To do this, we need to develop a probing predictor that can be trained on tree-annotated data. Suppose that there is a human annotated dependency tree of a given sentence. For each pair of words, we can obtain a distance  $\omega(i, j)$  by counting the number of edges between them. Then, we can learn a distance metric based on the internal representations of the encoder to approximate  $\omega(i, j)$ . A simple form of such a metric is defined to be the Euclidean distance [Manning et al., 2020]. Let  $\mathbf{A} \in \mathbb{R}^{d \times k_s}$  be a parameter matrix. The form of the Euclidean distance is given by

$$d_s(i, j) = \sqrt{\|(\mathbf{h}_i - \mathbf{h}_j)\mathbf{A}\|_2^2} \quad (6.62)$$

where  $\mathbf{h}_i$  and  $\mathbf{h}_j$  are the representations produced by an encoding layer at positions  $i$  and  $j$ <sup>6</sup>. Given a set of tree-annotated sentences  $S$ , we can optimize the model by

$$\hat{\mathbf{A}} = \arg \max_{\mathbf{A}} \sum_{s \in S} \frac{1}{|s|^2} \sum_{i \in s, j \in s} |\omega(i, j) - d_s^2(i, j)| \quad (6.63)$$

where  $|s|$  is length of the sentence  $s$ , and  $(i, j)$  indicates a pair of words in  $s$ . The optimized model is then used to parse test sentences via the minimum-spanning tree algorithm, and we can compare the parse trees against the human-annotated trees. To obtain directed trees, which are standard forms of dependency syntax, one can update the above model by considering the relative distance of a word to the root. More details can be found in Manning et al. [2020]’s work. Here the probing predictor functions similarly to a neural parser, trained to predict a syntax tree based on a representation of the input sentence. This idea can be extended to other forms of syntactic structure, such as phrase structure trees [Shi et al., 2016].

- **Syntactic and Semantic Labels.** Many syntactic and semantic parsing tasks can be framed as problems of predicting linguistic labels given a sentence or its segments. A simple example is part-of-speech tagging in which each word of a sentence is labeled with a word class. A probe for part-of-speech tagging can be a classifier that takes a representation  $\mathbf{h}_j$  each time and outputs the corresponding word class. One general probing approach to these problems is **edge probing** [Tenney et al., 2019b;a]. Given a sentence, a labeled edge is defined as a tuple

$$(\text{span}_1, \text{span}_2, \text{label})$$

where  $\text{span}_1$  is a span  $[i_1, j_1]$ , and  $\text{span}_2$  is another span  $[i_2, j_2]$  (optionally), and label

<sup>6</sup>In general,  $\mathbf{h}_i$  and  $\mathbf{h}_j$  are the outputs of the last layer of the encoder. Alternatively, they can be weighted sums of the outputs of all the layers.

is the corresponding label. Our goal is to learn a probe to predict label given  $\text{span}_1$  and  $\text{span}_2$ . For example, for part-of-speech tagging,  $\text{span}_1$  is a unit span  $[j, j]$  for each position  $j$ ,  $\text{span}_2$  is an empty span, and label is the part-of-speech tag corresponding to the  $j$ -th word of the sentence; for dependency parsing and coreference resolution,  $\text{span}_1$  and  $\text{span}_2$  are two words or entities, and label is the relationship between them; for constituency parsing,  $\text{span}_1$  is a span of words,  $\text{span}_2$  is an empty span, and label is the syntactic category of the tree node yielding  $\text{span}_1$ . In simple cases, the probing model can be a multi-layer feed-forward neural network with a Softmax output layer. As usual, this model is trained on labeled data, and then tested on new data.

- **Surface Forms of Words and Sentences.** Probing tasks can also be designed to examine whether the representations embed the surface information of sentences or words [Adi et al., 2016; Conneau et al., 2018]. A simple sentence-level probing task is **sentence length prediction**. To do this, we first represent the sentence as a single vector  $\mathbf{h}$ <sup>7</sup>, and then build a classifier to categorize  $\mathbf{h}$  into the corresponding length bin. Similarly, probes can be built to predict whether two words at positions  $i$  and  $j$  are reordered in the sentence given  $\mathbf{h}_i$  and  $\mathbf{h}_j$ . Also, we can develop probes to address conventional problems in morphology. For example, we reconstruct the word at position  $j$  or predict its sense with the representation  $\mathbf{h}_j$ . In addition, probing tasks can be focused on particular linguistic problems, for example, numeracy [Wallace et al., 2019] and function words [Kim et al., 2019].
- **Cloze.** Of course, we can probe neural models for problems beyond syntax and morphology. One perspective on large-scale pre-trained Transformer models is to view them as knowledge bases containing facts about the world. It is therefore tempting to see if we can apply them to test factual knowledge. A simple method is to ask a probe to recover the missing item of a sentence [Petroni et al., 2019]. For example, if we have a cloze test

Shiji was written by \_\_\_\_.

we wish the probe to give an answer *Sima Qian* because there is a subject-object-relation fact (Shiji, Sima Qian, written-by). This probe can simply be a **masked language model** that is widely used in self-supervised learning of Transformer encoders.

In NLP, probing is closely related to pre-training of large language models (see Chapters 7 and 8). In general, we can see probing tasks as applications of these pre-trained language models, though probing is ordinarily used to give a quick test of the models. Ideally we would like to develop a probe that makes best use of the representations to deal with the problems. However, when a probe is complex and sufficiently well-trained, it might be difficult to say if the problem is solved by using the representations or the probe itself. A common way to emphasize the contribution of probes in problem-solving is to compare them with reasonable baselines or conduct the comparison on control tasks [Hewitt and Liang, 2019; Belinkov, 2022].

---

<sup>7</sup> $\mathbf{h}$  can be computed by performing a pooling operation on  $\{\mathbf{h}_1, \dots, \mathbf{h}_m\}$

## 6.3 Improved Architectures

In this section we present several improvements to the vanilla Transformer model. Unlike the previous section, most of the improvements are from the perspective of machine learning, rather than linguistics.

### 6.3.1 Locally Attentive Models

Methods of self-attention, as discussed in Section 6.1.3, can also be viewed as learning representations of the entire input sequence. The use of this global attention mechanism can lead to a better ability to deal with long-distance dependencies, but this model has a shortcoming: local information is not explicitly captured. Here we consider a few techniques that attempt to model the localness of representations.

#### 1. Priors of Local Modeling

One of the simplest ways of introducing local models into Transformers is to add a penalty term to the attention function in order to discourage large attention weights between distant positions. On the encoder-side, this leads to a form that we have already encountered several times in this chapter.

$$\text{AttLocal}_{\text{self}}(\mathbf{H}) = \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}} - \gamma \cdot \mathbf{G}\right)\mathbf{H}^v \quad (6.64)$$

where  $\gamma$  is the weight (or temperature) of the penalty term, and  $\mathbf{G} \in \mathbb{R}^{m \times m}$  is the matrix of penalties. Each entry  $G(i, j)$  indicates how much we penalize the model given positions  $i$  and  $j$ . A simple form of  $G(i, j)$  is a distance metric between  $i$  and  $j$ , for example

$$G(i, j) = |i - j| \quad (6.65)$$

Or  $G(i, j)$  can be defined as a Gaussian penalty function [Yang et al., 2018a]

$$G(i, j) = \frac{(i - j)^2}{2\sigma_i^2} \quad (6.66)$$

where  $\sigma_i$  is the standard deviation of the Gaussian distribution. For different  $j$ , both of the above penalty terms increase, linearly or exponentially, away from the maximum at  $i$  with distance  $|i - j|$ .

This method can be extended to the cross-attention model, like this

$$\text{AttLocal}_{\text{cross}}(\mathbf{H}, \mathbf{S}) = \text{Softmax}\left(\frac{\mathbf{S}^q[\mathbf{H}^k]^T}{\sqrt{d}} - \gamma \cdot \mathbf{G}\right)\mathbf{H}^v \quad (6.67)$$

where  $\mathbf{G}$  is an  $n \times m$  matrix. Each entry of  $\mathbf{G}$  can be defined as

$$G(i, j) = \frac{(\mu_i - j)^2}{2\sigma_i^2} \quad (6.68)$$

where  $\mu_i$  is the mean of the Gaussian distribution over the source-side positions. Both  $\mu_i$  and  $\sigma_i$  can be determined using heuristics. Alternatively, we can develop additional neural networks to model them and learn corresponding parameters together with other parameters of the Transformer model. For example, we can use a feed-forward neural network to predict  $\mu_i$  given  $\mathbf{s}_i$ .

One alternative to Eq. (6.64) (or Eq. (6.67)) treats the penalty term as a separate model and combines it with the original attention model. For example, we can define the self-attention model as

$$\text{AttLocal}_{\text{self}}(\mathbf{H}) = \left( (1-\beta) \cdot \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}}\right) + \beta \cdot \text{Softmax}(-\gamma \cdot \mathbf{G}) \right) \mathbf{H}^v \quad (6.69)$$

where  $\beta \in [0, 1]$  is the coefficient of the linear combination. Note that, to avoid empirical choices of the values of  $\alpha$  and  $\beta$ , we can use gating functions to predict  $\alpha$  and  $\beta$  and train these functions as usual.

Another alternative is to use a multiplicative mask to incorporate the prior into modeling, as in Eq. (6.54). This is given by

$$\text{AttLocal}_{\text{self}}(\mathbf{H}) = \text{Softmax}\left(\frac{\mathbf{H}^q[\mathbf{H}^k]^T}{\sqrt{d}} \odot \mathbf{G}'\right) \mathbf{H}^v \quad (6.70)$$

Here  $\mathbf{G}' \in [0, 1]^{m \times m}$  is a matrix of scalars. The scalar  $G'(i, j)$  gives a value of 1 when  $i = j$ , and a smaller value as  $j$  moves away from  $i$ .  $G'(i, j)$  can be obtained by normalizing  $-G(i, j)$  over all  $j$  or using alternative functions.

## 2. Local Attention

The term local attention has been used broadly to cover a wide range of problems and to refer to many different models in the NLP literature. The methods discussed above are those that impose soft constraints on attention models. In fact, local attention has its origins in attempts to restrict the scope of attention models for considerations of modeling and computational problems [Luong et al., 2015]. Research in this area often looks into introducing hard constraints, so that the resulting models can focus on parts of the input and ignore the rest. For example, we can predict a span of source-side positions for performing the attention function given a target-side position [Sperber et al., 2018; Yang et al., 2018a; Sukhbaatar et al., 2019]. Also, attention spans can be induced from syntax trees, for example, knowing sub-tree structures of a sentence may help winnow the field that the model concentrates on in learning the representation. Thus, many of the syntax-constrained models are instances of local attention-based models (see Section 6.2.4). In addition, the concept of local attention can be extended to develop a rich set of models, such as **sparse attention models**, although these models are often discussed in the context of efficient machine learning methods. We will see a few examples of them in Section 6.4.

In deep learning, one of the most widely used models for learning features from a restricted region of the input is CNNs. It is thus interesting to consider methods of combining CNNs and Transformer models to obtain the benefits of both approaches, for example, CNNs deal

with short-term dependencies, and self-attention models deal with long-term dependencies. One approach is to build a two-branch sequence model where one branch is based on CNNs and the other is based on self-attention models [Wu et al., 2020b]. Another approach is to incorporate CNN layers into Transformer blocks in some way that we can learn both local and global representations through a deep model [Wu et al., 2019; Gulati et al., 2020].

### 3. Relative Positional Embedding

Relative positional embedding, also known as **relative positional representation (RPR)**, is an improvement to the absolute positional embedding method used in standard Transformer systems [Shaw et al., 2018; Huang et al., 2018]. The idea of RPR is that we model the distance between two positions of a sequence rather than giving each position a fixed representation. As a result, we have a pair-wise representation  $\text{PE}(i, j)$  for any two positions  $i$  and  $j$ . One simple way to define  $\text{PE}(i, j)$  is to consider it as a lookup table for all pairs of  $i$  and  $j$ . More specifically, let  $\mathbf{u}_\pi$  be a  $d$ -dimensional representation for a given distance  $\pi$ . The form of  $\text{PE}(i, j)$  in the vanilla RPR method is given by

$$\text{PE}(i, j) = \mathbf{u}_{\text{clip}(j-i, k_{\text{rpr}})} \quad (6.71)$$

where  $\text{clip}(x, k_{\text{rpr}})$  is a function that clips  $x$  in the interval  $[-k_{\text{rpr}}, k_{\text{rpr}}]$

$$\text{clip}(x, k_{\text{rpr}}) = \max\{-k_{\text{rpr}}, \min\{x, k_{\text{rpr}}\}\} \quad (6.72)$$

Thus, we have a model with parameters

$$\mathbf{U}_{\text{rpr}} = \begin{bmatrix} \mathbf{u}_{-k_{\text{rpr}}} \\ \vdots \\ \mathbf{u}_0 \\ \vdots \\ \mathbf{u}_{k_{\text{rpr}}} \end{bmatrix} \quad (6.73)$$

While this matrix notation is used in a relatively informal way, we can view  $\mathbf{U}_{\text{rpr}}$  as a matrix  $\in \mathbb{R}^{(2k_{\text{rpr}}+1) \times d}$ , and select a row corresponding to  $\text{clip}(j-i, k_{\text{rpr}})$  when RPR is required for given  $i$  and  $j$ .

Using the above method, we can define three RPR models  $\text{PE}^q(i, j)$ ,  $\text{PE}^k(i, j)$  and  $\text{PE}^v(i, j)$  for queries, keys, and values, respectively. Then, following the form of Eq. (6.17), the output of the self-attention model at position  $i$  can be written as

$$\begin{aligned} \mathbf{c}_i &= \sum_{j=1}^m \alpha_{i,j} [\mathbf{h}_j^v + \text{PE}^v(i, j)] \\ &= \sum_{j=1}^m \alpha_{i,j} \mathbf{h}_j^v + \sum_{j=1}^m \alpha_{i,j} \text{PE}^v(i, j) \end{aligned} \quad (6.74)$$

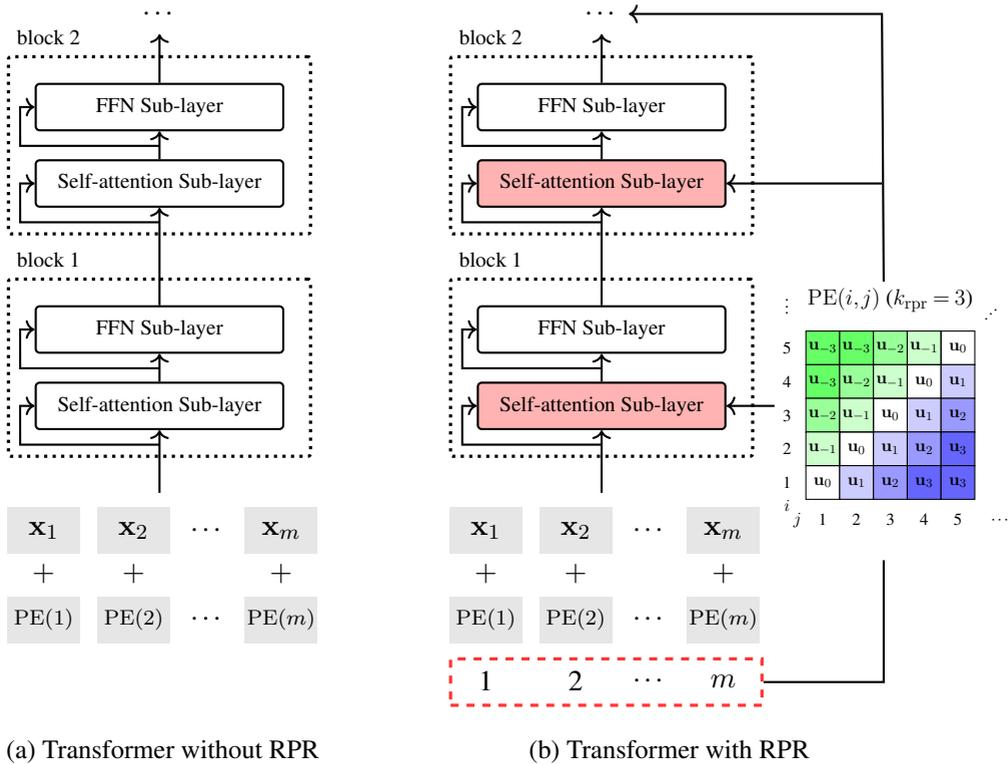


Figure 6.10: Transformer encoders without and with relative positional representation (RPR). In RPR, each pair of positions is represented as a vector  $PE(i, j)$  using a model parameterized by  $U_{\text{RPR}}$ .  $PE(i, j)$  is fed into each self-attention sub-layer so that we can make use of the positional information in intermediate steps of learning representations.

where  $\mathbf{h}_j^v$  is the  $j$ -th row vector of  $\mathbf{H}^v$ . This representation comprises two components:  $\sum_{j=1}^m \alpha_{i,j} \mathbf{h}_j^v$  is the basic representation, and  $\sum_{j=1}^m \alpha_{i,j} PE^v(i, j)$  is the positional representation.

The attention weight  $\alpha_{i,j}$  is computed in a regular way, but with additional terms  $PE^q(i, j)$  and  $PE^k(i, j)$  added to each query and key.

$$\alpha_{i,j} = \text{Softmax}\left(\frac{[\mathbf{h}_i^q + PE^q(i, j)][\mathbf{h}_j^k + PE^k(i, j)]^T}{\sqrt{d}}\right) \tag{6.75}$$

Figure 6.10 shows the Transformer encoder architectures with and without RPR. When RPR is adopted,  $PE^q(i, j)$ ,  $PE^k(i, j)$ ,  $PE^v(i, j)$  are directly fed to each self-attention sub-layer, and so we can make better use of positional information for sequence modeling. Note that, the use of the clipping function (see Eq. (6.72)) makes the modeling simple because we do not need to distinguish the relative distances for the cases  $|j - i| \geq k_{\text{rpr}}$ . This clipped distance-based model can lead, in turn, to better modeling in local context windows.

Eqs. (6.74) and (6.75) provide a general approach to position-sensitive sequence modeling. There are many variants of this model. In Shaw et al. [2018]’s early work on RPR, the

positional representations for queries are removed, and the model works only with  $\text{PE}^k(i, j)$  and  $\text{PE}^v(i, j)$ , like this

$$\alpha_{i,j} = \text{Softmax}\left(\frac{\mathbf{h}_i^q[\mathbf{h}_j^k + \text{PE}^k(i, j)]^T}{\sqrt{d}}\right) \quad (6.76)$$

By contrast, there are examples that attempt to improve the RPR model in computing attention weights but ignore  $\text{PE}^v(i, j)$  in learning values [Dai et al., 2019; He et al., 2021]. Instead of treating RPR as an additive term to each representation, researchers also explore other ways of introducing RPR into Transformer [Huang et al., 2020b; Raffel et al., 2020]. We refer the interested readers to these papers for more details.

### 6.3.2 Deep Models

Many state-of-the-art NLP systems are based on deep Transformer models. For example, recent large language models generally comprise tens of Transformer layers (or more precisely, hundreds of layers of neurons), demonstrating strong performance on many tasks [Ouyang et al., 2022; Touvron et al., 2023a]. By stacking Transformer layers, it is straightforward to obtain a deep model. However, as is often the case, training very deep neural networks is challenging. A difficulty arises from the fact that the error surfaces of deep neural networks are highly non-convex and have many local optima that make the training process likely to get stuck in them. While there are optimization algorithms that can help alleviate this problem, most of the practical efforts explore the use of gradient-based methods for optimizing deep neural networks. As a result, training a model with many Transformer layers becomes challenging due to vanishing and exploding gradients during back-propagation. Here we consider several techniques for training deep Transformer models.

#### 1. Re-thinking the Pre-Norm and Post-Norm Architectures

As introduced previously, a Transformer sub-layer is a residual network where a shortcut is created to add the input of the network directly to the output of this sub-layer. This allows gradients to flow more directly from the output back to the input, mitigating the vanishing gradient problem. In general, a residual connection in Transformer is used together with a layer normalization unit to form a sub-layer. This leads to two types of architecture, called post-norm and pre-norm. To be specific, recall from Section 6.1.4 that the post-norm architecture can be expressed as

$$\mathbf{z}^l = \text{LNorm}(F^l(\mathbf{z}^{l-1}) + \mathbf{z}^{l-1}) \quad (6.77)$$

where  $\mathbf{z}^l$  and  $\mathbf{z}^{l-1}$  are the output and input of the sub-layer  $l$ , and  $F^l(\cdot)$  is the core function of this sub-layer. The pre-norm architecture takes the identity mapping  $\mathbf{z}^l$  outside the layer normalization function, given in the form

$$\mathbf{z}^l = \text{LNorm}(F^l(\mathbf{z}^{l-1})) + \mathbf{z}^{l-1} \quad (6.78)$$

Consider the difference between the information flow in these two architectures:

- The post-norm architecture prevents the identity mapping of the input from adding to the output of the sub-layer. This is not a true residual network, because all the information is passed on through a non-linear function (i.e., the layer normalization unit). Thus, the post-norm architecture is not very “efficient” for back-propagation. Wang et al. [2019a] show that the gradient of the loss of an  $L$  sub-layer Transformer network with respect to  $\mathbf{z}^l$  is given by

$$\frac{\partial E}{\partial \mathbf{z}^l} = \frac{\partial E}{\partial \mathbf{z}^L} \cdot \prod_{k=l}^{L-1} \frac{\partial \text{LNorm}(\mathbf{v}^k)}{\partial \mathbf{v}^k} \cdot \prod_{k=l}^{L-1} \left( 1 + \frac{\partial F^k(\mathbf{z}^k)}{\partial \mathbf{z}^k} \right) \quad (6.79)$$

where  $\mathbf{z}^L$  is the output of the last layer,  $\mathbf{v}^k$  is a short for  $F^k(\mathbf{z}^{k-1})$ , and  $E$  is the error measured by some loss function.  $\frac{\partial \text{LNorm}(\mathbf{v}^k)}{\partial \mathbf{v}^k}$  and  $\frac{\partial F^k(\mathbf{z}^k)}{\partial \mathbf{z}^k}$  are the gradients of the layer normalization function and the core function, respectively. Although the equation here appears a bit complex, we see that  $\prod_{k=l}^{L-1} \frac{\partial \text{LNorm}(\mathbf{v}^k)}{\partial \mathbf{v}^k}$  is simply a product of  $L-l$  factors. This means that the error gradient will be rescaled more times if  $L$  becomes larger, and there is a higher risk of vanishing and exploding gradients for a deeper model.

- The pre-norm architecture describes a standard residual neural network where the input of a whole network is added to its output. We can write the gradient of the error at  $\mathbf{z}^l$  as

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{z}^l} &= \frac{\partial E}{\partial \mathbf{z}^L} \cdot \left( 1 + \prod_{k=l}^{L-1} \frac{\partial F^k(\text{LNorm}(\mathbf{z}^k))}{\partial \mathbf{z}^k} \right) \\ &= \frac{\partial E}{\partial \mathbf{z}^L} + \frac{\partial E}{\partial \mathbf{z}^L} \cdot \prod_{k=l}^{L-1} \frac{\partial F^k(\text{LNorm}(\mathbf{z}^k))}{\partial \mathbf{z}^k} \end{aligned} \quad (6.80)$$

It is easy to see that  $\frac{\partial E}{\partial \mathbf{z}^l}$  receives direct feedback regarding the errors made by the model, because the first term of the summation on the right-hand side (i.e.,  $\frac{\partial E}{\partial \mathbf{z}^L}$ ) is the gradient of the model output which is independent of the network depth.

The use of the pre-norm architecture also helps optimization during early gradient descent steps. For example, it has been found that pre-norm Transformer models can be trained by using a larger learning rate in the early stage of training instead of gradually increasing the learning rate from a small value [Xiong et al., 2020].

While the pre-norm architecture leads to easier optimization of deep Transformer models, we would not simply say that it is a better choice compared to the post-norm architecture. In fact, both post-norm and pre-norm Transformer models have been successfully used in many applications. For example, the post-norm architecture is widely used in BERT-like models, while the pre-norm architecture is a more popular choice in recent generative large language models. Broadly, these two architectures provide different ways to design a deep Transformer model, as well as different advantages and disadvantages in doing so. The post-norm architecture forces the representation to be learned through more non-linear functions,

but in turn results in a complicated model that is relatively hard to train. By contrast, the pre-norm architecture can make the training of Transformer models easier, but would be less expressive than the post-norm counterpart if the learned models are overly dependent on the shortcut paths.

An improvement to these architectures is to control the extent to which we want to “skip” a sub-layer. A simple way to do this is to weight different paths rather than treating them equally. For example, a scalar factor of a residual connection can be introduced to determine how heavily we weight this residual connection relative to the path of the core function [He et al., 2016b; Liu et al., 2020a;b]. A more general form of this model is given by

$$\mathbf{z}^l = \text{LNorm}(F^l(\mathbf{z}^{l-1}) + \beta \cdot \mathbf{z}^{l-1}) + \gamma \cdot \mathbf{z}^{l-1} \quad (6.81)$$

where  $\beta$  is the weight of the identity mapping inside the layer normalization function, and  $\gamma$  is the weight of the identity mapping outside the layer normalization function. Clearly, both the post-norm and pre-norm architectures can be seen as special cases of this equation. That is, if  $\beta = 1$  and  $\gamma = 0$ , then it will become Eq. (6.77); if  $\beta = 0$  and  $\gamma = 1$ , it will become Eq. (6.78). This model provides a multi-branch view of building residual blocks. The input to this block can be computed through multiple paths with different modeling complexities. When  $\beta$  and  $\gamma$  are small, the representation is forced to be learned through a “deep” model with multiple layers of cascaded non-linear units. In contrast, when  $\beta$  and  $\gamma$  are large, the representation is more likely to be learned using a “shallow” model with fewer layers. To determine the optimal choices of  $\beta$  and  $\gamma$ , one can give them fixed values by considering some theoretical properties or system performance on validation sets, or compute these values by using additional functions that can be trained to do so [Srivastava et al., 2015]. It should be emphasized that many other types of architecture can be considered in the design of a Transformer sub-layer. It is possible, for instance, to introduce more layer normalization units into a sub-layer [Ding et al., 2021; Wang et al., 2022b], or, on the contrary, to simply remove them from a sub-layer [Bachlechner et al., 2021].

## 2. Parameter Initialization

As with other deep neural networks, there is interest in developing parameter initialization methods for deep Transformer models in order to perform optimization on some region around a better local optimum. However, initialization is a wide-ranging topic for optimization of machine learning models, and the discussion of this general topic lies beyond the scope of this section. Here we will discuss some of the parameter initialization methods used in Transformer-based systems rather than the general optimization problems.

While the parameters of a neural network can be set in various different ways, most practical systems adopt simple techniques to give appropriate initial values of model parameters. Consider, for example, the Xavier initialization for a parameter matrix  $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$  [Glorot and Bengio, 2010]. We define a variable  $\eta$  by

$$\eta = \text{gain} \cdot \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}} \quad (6.82)$$

where gain is a hyper-parameter which equals 1 by default. Then, each entry of  $\mathbf{W}$  can be initialized by using a uniform distribution

$$W \sim U(-\eta, \eta) \quad (6.83)$$

or, alternatively, using a Gaussian distribution

$$W \sim \text{Gaussian}(0, \eta^2) \quad (6.84)$$

This method can be easily adapted to initialize Transformer models having a large number of layers. One common way is to find a more suitable value of gain by taking into account the fact that the initial states of optimization might be different for neural networks of different depths. For example, one can increase the value of gain as the depth of the model grows. Then, gain can be defined as a function of the network depth in the form

$$\text{gain} = a \cdot L^b \quad (6.85)$$

where  $a$  is the scalar, and  $L^b$  is the network depth raised to the power of  $b$ . Typically,  $a$  and  $b$  can be positive numbers, which means that it is preferred to have larger initial values for the parameters for deeper models. For example, Wang et al. [2022a] show that, by choosing appropriate values for  $a$  and  $b$ , a very deep Transformer model can be successfully trained.

Eq. (6.85) assigns gain the same value for all of the sub-layers. However, it is found that the norm of gradients becomes smaller when a sub-layer moves away from the output layer. This consistent application of gain across the entire model could result in under-training of the lower layers due to the gradient vanishing problem. For this reason, one can develop methods that are sensitive to the position of a sub-layer in the neural network. The general form of such methods is given by

$$\text{gain} = \frac{a}{l^b} \quad (6.86)$$

Here  $l$  denotes the depth of a sub-layer. If  $l$  is larger (i.e., the sub-layer is closer to the output), gain will be smaller and the corresponding parameters will be set to smaller values. An example of this method can be found in Zhang et al. [2019a]’s work.

It is also, of course, straightforward to apply general methods of initializing deep multi-layer neural networks to Transformer models. An example is to consider the **Lipschitz constant** in parameter initialization, which has been shown to help improve the stability of training deep models [Szegedy et al., 2014b; Xu et al., 2020]. Another approach is to use second-order methods to estimate the proper values of the parameters. For example, one can compute the Hessian of each parameter matrix to model its curvature [Skorski et al., 2021].

For models with a large number of layers, it is also possible to pre-train some of the layers via smaller models and use their trained parameters to initialize bigger models [Chen et al., 2015]. That is, we first obtain a rough estimation of the parameters in a cheap way, and then continue the training process on the whole model as usual. These methods fall into a class of

training methods, called **model growth** or **depth growth**.

As a simple example, consider a Transformer model (e.g., a Transformer encoder) of  $2L$  sub-layers. We can train this model by using the **shallow-to-deep training** method [Li et al., 2020b]. First, we train an  $L$ -sub-layer model (call it the shallow model) in a regular way. Then, we create a  $2L$ -sub-layer model (call it the deep model) by stacking the shallow model twice, and further train this deep model. To construct deeper models, this procedure can be repeated multiple times, say, we start with a model of  $L$  sub-layers, and obtain a model of  $L^{2^I}$  after  $I$  iterations. Note that many of the pre-training models are used in the same manner. For example, for BERT-like methods, a transformer encoder is trained on large-scale data, and the optimized parameters are then used to initialize downstream systems.

### 3. Layer Fusion

Another problem with training a deep Transformer model is that the prediction is only conditioned on the last layer of the neural network. While the use of residual connections enables the direct access to lower-level layers from a higher-level layer, there is still a “long” path of passing information from the bottom to the top. One simple way to address this is to create residual connections that skip more layers. For example, consider a group of  $L$  Transformer sub-layers. For the sub-layer at depth  $l$ , we can build  $l - 1$  residual connections, each connecting this sub-layer with a previous sub-layer. In this way, we develop a densely connected network where each sub-layer takes the outputs of all previous sub-layers [Huang et al., 2017a]. The output of the last sub-layer can be seen as some combination of the outputs at different levels of representation of the input.

Following the notation used in the previous subsections, we denote the output of the sub-layer at depth  $l$  by  $\mathbf{z}^l$ , and denote the function of the sub-layer by  $\text{Layer}^l(\cdot)$ . Then,  $\mathbf{z}^l$  can be expressed as

$$\mathbf{z}^l = \text{Layer}^l(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) \quad (6.87)$$

We can simply view  $\text{Layer}^l(\cdot)$  as a function that fuses the information from  $\{\mathbf{z}^1, \dots, \mathbf{z}^{l-1}\}$ . There are many possible choices for  $\text{Layer}^l(\cdot)$ . For example, a simple form of  $\text{Layer}^l(\cdot)$  is given by

$$\text{Layer}^l(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \text{LNorm}(F^l(\mathbf{Z}^l)) \quad (6.88)$$

$$\mathbf{Z}^l = \phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) \quad (6.89)$$

Here  $\phi(\cdot)$  takes the layer outputs  $\{\mathbf{z}^1, \dots, \mathbf{z}^{l-1}\}$  and fuses them into a single representation  $\mathbf{Z}^l$ . A simple instance of  $\phi(\cdot)$  is average pooling which computes the sum of  $\{\mathbf{z}^1, \dots, \mathbf{z}^{l-1}\}$  divided by  $l - 1$ . See Table 6.2 for more examples of  $\phi(\cdot)$ .

Taking a similar architecture of a Transformer sub-layer, we can also consider a post-norm

Entry	Function
Average Pooling	$\phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \frac{1}{l-1} \sum_{k=1}^{l-1} \mathbf{z}^k$
Weighted Sum	$\phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \sum_{k=1}^{l-1} \text{weight}_k \cdot \mathbf{z}^k$
Feedforward Network	$\phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \text{FFN}([\mathbf{z}^1, \dots, \mathbf{z}^{l-1}])$
Self Attention	$\phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \text{FFN}([\text{Att}_{\text{self}}(\mathbf{z}^1, \dots, \mathbf{z}^{l-1})])$

Table 6.2: Fusion functions.  $\text{FFN}(\cdot)$  = feedforward neural network,  $[\cdot]$  = concatenating the input vectors, and  $\text{Att}_{\text{self}}(\cdot)$  = self-attention function. All of the fusion functions can be followed by a layer normalization function, for example, we can write the weighted sum of  $\{\mathbf{z}^1, \dots, \mathbf{z}^{l-1}\}$  as  $\phi(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \text{LNorm}(\sum_{k=1}^{l-1} \text{weight}_k \cdot \mathbf{z}^k)$ .

form

$$\text{Layer}^l(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \text{LNorm}(\mathbf{Z}^l) \quad (6.90)$$

$$\mathbf{Z}^l = \phi(F^l(\mathbf{z}^{l-1}), \mathbf{z}^1, \dots, \mathbf{z}^{l-1}) \quad (6.91)$$

or a pre-norm form

$$\text{Layer}^l(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = \mathbf{Z}^l \quad (6.92)$$

$$\mathbf{Z}^l = \phi(\text{LNorm}(F^l(\mathbf{z}^{l-1})), \mathbf{z}^1, \dots, \mathbf{z}^{l-1}) \quad (6.93)$$

These models are very general. For example, a standard post-norm encoder sub-layer can be recovered as a special case of Eqs. (6.90-6.91), if we remove the dependencies of sub-layers from 1 to  $l-2$ , and define  $\phi(\cdot)$  to be

$$\phi(F^l(\mathbf{z}^{l-1}), \mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = F^l(\mathbf{z}^{l-1}) + \mathbf{z}^{l-1} \quad (6.94)$$

Densely connected network makes the information easier to flow through direct connections between sub-layers, but the resulting models are a bit more complex, especially when we use parameterized fusion functions. In practice, we typically add dense connections only to some of the sub-layers, and so the overall networks are not very dense. For example, we only add connections from bottom sub-layers to the last few sub-layers. Thus, the prediction can be made by having direct access to different levels of representation [Wang et al., 2018a].

#### 4. Regularization

In machine learning, regularization is used to avoid overfitting in training deep neural networks. It is therefore straightforward to apply regularization techniques to Transformer models. Since the regularization issue has been discussed in Chapter 2, here we consider some of the methods that have not been covered yet in this book but could be used for training deep Transformer models.

One approach to regularizing a deep Transformer model is to randomly skip sub-layers or layers during training [Huang et al., 2016; Pham et al., 2019]. In each run of the model, such as running the backpropagation algorithm on a batch of samples, we select each of the

sub-layers with a probability  $\rho$ , and stack the selected sub-layers to form a “new” model. Thus, we essentially train different neural networks with shared architectures and parameters on the same dataset. In this way, a sub-layer learns to operate somewhat independently, and so overfitting is reduced by preventing the co-adaptation of sub-layers. In fact, dropping out sub-layers (or layers) and dropping out neurons are two different methods on a theme. Sometimes, the method described here is called **sub-layer dropout** or **layer dropout**.

At test time, we need to combine all the possible networks to make predictions of some output. A simple method to achieve this is to rescale the outputs of the stochastic components of the model [Li et al., 2021a]. As an example, suppose each sub-layer has a pre-norm architecture. Then, the output of the sub-layer at depth  $l$  is given by

$$\mathbf{z}^l = \rho \cdot \text{LNorm}(F^l(\mathbf{z}^{l-1})) + \mathbf{z}^{l-1} \quad (6.95)$$

Another idea is to force the parameters to be shared across sub-layers. One of the simplest methods is to use the same parameters for all the corresponding sub-layers [Dehghani et al., 2018], for example, all the FFN sub-layers are based on the same feedforward network. This method has a similar effect as the methods that add norms of parameter matrices to the loss function for penalizing complex models. For practical systems, there can be significant benefit in adopting a shared architecture because we can reuse the same sub-model to build a multi-layer neural network and reduce the memory footprint. We will see more discussions on the efficiency issue in Section 6.4.4.

### 6.3.3 Numerical Method-Inspired Models

A residual network computes its output through the sum of the identity mapping and some transformation of the input. Such a model can be interpreted as an Euler discretization of **ordinary differential equations (ODEs)** [Ee, 2017; Haber and Ruthotto, 2017]. To illustrate this idea, we consider a general form of residual networks

$$\mathbf{z}^l = f^l(\mathbf{z}^{l-1}) + \mathbf{z}^{l-1} \quad (6.96)$$

where  $f^l(\mathbf{z}^{l-1})$  denotes a function takes an input variable  $\mathbf{z}^{l-1}$  and produces an output variable in the same space. Clearly, a Transformer sub-layer is a special case of this equation. For example, for pre-norm Transformer, we have  $f^l(\cdot) = \text{LNorm}(F^l(\cdot))$ .

For notational simplicity, we rewrite the above equation in an equivalent form

$$\mathbf{z}(l) = f(\mathbf{z}(l-1), l) + \mathbf{z}(l-1) \quad (6.97)$$

We use the notations  $\mathbf{z}(l)$  and  $f(\mathbf{z}(\cdot), l)$  to emphasize that  $\mathbf{z}(\cdot)$  and  $f(\cdot)$  are functions of  $l$ . Here we assume that  $l$  is a discrete variable. If we relax  $l$  to a continuous variable and  $\mathbf{z}(l)$  to a continuous function of  $l$ , then we can express Eq. (6.97) as

$$\mathbf{z}(l) = \Delta l \cdot f(\mathbf{z}(l - \Delta l), l) + \mathbf{z}(l - \Delta l) \quad (6.98)$$

This can be further written as

$$\frac{\mathbf{z}(l) - \mathbf{z}(l - \Delta l)}{\Delta l} = f(\mathbf{z}(l - \Delta l), l) \quad (6.99)$$

Taking the limit  $\Delta l \rightarrow 0$ , we have an ODE

$$\frac{d\mathbf{z}(l)}{dl} = f(\mathbf{z}(l), l) \quad (6.100)$$

We say that a pre-norm Transformer sub-layer (i.e., Eqs. (6.97) and (6.96)) is an Euler discretization of solutions to the above ODE. This is an interesting result! A sub-layer is actually a solver of the ODE.

Eqs. (6.97) and (6.96) are standard forms of the **Euler method**. It computes a new estimation of the solution by moving from an old estimation one step forward along  $l$ . In general, two dimensions can be considered in design of numerical methods for ODEs.

- **Linear Multi-step Methods.** A linear multi-step method computes the current estimation of the solutions by taking the estimations and derivative information from multiple previous steps. A general formulation of  $p$ -step methods can be expressed as

$$\mathbf{z}(l) = \sum_{i=1}^p a_i \cdot \mathbf{z}(l-i) + h \sum_{i=1}^{p+1} b_i \cdot f(\mathbf{z}(l-i), l-i+1) \quad (6.101)$$

where  $h$  is the size of the step we move each time<sup>8</sup>, that is,  $\Delta l$  in Eqs. (6.98) and (6.99).  $\{a_i\}$  and  $\{b_i\}$  are coefficients of the solution points and derivatives in the linear combination. Given this definition, we can think of the Euler method as a single-step, low-order method of solving ODEs<sup>9</sup>.

- **(Higher-order) Runge-Kutta Methods.** Runge-Kutta (RK) methods and their variants provide ways to compute the next step solution by taking intermediate results in solving an ODE. As a result, we obtain higher-order methods but still follow the form of single-step methods, that is, the estimated solution is dependent only on  $\mathbf{z}(l-1)$  rather than on the outputs at multiple previous steps.

In fact, linear multi-step methods, though not explicitly mentioned, have been used in layer fusion discussed in Section 6.3.2. For example, taking Eqs. (6.92) and (6.93) and a linear fusion function, a pre-norm sub-layer with dense connections to all previous sub-layers can be expressed as

$$\text{Layer}^l(\mathbf{z}^1, \dots, \mathbf{z}^{l-1}) = a_1 \cdot \mathbf{z}^{l-1} + \dots + a_{l-1} \cdot \mathbf{z}^1 + b_1 \cdot \text{LNorm}(F^l(\mathbf{z}^{l-1})) \quad (6.102)$$

<sup>8</sup>Let  $\{t_0, \dots, t_i\}$  denote the values of the variable  $l$  at steps  $\{0, \dots, i\}$ . In linear multi-step methods, it is assumed that  $t_i = t_0 + ih$ .

<sup>9</sup>In numerical analysis, the **local truncation error** of a method of solving ODEs at a step is defined to be the difference between the approximated solution computed by the method and the true solution. The method is called order  $p$  if it has a local truncation error  $O(h^{p+1})$ .

This equation is an instance of Eq. (6.101) where we set  $h = 1$  and remove some of the terms on the right-hand side.

It is also straightforward to apply Runge-Kutta methods to Transformer [Li et al., 2022a]. Given an ODE as described in Eq. (6.100), an explicit  $p$ -order Runge-Kutta solution is given by

$$\mathbf{z}(l) = \mathbf{z}(l-1) + \sum_{i=1}^p \gamma_i \cdot \mathbf{g}_i \quad (6.103)$$

$$\mathbf{g}_i = h \cdot f\left(\mathbf{z}(l-1) + \sum_{j=1}^{i-1} \beta_{i,j} \cdot \mathbf{g}_j, l-1 + \lambda_i \cdot h\right) \quad (6.104)$$

Here  $\mathbf{g}_i$  represents an intermediate step which is present only during the above process.  $\{\gamma_i\}$ ,  $\{\beta_{i,j}\}$  and  $\{\lambda_i\}$  are coefficients that are determined by using the Taylor series of  $\mathbf{z}(l)$ . To simplify the model, we assume that the same function  $f$  is used for all  $\{\mathbf{g}_i\}$ . Then, we remove the dependency of the term  $l-1 + \lambda_i \cdot h$  in  $f$ , and rewrite Eq. (6.104) as

$$\mathbf{g}_i = h \cdot f\left(\mathbf{z}(l-1) + \sum_{j=1}^{i-1} \beta_{i,j} \cdot \mathbf{g}_j\right) \quad (6.105)$$

where  $f(\cdot)$  is a function independent of  $i$ .

As an example, consider the 4th-order Runge-Kutta (RK4) solution

$$\mathbf{z}(l) = \mathbf{z}(l-1) + \frac{1}{6}(\mathbf{g}_1 + 2\mathbf{g}_2 + 2\mathbf{g}_3 + \mathbf{g}_4) \quad (6.106)$$

$$\mathbf{g}_1 = h \cdot f(\mathbf{z}(l-1)) \quad (6.107)$$

$$\mathbf{g}_2 = h \cdot f\left(\mathbf{z}(l-1) + \frac{1}{2}\mathbf{g}_1\right) \quad (6.108)$$

$$\mathbf{g}_3 = h \cdot f\left(\mathbf{z}(l-1) + \frac{1}{2}\mathbf{g}_2\right) \quad (6.109)$$

$$\mathbf{g}_4 = h \cdot f(\mathbf{z}(l-1) + \mathbf{g}_3) \quad (6.110)$$

These equations define a new architecture of sub-layer. For example, by setting  $h = 1$  and  $f(\cdot) = \text{LNorm}(F^l(\cdot))$ , we obtain an RK4 Transformer sub-layer, as shown in Figure 6.11. This method leads to a deep model because each sub-layer involves four runs of  $f(\cdot)$  in sequence. On the other hand, the resulting model is parameter efficient because we reuse the same function  $f(\cdot)$  within the sub-layer, without introducing new parameters.

So far in this subsection our discussion has focused on applying dynamic systems to Transformer models by designing architectures of Transformer sub-layers. While the basic ODE model is continuous with respect to the depth  $l$ , these methods still follow the general framework of neural networks in which  $l$  is a discrete variable and the representational power of the models is largely determined by this hyper-parameter. An alternative approach is to use neural ODE models to relax the “depth” to a truly continuous variable. In this way, we can have a model with continuous depth for computing the solution of ODEs. However, as the

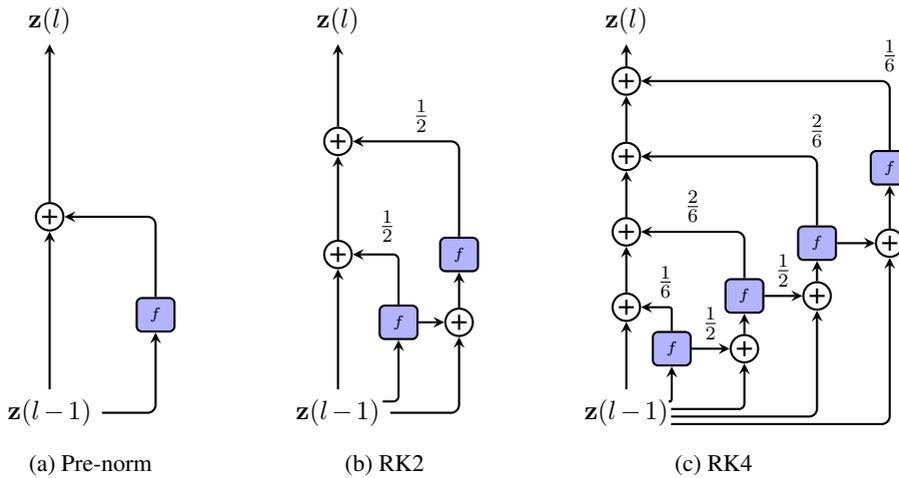


Figure 6.11: Pre-norm (a) and Runge-Kutta (b and c) sub-layer architectures.  $\mathbf{z}(l-1)$  denotes the input of a sub-layer at depth  $l$ ,  $\mathbf{z}(l)$  denotes the output of the sub-layer, and  $f$  (in blue boxes) denotes the function  $f(\cdot) = \text{LNORM}(F^l(\cdot))$

discussion of neural ODE lies beyond the scope of this chapter, we refer the reader to related papers for more details [Chen et al., 2018c; Kidger, 2022].

### 6.3.4 Wide Models

Most of the methods that we have studied so far in this section are examples of learning and using deep models. Another design choice we generally face is to determine the width for a neural network. Typically, the **width** of a Transformer model can be defined as the number of dimensions of a representation at some position of the input sequence, that is, the parameter  $d$ . Increasing this width is a common method to obtain a more complex and more powerful model. For example, in Vaswani et al. [2017]’s work, a wide model (called Transformer big) leads to significant improvements in translation quality for machine translation systems. More recently, wider models have been proposed to boost systems on large-scale tasks [Lepikhin et al., 2021; Fedus et al., 2022b].

However, developing very wide Transformer models is difficult. One difficulty is that training such systems is computationally expensive. While the number of the model parameters (or model size) grows linearly with  $d$ , the time complexity of the models grows quadratic with  $d$  (see Table 6.1). In some NLP tasks, it is found empirically that the training effort that we need to obtain satisfactory performance is even an exponential function of the model size [Kaplan et al., 2020]. These results suggest ways to improve the efficiency of training when we enlarge  $d$ .

One simple method is to incrementally grow the model along the dimension of  $d$ , rather than training the model from scratch. Suppose we have an initial model involving a  $d_1 \times d_1$  parameter matrix  $\mathbf{W}_1$ , for example, the linear transformation of each query or key in some layer. We can train this model to obtain optimized  $\mathbf{W}_1$  in a regular way. Then, we want to extend this model to a wider model where  $\mathbf{W}_1$  is replaced by a  $d_2 \times d_2$  parameter matrix  $\mathbf{W}_2$ .

Let us assume for simplicity that  $d_2 = kd_1$ . There are several ways to expand a  $d_1 \times d_1$  matrix to a  $kd_1 \times kd_1$  matrix. The simplest of these may be to use  $\mathbf{W}_1$  to fill  $\mathbf{W}_2$ . We can write  $\mathbf{W}_2$  in the form

$$\mathbf{W}_2 = \begin{matrix} & & k \text{ times} & & \\ & & \left[ \begin{array}{ccc} \mathbf{W}_1 & \dots & \mathbf{W}_1 \\ \rho & & \rho \\ \vdots & & \vdots \\ \mathbf{W}_1 & \dots & \mathbf{W}_1 \\ \rho & & \rho \end{array} \right] & & \\ & & k \text{ times} & & \end{matrix} \quad (6.111)$$

where  $\rho$  is a hyper-parameter that is used to control the norm of  $\mathbf{W}_2$ . For example, if  $\rho = k$ ,  $\mathbf{W}_2$  will have the same  $l_1$  norm as  $\mathbf{W}_1$ . The above equation provides a good starting point for training the wide model, and we can train  $\mathbf{W}_2$  as usual after initialization. The procedure can be repeated a number of times for constructing a model with arbitrary width. Both this method and the depth growth method described in Section 6.3.2 are instances of the general method of model growth. In other words, we can obtain a larger model by extending a small model either vertically or horizontally, or both. Alternative methods for transforming  $\mathbf{W}_2$  to  $\mathbf{W}_1$  involve those considering other mathematical properties of the transformation [Chen et al., 2015]. These models can fall under the reusable neural networks where we are concerned with models and algorithms for transferring parameters from small models to (significantly) larger models [Wang et al., 2023b].

A second difficulty in building a wide Transformer model is the large memory requirement. Since the feedforward network generally has a larger hidden layer than other parts of the model, it demands relatively more memory as the model becomes wider. Consider the feedforward network described in Section 6.1.5

$$\begin{aligned} \mathbf{H}_{\text{out}} &= \text{FFN}(\mathbf{H}_{\text{in}}) \\ &= \text{ReLU}(\mathbf{H}_{\text{in}} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_f + \mathbf{b}_f \end{aligned} \quad (6.112)$$

where  $\mathbf{W}_h \in \mathbb{R}^{d \times d_{\text{ffn}}}$  and  $\mathbf{W}_f \in \mathbb{R}^{d_{\text{ffn}} \times d}$  are the parameters of the linear transformations.  $d_{\text{ffn}}$  is typically several times larger than  $d$ . Therefore,  $\mathbf{W}_h$  and  $\mathbf{W}_f$  will occupy the model if  $d$  and  $d_{\text{ffn}}$  have very large values.

In some cases, the size of the feedforward network may exceed the memory capacity of a single device. This problem can be addressed by using the **mixture-of-experts (MoE)** models [Shazeer et al., 2017]. An MoE model consists of  $M$  expert models  $\{e_1(\cdot), \dots, e_M(\cdot)\}$ . Given an input  $\mathbf{h}_{\text{in}} \in \mathbb{R}^d$ , each expert model produces an output  $e_k(\mathbf{h}_{\text{in}})$ . The output of the MoE model is a linear combination of  $\{e_1(\mathbf{h}_{\text{in}}), \dots, e_M(\mathbf{h}_{\text{in}})\}$ , given by

$$\mathbf{h}_{\text{out}} = \sum_{i=1}^M g_i(\mathbf{h}_{\text{in}}) \cdot e_i(\mathbf{h}_{\text{in}}) \quad (6.113)$$

where  $g(\cdot)$  is a gating model (also called **routing model**). Its output is a vector  $g(\mathbf{h}_{\text{in}}) = [g_1(\mathbf{h}_{\text{in}}) \ \dots \ g_M(\mathbf{h}_{\text{in}})]$  in which each entry  $g_i(\mathbf{h}_{\text{in}})$  indicates the weight of the corresponding

expert model. In many applications, it is assumed that  $g(\mathbf{h}_{\text{in}})$  is a sparse vector. This means that only a small number of expert models are involved in computing the output. A widely-used form of  $g(\mathbf{h}_{\text{in}})$  is given by using the Softmax layer

$$g(\mathbf{h}_{\text{in}}) = \text{Softmax}(\mathbf{h}_{\text{in}} \cdot \mathbf{W}_g) \quad (6.114)$$

where  $\mathbf{W}_g \in \mathbb{R}^{d \times M}$  is the parameter matrix of the layer. To enforce sparsity on  $g(\mathbf{h}_{\text{in}})$ , we can simply select the top- $k$  entries of  $g(\mathbf{h}_{\text{in}})$ , that is, we set non-top- $k$  entries to 0. An alternative method is to first perform top- $k$  selection on  $\mathbf{h}_{\text{in}} \cdot \mathbf{W}_g$  and then normalize the top- $k$  entries using the Softmax function.

Let  $\pi$  be the set of the indices of the top- $k$  expert models. The MoE model with top- $k$  routing has the following form

$$\mathbf{h}_{\text{out}} = \sum_{i \in \pi} g_i(\mathbf{h}_{\text{in}}) \cdot e_i(\mathbf{h}_{\text{in}}) \quad (6.115)$$

An advantage of this approach is that we can distribute different expert models to different processors, making it possible to execute these models on parallel computing machines. In each run of the MoE model, either during training or inference, we only need to activate and use  $k$  expert models rather than all of the expert models. In this way, the MoE approach is automatically learning a sparse model by limiting the number of active expert models each time in training and inference. The sparsity is determined by the hyperparameter  $k$ , say, a small value of  $k$  leads to a sparse model, and a large value of  $k$  leads to a dense model.

Let us return to the discussion of Eq. (6.112). It is straightforward to apply the MoE approach to feedforward neural networks. To simplify the discussion, consider the linear transformation of the first layer as shown in Eq. (6.112), that is,  $\mathbf{H}_{\text{in}} \cdot \mathbf{W}_h$ . We can approximate  $\mathbf{H}_{\text{in}} \cdot \mathbf{W}_h$  in an MoE form

$$\begin{aligned} \mathbf{H}_{\text{in}} \cdot \mathbf{W}_h &\approx \sum_{i \in \pi} g_i(\mathbf{H}_{\text{in}}) \cdot e_i(\mathbf{H}_{\text{in}}) \\ &= \sum_{i \in \pi} g_i(\mathbf{H}_{\text{in}}) \cdot [\mathbf{H}_{\text{in}} \cdot \mathbf{W}_h^i] \end{aligned} \quad (6.116)$$

Here  $\mathbf{W}_h$  is divided into  $M$  slides (or sub-matrices)  $\{\mathbf{W}_h^1, \dots, \mathbf{W}_h^M\}$ , written as

$$\mathbf{W}_h = \begin{bmatrix} \mathbf{W}_h^1 & \dots & \mathbf{W}_h^M \end{bmatrix} \quad (6.117)$$

Hence each expert model  $e_i(\mathbf{H}_{\text{in}}) = \mathbf{H}_{\text{in}} \cdot \mathbf{W}_h^i$  solves a sub-problem of the original linear mapping, and Eq. (6.116) can be thought of as a divide-and-conquer solution to the matrix multiplication problem.

We can, of course, treat any feedforward neural network as an expert model, resulting in

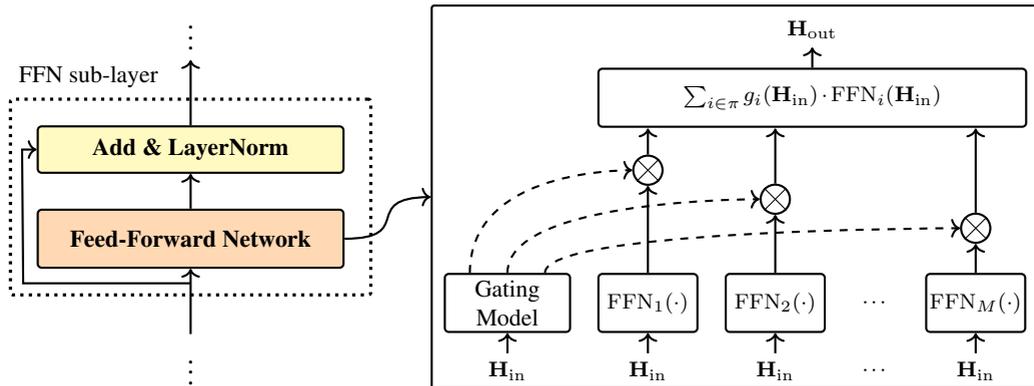


Figure 6.12: An illustration of the MoE model applied to an FFN sub-layer. There are  $M$  FFNs (call them expert models) and a gating model. Each FFN is weighted by the gating model. The output of the model is the sum of the weighted outputs of the top- $k$  FFNs (denoted by  $\pi$ ). Because these FFNs work independently and can be placed on different computing devices, the model can be easily scaled up as  $M$  is larger.

the following model

$$\mathbf{H}_{out} = \sum_{i \in \pi} g_i(\mathbf{H}_{in}) \cdot \text{FFN}_i(\mathbf{H}_{in}) \quad (6.118)$$

where  $\text{FFN}_i(\cdot)$  is a “small” feedforward neural network that has the same form as Eq. (6.112). This model is illustrated with an example in Figure 6.12. In practical implementations, all these expert models can be run in parallel on different devices, and so the resulting system is efficient.

Note that, from a perspective of machine learning, MoE is a general approach to combining different neural networks, each of which is developed to address a different aspect of the problem [Yukselet al., 2012; Masoudnia and Ebrahimipour, 2014]. The application here is just a special instance of the general framework of MoE. The approach is also often used to improve the overall performance of predictors, which can be discussed in the field of ensemble learning [Zhou, 2012a].

Another difficulty in developing large Transformer models is the training instability problem. As with many other large neural networks, straightforward optimization of a Transformer model with a large number of parameters may lead to getting trapped in local minimums, and, occasionally, large spikes in the loss during training [Lepikhin et al., 2021; Fedus et al., 2022b; Chowdhery et al., 2022]. Even with careful choices about hyperparameters, training strategies, and initial model parameters, we still encounter the situation that we have to restart the training at some point in order to jump out of the tough regions in optimization. One of the reasons for this training difficulty is that the usual implementations of the linear algebra operations, such as matrix multiplication, will be numerically unstable if they operate on very large vectors and matrices. It is therefore possible to improve the training by considering numerically stable methods instead.

## 6.4 Efficient Models

Efficiency is an important consideration for many practical applications of Transformer models. For example, we may wish to run and/or train a Transformer model given memory and time constraints. Efficiency is not a single problem, but covers a wide range of problems. While these problems can be categorized in several different ways, there are two fundamental aspects one may consider in an efficiency problem.

- **Time and Space Efficiencies.** For a given problem, we wish the model to be small and fast, and meanwhile to be as accurate as possible in solving the problem. For example, in some machine translation applications, we may learn a model with a small number of parameters to fit the model to limited memory, and may develop a fast search algorithm to achieve low-latency translation. A practical difficulty here is that improving efficiency often leads to worse predictions. In many cases, we need to seek a trade-off between efficiency and accuracy.
- **Scalability.** When the problem is scaled up, we wish that the additional effort we made for solving this problem is as small as possible. For example, the training of a neural network is called efficient if it takes a reasonably short time to optimize it as more training samples are involved. Another example of efficiency is that used to measure the amount of resources consumed in processing more inputs. For example, a machine translation system is inefficient in translating long sentences if the memory footprint and latency grow exponentially with the number of input words.

In this section, we will not discuss all the issues related to efficiency, which is a very broad topic. We instead consider the widely-used efficient approaches to Transformer-based sequence modeling and generation, some of which are refinements of model architectures, and some of which are model-free approaches and could be used in other systems as well. Most of the discussions here are focused on developing lightweight and fast Transformer models that are relatively robust to long input and output sequences.

In general, the same optimization method can be applied to different modules of a Transformer system. To simplify the discussion, we will mostly consider self-attention sub-layers and FFN sub-layers in this section. Our discussion, however, is general and the methods presented here can be applied to other parts of a Transformer system, for example, cross-attention sub-layers.

### 6.4.1 Sparse Attention

In practice, the attention approaches used in Transformer are time consuming, especially when the input sequences are long. To illustrate, consider a Transformer decoder that predicts a distribution of words at a time given the previous words. Suppose the sequence generated by the decoder is of size  $n$  and the input of a self-attention sub-layer is an  $n \times d$  matrix  $\mathbf{S}$ . First,  $\mathbf{S}$  is linearly transformed to obtain the queries  $\mathbf{S}^q \in \mathbb{R}^{n \times d}$ , keys  $\mathbf{S}^k \in \mathbb{R}^{n \times d}$ , and values  $\mathbf{S}^v \in \mathbb{R}^{n \times d}$ . To simplify the notation in this subsection, we use  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$  to represent  $\mathbf{S}^q$ ,  $\mathbf{S}^k$ , and  $\mathbf{S}^v$ , respectively.

The output of the self-attention sub-layer can then be computed using

$$\text{Att}_{\text{self}}(\mathbf{S}) = \mathbf{A}\mathbf{V} \quad (6.119)$$

where  $\mathbf{A}$  is an  $n \times n$  attention matrix or attention map

$$\mathbf{A} = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \mathbf{M}\right) \quad (6.120)$$

$\mathbf{M}$  is a masking matrix that is used to prevent the model from seeing the right context words at each position, that is, for a position  $i$ ,  $M(i, j) = 0$  for  $j \leq i$ , and  $M(i, j) = -\infty$  otherwise. Both the time and space complexities of the self-attention sub-layer are quadratic functions of  $n$ <sup>10</sup>. Therefore, if  $n$  is large, the model would be computationally expensive.

The usual implementation of the above model depends on dense matrix computation, for example, the dense matrix multiplications in Eqs. (6.119-6.120). One approach to reducing the amount of memory and the number of floating-point calculations in a dense computation system is to sparsify the problem. To do this, we assume that  $\mathbf{A}$  is a sparse matrix, for example, only  $\varrho \cdot n^2$  entries of  $\mathbf{M}$  have non-zero values, where  $\varrho$  indicates how sparse the matrix is, also called **sparsity ratio**. Since we only need to store these non-zero entries, the memory requirement of  $\mathbf{A}$  can be reduced by using sparse matrix representations. Another advantage of using a sparse attention matrix is that the models of  $\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}$  and  $\mathbf{A}\mathbf{V}$  can be simplified, as we consider only a “small” number of related positions when learning a representation.

Given a position  $i$ , we define the **attention field**  $\pi_i$  to be the set of positions that are considered in computing the representation at this position. We therefore only need to compute the dot-product attention between the given position  $i$  and each position  $j \in \pi_i$ . This results in a sparse attention matrix  $\mathbf{A}'$  where

$$A'(i, j) = \begin{cases} \text{some weight} & j \in \pi_i \text{ and } j \leq i \\ 0 & \text{otherwise} \end{cases} \quad (6.121)$$

A simple implementation of this model involves a slight modification to  $\mathbf{M}$ , leading to a new masking variable  $\mathbf{M}'$

$$M'(i, j) = \begin{cases} 0 & j \in \pi_i \text{ and } j \leq i \\ -\infty & \text{otherwise} \end{cases} \quad (6.122)$$

In practical implementation, a more efficient approach is to employ sparse operations for  $\mathbf{Q}\mathbf{K}^T$  and  $\mathbf{A}'\mathbf{V}$  by considering  $\mathbf{M}'$  and  $\mathbf{A}'$ , respectively. That is, we save on computation for pairs of positions whose attention weights are non-zero, and skip the rest.

There are several approaches that we can take to the sparse modeling of self-attention. We describe briefly some of them as follows

<sup>10</sup>More precisely, the amount of memory used by the self-attention function is  $n^2 + n \cdot d$ , and so it will be dominated by the quadratic term  $n^2$  if  $n \gg d$ .

- **Span-based Attention/Local Attention.** As discussed in Section 6.3.1, the use of context in sequence modeling is local in many cases. The basic idea of local attention is to span the attention weights to a restricted region of the input sequence. We can then write  $\pi_i$  as

$$\pi_i = [a_i^l, a_i^r] \quad (6.123)$$

where  $a_i^l$  and  $a_i^r$  are the left and right ends of  $\pi_i$ .  $a_i^r - a_i^l + 1$  determines how small the region is, and so we can use it to control the sparsity of the attention model, for example, if  $a_i^r - a_i^l + 1 \ll n$ , the model would be very sparse.  $a_i^l$  and  $a_i^r$  can be obtained by using either heuristics or machine learning methods. The reader may refer to related papers for more details [Luong et al., 2015; Sperber et al., 2018; Yang et al., 2018a; Sukhbaatar et al., 2019]. See Figure 6.13 (b) for an illustration of local attention.

- **Chunked Attention.** When a problem is too difficult to solve, one can transform it into easier problems and solve each of them separately, as is often the case in practice. This motivates the chunked attention approach in which we segment a sequence into chunks and run the attention model on each of them [Parmar et al., 2018; Qiu et al., 2020a]. Given a sequence  $\{1, \dots, n\}$ , we define  $\{\text{chunk}_1, \dots, \text{chunk}_q\}$  to be a segmentation of the sequence. A chunk can be expressed as a span

$$\text{chunk}_k = [c_k^l, c_k^r] \quad (6.124)$$

In the attention step, we treat each chunk as a sequence and perform self-attention on it as usual. In other words, the representation at position  $i$  is computed by using only the context in the chunk that  $i$  belongs to. In this sense, this model can be thought of as some sort of local attention model. Figure 6.13 (c) shows an illustration of this model. There remains the issue of how to segment the sequence. There are several ways to do this. For example, as discussed in Section 6.2.4, we can do segmentation from a linguistic perspective, and segment the sequence into linguistically motivated units. In practical systems, it is sometimes more convenient to segment the sequence into chunks that are of equal length. Thus, the sparsity of the model is controlled by the size of these chunks, for example, the use of smaller chunks would lead to a more sparse attention model.

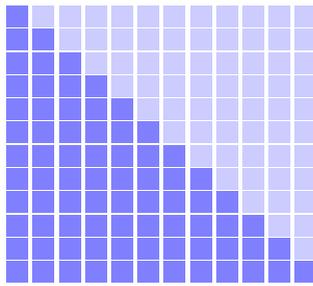
- **Strided Attention.** Since the chunked attention approach enforces a hard segmentation on the input sequence, it may lose the ability to learn representations from inputs in different chunks. An alternative way to achieve chunk-wise attention is to allow overlap between chunks [Child et al., 2019; Beltagy et al., 2020; Ainslie et al., 2020]. This approach is analogous to the family of approaches that are commonly used to apply a local model to 1D or 2D data to generate outputs of the same shape. Like CNNs, we use a context window to represent the field of input of the attention model. The context window slides along the sequence, each time moving forward a step of size *stride*. As a special case, if *stride* equals the size of the context window, this model is the same as the chunked attention model mentioned above. If *stride* chooses a value smaller than

the size of the context window, the attention model will become denser. Figure 6.13 (d) shows the case of  $stride = 1$  where the chunk overlapping is maximized. A way to achieve relatively sparser attention is to use a **dilated context window**. Figure 6.13 (e) shows an example of the dilated strided attention model, where the context window is discontinuous, with gaps of size 1.

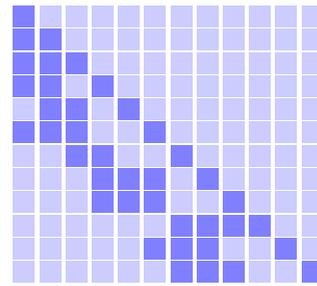
- **Learning Attention Fields.** Because the attention field  $\pi_i$  can be any sub-set of  $\{1, \dots, n\}$ , we can develop more general sparse attention models by considering attention maps beyond chunk-based patterns. The only question is how to determine which positions the model attends to for a given position. One simple approach is to use a computationally cheaper model to estimate the “importance” of each position. Then, attention weights are computed only for some of the positions which are thought to be most important [Zhou et al., 2021]. A second approach is grouping: positions are grouped, and then the attention weights are computed only for positions in the same group. It is often relatively easy to achieve this by running clustering algorithms on keys and queries. For example, we can cluster keys and queries via  $k$ -means clustering. The centroids of the clusters can be treated as additional parameters of the attention model, and so can be learned during optimization [Roy et al., 2021]. One benefit of learning attention fields is that the model can spread its attention broader over the sequence. This is a useful property for many NLP problems because word dependencies are sometimes long-range, not restricted to a local context window. See Figure 6.13 (f) for an example of the attention map learned through this model. Alternative approaches to learning to attend are to use sorting or hashing functions to group similar key and query vectors [Kitaev et al., 2020; Tay et al., 2020a]. These functions can be either heuristically designed functions or neural networks with learnable parameters. By using these functions, we can reorder the sequence so that the inputs in the same group are adjacent in the reordered sequence. In this way, the resulting attention map follows a chunk-wise pattern, and the model is computationally efficient through the use of the chunked attention approach.
- **Hybrid Methods.** Above, we have discussed a range of different sparse attention models. It is natural to explore methods that combine multiple models together to make use of their benefits in some way. A simple way to do this is to combine the attention fields of different models. For example, in Zaheer et al. [2020]’s system, the attention map is generated by considering three different sparse models, including local attention (chunked attention), global attention, and random attention<sup>11</sup>. The resulting model is still a sparse model, but is somewhat more robust as it involves multiple patterns from different perspectives of attention modeling. Another way of combining multiple attention models is to use different models for different heads in multi-head attention [Child et al., 2019; Beltagy et al., 2020]. For example, one can use one head as a local attention model, and use another head as a global attention model (see Figure 6.13 (g-h)).

---

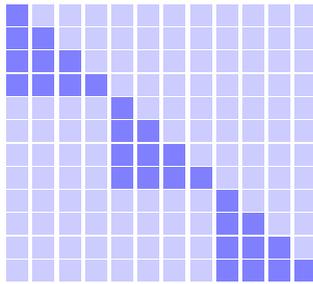
<sup>11</sup>Here the global attention model attends each word only to a special word which accounts for the entire sequence and is often placed at the beginning of the sequence. The random attention model attends each word to a random set of the words of the sequence.



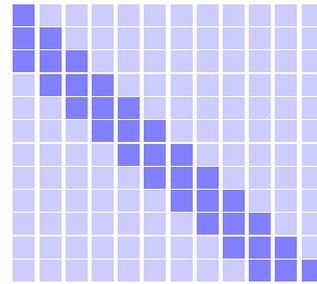
(a) Standard Attention



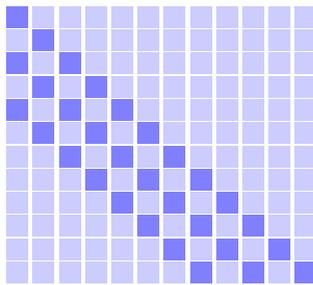
(b) Span-based Attention



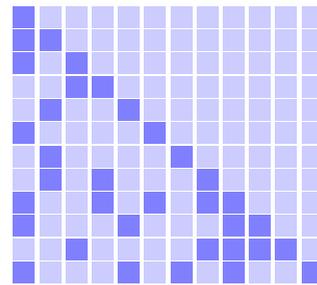
(c) Chunked Attention



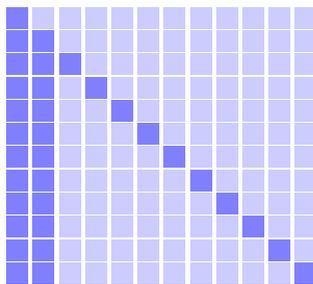
(d) Strided Attention



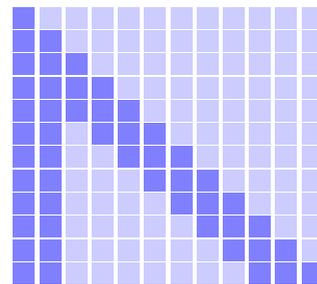
(e) Dilated Strided Attention



(f) Learning Attention Fields



(g) Global Attention



(h) Hybrid Methods

Figure 6.13: Illustration of the attention maps of different models (self-attention on the decoder side). Dark cells mean  $A'(i, j) \neq 0$  (i.e.,  $i$  attends to  $j$ ), and light cells mean  $A'(i, j) = 0$  (i.e.,  $i$  does not attend to  $j$ ). In all these attention maps, we assume that every position attends to itself by default (see diagonals).

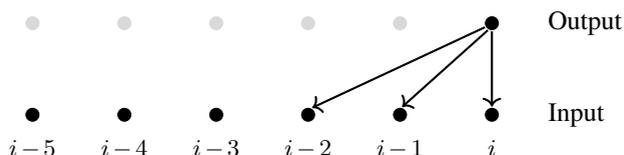
One disadvantage of sparse models compared to dense models is that they are not computationally efficient on GPUs or CPUs. While sparse models can ideally reduce both the memory and computation requirements, the actual rate at which work can be done by sparse models is much slower than by dense models. In practice, it is difficult for sparse models to approach the peak FLOPS of a GPU or CPU<sup>12</sup>. Therefore, they are often used for the purpose of high memory efficiency, not really for the purpose of efficient computation. On the other hand, sparse models are still of great use to NLP practitioners in the context of memory-efficient Transformer, especially when Transformer systems are used to deal with extremely long sequences.

## 6.4.2 Recurrent and Memory Models

For sequence generation problems, Transformer can also be thought of as a memory system. Consider again the general setting, in which we are given the states of previous  $i - 1$  positions, and we wish to predict the next state. In self-attention, this is done by using the query at position  $i$  (i.e.,  $\mathbf{q}_i$ ) to access the key-value pairs of the previous positions (i.e.,  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_{i-1}, \mathbf{v}_{i-1})\}$ ). Then, we move to position  $i + 1$ , and add  $(\mathbf{k}_i, \mathbf{v}_i)$  to the collection of key-value pairs. This procedure can be interpreted in terms of the memory mechanism (see Chapter 4). The Transformer model maintains a memory that retains the information of the past. When moving along the sequence, we repeat the same operation, each time generating some output by reading the memory, and then updating the memory so that new information could be stored in some way. This is illustrated in Figure 6.14.

### 1. Cache-based Memory

The memory here can be viewed as a datastore of vectors. From a machine learning perspective, this is a non-parametric model, and the cost of accessing the model grows as a longer subsequence is observed. Clearly, such a variable-length memory will generally be infeasible if the model deals with a very, very long sequence. For the modeling problem of arbitrary length sequences, it is common to use a fix-length memory instead. As in many NLP problems, one of the simplest ways to do this is to consider a cache saving recent information, that is, we restrict the modeling to a context window. Let  $n_c$  be the size of the context window. The model keeps track of the  $n_c - 1$  latest states to the current position, so that its closest successors can be considered at each step. This means that, for each position, a self-attention sub-layer attends to  $n_c - 1$  positions ahead, like this



If we stack multiple self-attention sub-layers, a larger context window would be considered.

<sup>12</sup>FLOPS = floating point operations per second.

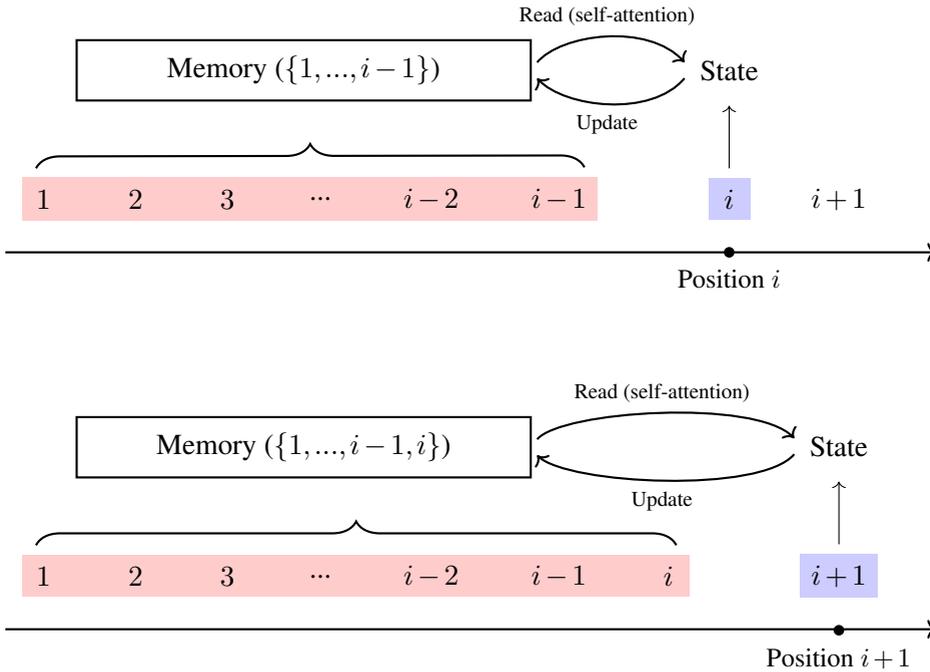
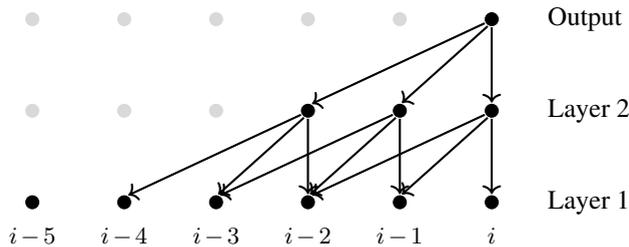


Figure 6.14: Transformer as a memory system. At position  $i$ , the collection of the key-value pairs of positions  $\{1, \dots, i-1\}$  is used as a memory of the past information. The Transformer model accesses this memory to generate some output, and then adds the key-value pair of position  $i$  to the memory. Moving to the next position, we repeat the same procedure of memory access and update.

For example, a model involving two self-attention sub-layers has a context window of size  $2n_c - 1$ , as follows



Therefore, we can take a sufficiently large context by using a multi-layer Transformer model. Note that the context window model here is essentially the same as the strided attention model presented in the preceding section. Systems of this type are often easy to implement: we slide a window along the sequence, and, in each move, we make predictions at the last position of the window (for inference), or back-propagate errors (for training).

An alternative way to train this context window model is by chunked attention. We divide the sequence into chunks (or sub-sequences) which are of the same length  $n_c$ . Then, we treat these chunks as individual training samples, and run the training program on each of

them as usual. This approach, however, completely ignores the relationship between inputs in different chunks. One way to address this issue is to introduce dependence between chunks. For example, the **Transformer-XL** model allows every chunk to access one or more preceding chunks [Dai et al., 2019]. In the simplest case, consider an example in which  $\text{chunk}_k$  can see its successor  $\text{chunk}_{k-1}$ . Each position in  $\text{chunk}_k$  can attend to all its preceding positions in both  $\text{chunk}_k$  and  $\text{chunk}_{k-1}$ .

In Transformer-XL, this approach is implemented in a simplified form. First, each position is constrained to attend to  $n_c - 1$  previous positions so that the size of the attention field of a position is the same in the training and inference stages. Such a method turns the problem back to strided attention, making the implementation of the attention model straightforward. On the other hand, the difference between the standard strided attention model and the Transformer-XL model is that in Transformer-XL, we perform training in a chunk-wise manner. Once we finish the training on a chunk, we directly move to the next chunk, rather than sliding the context window a small step forward. Second, while this approach allows for connections between chunks, the parameters of the sub-network on  $\text{chunk}_{k-1}$  are fixed, and we only update the parameters of the sub-network on  $\text{chunk}_k$  in the  $k$ -th step. See Figure 6.15 for an illustration.

The above model is similar in spirit to recurrent models because all of them require the computation in one step to depend on the states of the preceding steps. However, it is not in the standard form of a recurrent model, in which the output of a recurrent unit in one step is the input in the next step. Instead, the “recurrence” is expressed by involving connections between two different layers, that is, the output of one layer in  $\text{chunk}_{k-1}$  is used as the input of a higher-level layer in  $\text{chunk}_k$ .

## 2. Encoding Long-term Memory

Another idea for representing the states of a sequence is to frame the task as an encoding problem. Instead of storing all the key-value vectors during left-to-right generation, we construct the memory of the entire “history” as a fixed number of encoded key-value vectors. These encoded key-value vectors can be either a small sub-set of  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_{i-1}, \mathbf{v}_{i-1})\}$  or a small set of newly-generated vectors that encodes  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_{i-1}, \mathbf{v}_{i-1})\}$ .

One way to do the encoding is to apply a pooling operation to  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_{i-1}, \mathbf{v}_{i-1})\}$  [Rae et al., 2019a]. For example, by using average pooling, the memory contains only one key-value pair  $(\bar{\mathbf{k}}, \bar{\mathbf{v}})$

$$\bar{\mathbf{k}} = \frac{1}{i-1} \sum_{j=1}^{i-1} \mathbf{k}_j \quad (6.125)$$

$$\bar{\mathbf{v}} = \frac{1}{i-1} \sum_{j=1}^{i-1} \mathbf{v}_j \quad (6.126)$$

This leads to a very efficient model, and we only need to update the vectors  $(\bar{\mathbf{k}}, \bar{\mathbf{v}})$  at a time [Zhang et al., 2018a]. Let  $(\bar{\mathbf{k}}[i], \bar{\mathbf{v}}[i])$  be the state of the memory at position  $i$ . A more general

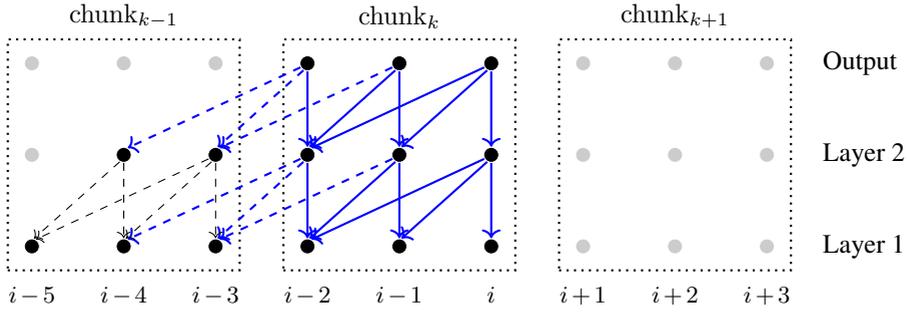
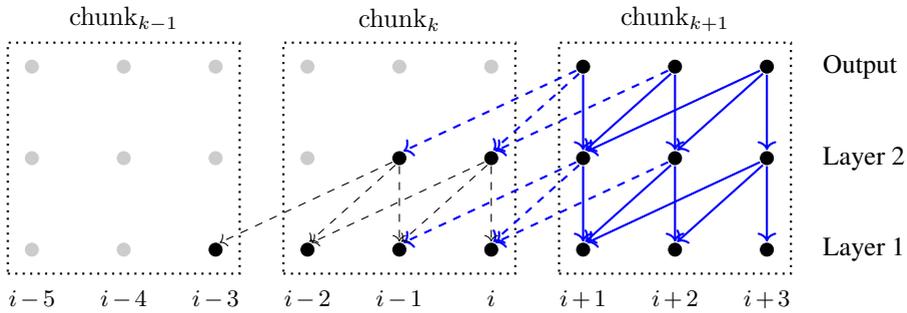
(a) Step  $k$  of chunk-wise training(b) Step  $k + 1$  of chunk-wise training

Figure 6.15: Illustration of chunk-wise training [Dai et al., 2019]. The input sequence is divided into chunks of the same length  $n_c$ . Training is performed on these chunks, each time dealing with a chunk. In chunk $_k$ , the attention field for every position in this chunk is a left context window of size  $n_c$ . Hence this model allows for attention across chunks, for example, position  $i - 2$  in chunk $_k$  can attend to positions  $i - 3$  and  $i - 4$  in chunk $_{k-1}$  (see sub-figure (a)). For training, errors are back-propagated only in the sub-network for chunk $_k$ , leaving other parts of the model unchanged. Here we use dashed lines to denote information flow that we consider in the forward pass but not in the backward pass. Once we finish the training on chunk $_k$ , we move to the next chunk, and repeat the same training procedure.

definition of  $(\bar{\mathbf{k}}[i], \bar{\mathbf{v}}[i])$  is given in a recursive form

$$\bar{\mathbf{k}}[i] = \text{KMem}(\bar{\mathbf{k}}[i-1], \mathbf{k}_{i-1}) \quad (6.127)$$

$$\bar{\mathbf{v}}[i] = \text{VMem}(\bar{\mathbf{v}}[i-1], \mathbf{v}_{i-1}) \quad (6.128)$$

where  $\text{KMem}(\cdot)$  and  $\text{VMem}(\cdot)$  are functions that update the memory by taking both the states of the memory at the previous position (i.e.,  $\bar{\mathbf{k}}[i-1]$  and  $\bar{\mathbf{v}}[i-1]$ ) and the new states (i.e.,  $\mathbf{k}_{i-1}$  and  $\mathbf{v}_{i-1}$ ). There are many forms of the functions like  $\text{KMem}(\cdot)$  and  $\text{VMem}(\cdot)$  in common use. For example, if  $\text{KMem}(\cdot)$  and  $\text{VMem}(\cdot)$  are weighted sum functions, we can derive the same forms as Eqs. (6.125) and (6.126). If  $\text{KMem}(\cdot)$  and  $\text{VMem}(\cdot)$  are recurrent cells in

RNNs or LSTM, we obtain a recurrent model of memory.

Extension of the above model to memories having more than one key-value pair is straightforward. One approach is to use the memory to represent sub-sequences. Let  $\{(\bar{\mathbf{k}}_1, \bar{\mathbf{v}}_1), \dots, (\bar{\mathbf{k}}_\kappa, \bar{\mathbf{v}}_\kappa)\}$  be a memory of size  $\kappa$ . Each  $(\bar{\mathbf{k}}_j, \bar{\mathbf{v}}_j)$  is a snapshot of a chunk of length  $n_c$ . Thus, this memory can encode a sequence with maximum length  $\kappa \cdot n_c$ . Then, we can compute  $(\bar{\mathbf{k}}_j, \bar{\mathbf{v}}_j)$  on the corresponding chunk using Eqs. (6.127) and (6.128). A second approach is to organize  $\{(\bar{\mathbf{k}}_1, \bar{\mathbf{v}}_1), \dots, (\bar{\mathbf{k}}_\kappa, \bar{\mathbf{v}}_\kappa)\}$  into a priority queue. We design some function to assign a score to any given key-value pair. The key-value pair can be inserted into the priority queue through the push operation. Ideally, we wish to develop a scoring function to estimate the value of a key-value pair, for example, we use another neural network to evaluate the key-value pair. In this way, the memory is a collection of the most valuable key-value pairs over the input sequence.

Although representing the memory as a set of vectors is an obvious choice for the model design in Transformer, the memory is discrete and its capacity is determined by the number of the vectors. An alternative form of memory is **continuous memory**. This type of model typically builds on the idea of function approximation, in which  $\{\mathbf{k}_1, \dots, \mathbf{k}_{i-1}\}$  or  $\{\mathbf{v}_1, \dots, \mathbf{v}_{i-1}\}$  is viewed as a series of data points, and a continuous function is developed to fit these data points. Then, we no longer need to store  $\{\mathbf{k}_1, \dots, \mathbf{k}_{i-1}\}$  and  $\{\mathbf{v}_1, \dots, \mathbf{v}_{i-1}\}$ . Instead, the memory is represented by the functions fitting these vectors. A simple method is to combine simple functions to fit complex curves of data points. For example, we can develop a set of basis functions and use a linear combination of them to approximate the key or value vectors [Martins et al., 2022]. The resulting model is parameterized by these basis functions and the corresponding weights in the combination.

It is also straightforward to use a short-term memory and a long-term memory simultaneously so that we can combine the merits of both. For example, we use a cache-based memory to capture local context, and use an efficient long-term memory that encodes the entire history to model long-range dependency. This idea is also similar to that used in combining different sparse attention models as discussed in the previous subsection.

### 3. Retrieval-based Methods

So far in this subsection, we have discussed approaches based on fixed-length models. It is also possible to develop efficient memory models by improving the efficiency of accessing the memories, instead of just reducing the memory capacities. One way to achieve this is to store the past key-value pairs in a database (call it a **vector database**), and to find the most similar ones when querying the database. To be more precise, given a query  $\mathbf{q}$ , we use the database to find a set of top- $p$  relevant key-value pairs (denoted by  $\Omega_p$ ) by performing similarity search based on the dot-product similarity measure between query and key vectors. Then, we attend  $\mathbf{q}$  to  $\Omega_p$  as in standard self-attention models. The idea behind this method is to consider only a small number of elements that contribute most to the attention result. Therefore, the model is essentially a sparse attention model which is computationally efficient. Another advantage of this method is that it allows for fast similarity search over a very large set of vectors because of the highly optimized implementation of vector databases. Building a memory as a retrieval

system can fall under the general framework called the **retrieval-augmented approach**. It provides a simple way to incorporate external memories into neural models like Transformer [Guu et al., 2020; Lewis et al., 2020b; Wu et al., 2021].

### 6.4.3 Low-dimensional Models

In many practical applications, Transformer models are “high-dimensional” models. This is not only because the input and/or output data is in high-dimensional spaces, but also because some of the intermediate representations of the data in the model are high-dimensional. As discussed in Section 6.4.1, this high dimensionality arises in part from the steps of computing the attention matrix as in Eq. (6.119) (for ease of presentation, we repeat the equation here)

$$\text{Att}_{\text{self}}(\mathbf{S}) = \mathbf{AV} \quad (6.129)$$

and the weighted sum of value vectors as in Eq. (6.120)

$$\mathbf{A} = \text{Softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d}} + \mathbf{M}\right) \quad (6.130)$$

which involves large matrix multiplications  $\mathbf{QK}^T$  and  $\mathbf{AV}$  when the length  $n$  and the hidden dimensionality  $d$  have large values.

The  $\mathbf{AV}$  and  $\mathbf{QK}^T$  operations have a time complexity of  $O(n^2 \cdot d)$  and a space complexity of  $O(n^2 + n \cdot d)$ . Several previously described approaches have reduced this complexity by using sparse models. In this subsection, we focus on methods that approximate these operations via dense computation. One simple idea is to transform  $\mathbf{Q}$ ,  $\mathbf{K}$ , and  $\mathbf{V}$  into smaller matrices, and thus to reduce the computational burden of matrix multiplication. Since  $\mathbf{Q}$ ,  $\mathbf{K}$ , and  $\mathbf{V}$  are all in  $\mathbb{R}^{n \times d}$ , we can achieve this by reducing either the  $n$  dimension or the  $d$  dimension, or both.

#### 1. Reducing $n$

Note that the output  $\text{Att}_{\text{self}}(\mathbf{S})$  is required to be an  $n \times d$  matrix, and so we cannot reduce the number of queries. We instead consider reducing the number of keys and values. Suppose  $n'$  is a number less than  $n$ , and  $\mathbf{K}$  and  $\mathbf{V}$  can be transformed into  $n' \times d$  matrices  $\mathbf{K}'$  and  $\mathbf{V}'$  in some way. We can obtain a “smaller” model simply by replacing  $\mathbf{K}$  and  $\mathbf{V}$  with  $\mathbf{K}'$  and  $\mathbf{V}'$ , giving

$$\text{Att}_{\text{self}}(\mathbf{S}) = \mathbf{AV}' \quad (6.131)$$

$$\mathbf{A} = \text{Softmax}\left(\frac{\mathbf{Q}[\mathbf{K}']^T}{\sqrt{d}} + \mathbf{M}\right) \quad (6.132)$$

This model is in the standard form of self-attention, but has lower time and space complexities, that is,  $O(n' \cdot n \cdot d) < O(n^2 \cdot d)$  and  $O(n' \cdot n + n' \cdot d) < O(n^2 + n \cdot d)$ . If  $n' \ll n$ , the resulting model will be linear with  $n$ .

The key problem here is how to obtain  $\mathbf{K}'$  and  $\mathbf{V}'$  in a way that retains much of the

information in  $\mathbf{K}$  and  $\mathbf{V}$ . There are several ways to do so. One simple method is to select the keys and values that are thought to be important. The importance of a key (or value) can be computed in terms of some computationally cheap measure. For example, we can sample a small number of query-key dot-products and estimate the importance of a key by collecting these dot-product results.

The above method is straightforward but still requires sparse operations, such as sampling and collection. As an alternative, we can use dense computation to transform  $\mathbf{K}$  and  $\mathbf{V}$  to  $\mathbf{K}'$  and  $\mathbf{V}'$ . A typical choice is to use CNNs [Liu et al., 2018]. Let  $\text{Conv}(\cdot)$  be a function describing a set of filters that slide along the  $n$  dimension.  $\mathbf{K}'$  is then given by

$$\mathbf{K}' = \text{Conv}(\mathbf{K}, \mathbf{W}_c, \text{size}_r, \text{stride}) \quad (6.133)$$

where  $\mathbf{W}_c$  is the parameter matrix of the filters,  $\text{size}_r$  is the size of the receptive field, and  $\text{stride}$  is the number of units the filters are translated at a time. In general, we can achieve a high compression rate by choosing large values for  $\text{size}_r$  and  $\text{stride}$ . Likewise, we can compute  $\mathbf{V}'$  using another convolutional function. It is worth noting that, if the parameter  $n'$  is fixed for all samples, compression of  $\mathbf{K}$  and  $\mathbf{V}$  along the length dimension is essentially the same as the fixed-length memory model as described in the preceding subsection. The methods presented here are more general and could be applied to variable-length memories.

We might also be tempted to model the attention function by considering the attention matrix  $\mathbf{A}$  as a high-dimensional representation of data and then applying conventional dimensionality reduction methods. For many problems, it is found that  $\mathbf{A}$  (or more precisely  $\mathbf{Q}\mathbf{K}^T$ ) is a low-rank matrix. In this case, we can compress  $\mathbf{A}$  while retaining as much information as possible. There are many ways to do so. For example, we might use a product of smaller matrices as an approximation to  $\mathbf{A}$  via the SVD technique (see Chapter 3). However, this introduces computational overhead in using SVD compared with the standard attention model. A simpler idea to directly transform  $\mathbf{K}$  and  $\mathbf{V}$  into smaller-sized matrices via linear mappings, given by

$$\mathbf{K}' = \mathbf{U}^k \mathbf{K} \quad (6.134)$$

$$\mathbf{V}' = \mathbf{U}^v \mathbf{V} \quad (6.135)$$

where  $\mathbf{U}^k \in \mathbb{R}^{n' \times n}$  and  $\mathbf{U}^v \in \mathbb{R}^{n' \times n}$  are parameter matrices. Clearly, this leads to a model which is equivalent to that described in Eqs. (6.131) and (6.132). While such a method is intuitive and simple, it is proven to obtain a sufficiently small approximation error  $\epsilon$  if  $n'$  is a linear function of  $d/\epsilon^2$  [Wang et al., 2020b].

## 2. Reducing $d$

Another approach to working in a low-dimensional space is to reduce the  $d$  dimension. One of the simplest methods is to project all queries and keys onto a  $d'$ -dimensional space ( $d' < d$ ), and to compute the dot-product of any key-value pair in the new space. For modeling, we only need to replace  $\mathbf{Q} \in \mathbb{R}^{n \times d}$  and  $\mathbf{K} \in \mathbb{R}^{n \times d}$  by new representations  $\mathbf{Q}' \in \mathbb{R}^{n \times d'}$  and  $\mathbf{K}' \in \mathbb{R}^{n \times d'}$ .

We can easily modify Eq. (6.130) to use  $\mathbf{Q}'$  and  $\mathbf{K}'$  in computing the attention matrix

$$\mathbf{A} = \text{Softmax}\left(\frac{\mathbf{Q}'[\mathbf{K}']^T}{\sqrt{d}} + \mathbf{M}\right) \quad (6.136)$$

$\mathbf{Q}'$  and  $\mathbf{K}'$  are given by

$$\mathbf{Q}' = \mathbf{Q}\mathbf{U}^q \quad (6.137)$$

$$\mathbf{K}' = \mathbf{K}\mathbf{U}^k \quad (6.138)$$

where  $\mathbf{U}^q \in \mathbb{R}^{d \times d'}$  and  $\mathbf{U}^k \in \mathbb{R}^{d \times d'}$  are parameter matrices of linear transformations.

It is also possible to exploit **kernel methods** to obtain an efficient dot-product attention model. The basic idea is to map all data points (represented as vectors) from one space to another space, so that the problem, which might be difficult to solve in the original space, is easier to solve in the new space. The “trick” of kernel methods is that we actually do not need to know the mapping function, but only need to know how to compute the inner product of vectors in the new space in one operation<sup>13</sup>. This operation of the inner product is usually called the kernel and denoted by  $K(\cdot, \cdot)$ .

It is interesting to approximate  $\mathbf{A}$  in a fashion analogous to  $K(\cdot, \cdot)$  in kernel methods. To illustrate, note in Eq. (6.130)  $\mathbf{A}$  is a fraction denoting the normalized attention weights. The numerator can be written in the form

$$\tilde{\mathbf{A}} = \text{Mask}\left(\exp\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\right) \quad (6.140)$$

$$(6.141)$$

Here  $\text{Mask}(\cdot)$  is a function which has the same effect as using the additive masking variable  $\mathbf{M}$ . Then,  $\mathbf{A}$  can be expressed as

$$\mathbf{A} = \mathbf{D}^{-1}\tilde{\mathbf{A}} \quad (6.142)$$

where  $\mathbf{D}$  is an  $n \times n$  diagonal matrix. Each entry of the main diagonal is the sum of the entries of the corresponding row in  $\tilde{\mathbf{A}}$ , denoting the normalization factor of Softmax. Substituting this equation into Eq. (6.130), we have

$$\text{Att}_{\text{self}}(\mathbf{S}) = \mathbf{D}^{-1}\tilde{\mathbf{A}}\mathbf{V} \quad (6.143)$$

<sup>13</sup>In mathematical analysis, the inner product is a generalized notion of the dot-product. It is typically denoted by  $\langle \cdot, \cdot \rangle$ . A formal definition of the inner product requires that  $\langle \cdot, \cdot \rangle$  satisfies several properties in a vector space. Although the inner product has different forms in different contexts, in the Euclidean space  $\mathbb{R}^d$ , it is the same thing as the dot-product, that is, given two vectors  $\mathbf{a} \in \mathbb{R}^d$  and  $\mathbf{b} \in \mathbb{R}^d$ , we have

$$\begin{aligned} \langle \mathbf{a}, \mathbf{b} \rangle &= \mathbf{a} \cdot \mathbf{b} \\ &= \sum_{i=1}^d a_i \cdot b_i \end{aligned} \quad (6.139)$$

In this model,  $\tilde{A}(i, j)$  can be viewed as a similarity function over all query-key pairs in a  $d$ -dimensional space. Here we assume that this function, which is in the form of the dot-product of vectors, can be approximated by a kernel function

$$\begin{aligned}\tilde{A}(i, j) &= K(\mathbf{q}_i, \mathbf{k}_j) \\ &= \langle \phi(\mathbf{q}_i), \phi(\mathbf{k}_j) \rangle\end{aligned}$$

$\phi(\cdot)$  is a mapping from  $\mathbb{R}^d$  to  $\mathbb{R}^{d'}$ . We can represent the queries and keys in the following form

$$\begin{aligned}\mathbf{Q}' &= \phi(\mathbf{Q}) \\ &= \begin{bmatrix} \phi(\mathbf{q}_1) \\ \vdots \\ \phi(\mathbf{q}_n) \end{bmatrix}\end{aligned}\tag{6.144}$$

$$\begin{aligned}\mathbf{K}' &= \phi(\mathbf{K}) \\ &= \begin{bmatrix} \phi(\mathbf{k}_1) \\ \vdots \\ \phi(\mathbf{k}_n) \end{bmatrix}\end{aligned}\tag{6.145}$$

Then, we develop a kernelized attention model by approximating the attention weight  $\alpha_{i,j}$  in the form

$$\alpha_{i,j} \approx \frac{\phi(\mathbf{q}_i)\phi(\mathbf{k}_j)^\top}{\sum_{j'=1}^n \phi(\mathbf{q}_i)\phi(\mathbf{k}_{j'})^\top}\tag{6.146}$$

The key idea behind this kernelized attention model is that we can remove the Softmax function if the queries and keys are mapped to a new space. Using this approximation, the  $i$ -th output vector of the attention model (i.e., the  $i$ -th row vector of  $\text{Att}_{\text{self}}(\mathbf{S})$ ) is given by

$$\begin{aligned}\mathbf{c}_i &= \sum_{j=1}^n \alpha_{i,j} \cdot \mathbf{v}_j \\ &\approx \sum_{j=1}^n \left( \frac{\phi(\mathbf{q}_i)\phi(\mathbf{k}_j)^\top}{\sum_{j'=1}^n \phi(\mathbf{q}_i)\phi(\mathbf{k}_{j'})^\top} \cdot \mathbf{v}_j \right) \\ &= \frac{\sum_{j=1}^n \phi(\mathbf{q}_i)\phi(\mathbf{k}_j)^\top \mathbf{v}_j}{\sum_{j'=1}^n \phi(\mathbf{q}_i)\phi(\mathbf{k}_{j'})^\top} \\ &= \frac{\phi(\mathbf{q}_i) \left( \sum_{j=1}^n \phi(\mathbf{k}_j)^\top \mathbf{v}_j \right)}{\phi(\mathbf{q}_i) \left( \sum_{j'=1}^n \phi(\mathbf{k}_{j'})^\top \right)}\end{aligned}\tag{6.147}$$

Although the equation appears a bit complicated, the idea is simple: instead of attending the query to all keys to obtain the attention weight  $\alpha_{i,j}$ , we can compute the sum of the multiplications  $\sum_{j=1}^n \phi(\mathbf{k}_j)^\top \mathbf{v}_j \in \mathbb{R}^{d' \times d}$  and then multiply it with the kernelized query  $\phi(\mathbf{q}_i)$ . Returning to the notation used in Eq. (6.143), we define the  $i$ -th entry of  $\mathbf{D}$  to be  $\phi(\mathbf{q}_i) \sum_{j'=1}^n \phi(\mathbf{k}_{j'})^\top$ .

Then, the attention model can be re-expressed in the form

$$\begin{aligned}
 \text{Att}_{\text{self}}(\mathbf{S}) &= \mathbf{D}^{-1} \phi(\mathbf{Q}) \phi(\mathbf{K})^T \mathbf{V} \\
 &= \mathbf{D}^{-1} \mathbf{Q}' \mathbf{K}'^T \mathbf{V} \\
 &= \mathbf{D}^{-1} (\mathbf{Q}' (\mathbf{K}'^T \mathbf{V}))
 \end{aligned} \tag{6.148}$$

Here we change the order of computation from left-to-right to right-to-left using parentheses. Given that  $\mathbf{Q}' \in \mathbb{R}^{n \times d'}$  and  $\mathbf{K}' \in \mathbb{R}^{n \times d'}$ , this model has time and space complexities of  $O(n \cdot d \cdot d')$  and  $O(n \cdot d + n \cdot d' + d \cdot d')$ , respectively. Therefore, the model is linear with respect to the sequence length  $n$ , and is sometimes called the **linear attention model**. One computational advantage of this model is that we need only compute the multiplication  $\mathbf{K}'^T \mathbf{V}$  (i.e.,  $\sum_{j=1}^n \phi(\mathbf{k}_j)^T \mathbf{v}_j$ ) and the corresponding normalization factor (i.e.,  $\sum_{j'=1}^n \phi(\mathbf{k}_{j'})^T$ ) once. The results can then be used for any query [Katharopoulos et al., 2020]. The memory needs to maintain  $\sum_{j=1}^n \phi(\mathbf{k}_j)^T \mathbf{v}_j$  and  $\sum_{j'=1}^n \phi(\mathbf{k}_{j'})^T$  and update them when new key and value vectors come.

Still, there are several problems regarding this kernelized model, for example, how to develop the feature map  $\phi(\cdot)$  to obtain a good approximation to the standard attention model. Interested readers may refer to Choromanski et al. [2020]’s work for more details.

A second idea for reducing  $d$  is to take sub-space models, in which a problem in a  $d$ -dimensional space is transformed into sub-problems in lower-dimensional spaces, and the solution to the original problem is approximated by some combination of the solutions to these sub-problems. In a general sub-space model, a  $d$ -dimensional key vector  $\mathbf{k}$  can be mapped into a set of  $d'$ -dimensional vectors  $\{\mathbf{K}'_1, \dots, \mathbf{K}'_\eta\}$ . To simplify modeling, we can do this by vector segmentation, that is, we segment  $\mathbf{k}$  into  $\eta$  sub-vectors, each having  $d' = \frac{d}{\eta}$  dimensions. We can transform all query and value vectors in the same way. Then, the attention model is applied in each of these sub-spaces.

This method, however, does not reduce the total amount of computation. As presented in Lample et al. [2019]’s work, we can instead approximate the dot-product attention over a set of key-value pairs by considering top- $p$  candidates in each sub-space. More precisely, we find  $p$ -best key-value pairs in each sub-space, which is computationally cheaper. The Cartesian product of these  $p$ -best key sets consists of  $p^\eta$  product keys. Likewise, we obtain  $p^\eta$  product values. The remaining work is simple: the  $d$ -dimensional queries attend to these  $d$ -dimensional product keys and values. An interesting difference between this sub-space model and the  $d$ -dimensional space model is that the generated product keys and values may be different from any of the original key-values  $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_{i-1}, \mathbf{v}_{i-1})\}$ . This provides a way for learning new representations of the past information.

So far we have discussed approaches to dimensionality reduction along either the  $n$  or  $d$  dimension. It is straightforward to combine them to develop a “lower-dimensional” model. As an example, suppose that we have the  $n \rightarrow n'$  reduction for keys and values, and the  $d \rightarrow d'$

reduction for queries and keys. The model takes the form

$$\begin{aligned} \text{Att}_{\text{self}}(\mathbf{S}) &= \mathbf{A}\mathbf{V}' \\ \mathbf{A} &= \text{Softmax}\left(\frac{\mathbf{Q}'\mathbf{K}'^T}{\sqrt{d'}} + \mathbf{M}\right) \end{aligned} \quad (6.149)$$

where  $\mathbf{Q}' \in \mathbb{R}^{n \times d'}$ ,  $\mathbf{K}' \in \mathbb{R}^{n' \times d'}$ , and  $\mathbf{V}' \in \mathbb{R}^{n' \times d'}$  are low-dimensional representations for queries, keys and values. As usual, we can easily obtain these representations through the linear mappings of  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$ . The time and space complexities of this model are  $O(n' \cdot n \cdot d')$  and  $O(n' \cdot n + n' \cdot d')$ .

#### 6.4.4 Parameter and Activation Sharing

Redundancy is common to most large-scale neural networks. As a result, many of these models are over-parameterized, making the training and inference less efficient. One common approach to redundancy reduction is to simplify the modeling by removing useless components of the models, for example, we can either prune a complex model or share sub-models among different components of it to obtain a reasonably small model. In this subsection, we discuss methods of parameter and intermediate state sharing in Transformer models. We leave the discussion of model transfer and pruning to Section 6.4.7.

Shared-parameter architectures are widely used in neural network-based systems. Well-known examples include CNNs and RNNs, where the same set of parameters (or layers) is applied across different regions of the input. This produces a “big” neural network, parts of which have the same architecture and the same shared parameters. For Transformers as well as other sequence models, the sharing mechanism can be applied to different levels of modeling. A simple example, which might be not related to architecture design, is shared embedding. In machine translation, a typical strategy for dealing with words in two languages is to develop two separate embedding models. Alternatively, one can use a single embedding model for both languages. The parameters of the model are then learned during the training of both the source-side and target-side networks. Such a strategy is also often adopted in multi-lingual sequence models, such as language models that are able to deal with texts in many different languages.

For multi-layer neural networks, a popular method is layer-wise sharing. Suppose there is a stack of layers, all of which have the same form

$$\mathbf{S}^l = \text{Layer}(\mathbf{S}^{l-1}; \theta^l) \quad (6.150)$$

We can tie the parameters for some or all of these layers. For example, given a set of layers  $\{l_1, l_2, \dots, l_n\}$ , we enforce the constraint  $\theta^{l_1} = \theta^{l_2} = \dots = \theta^{l_n}$ , so that we can obtain a smaller model and the optimization of the model can be easier. In practice, this shared-layer model is highly advantageous if many layers are involved, because we can repeat the same process many times to construct a very deep neural network [Dehghani et al., 2018]. For example, sharing a single FFN sub-layer across the Transformer encoder is found to be effective in reducing the redundancy in machine translation systems [Pires et al., 2023].

For Transformers, sharing can also be performed in multi-head attention. An example of this is **multi-query attention** [Shazeer, 2019]. Recall from Section 6.1.3 that the output of a head  $h$  in standard multi-head self-attention can be written as

$$\begin{aligned} \mathbf{C}_h^{\text{head}} &= \text{Att}_{\text{qkv}}(\mathbf{S}_h^q, \mathbf{S}_h^k, \mathbf{S}_h^v) \\ &= \text{Att}_{\text{qkv}}(\mathbf{S}\mathbf{W}_h^q, \mathbf{S}\mathbf{W}_h^k, \mathbf{S}\mathbf{W}_h^v) \end{aligned} \quad (6.151)$$

Here  $\mathbf{S}_h^q = \mathbf{S}\mathbf{W}_h^q$ ,  $\mathbf{S}_h^k = \mathbf{S}\mathbf{W}_h^k$ , and  $\mathbf{S}_h^v = \mathbf{S}\mathbf{W}_h^v$  are the query, key, and value, which are obtained by linearly transforming the input  $\mathbf{S}$  with distinct parameter matrices  $\mathbf{W}_h^q$ ,  $\mathbf{W}_h^k$ , and  $\mathbf{W}_h^v$ . In multi-query attention, we share the same key and value across all the heads, but use different queries for different heads. The form of this model is given by

$$\mathbf{C}_h^{\text{head}} = \text{Att}_{\text{qkv}}(\mathbf{S}\mathbf{W}_h^q, \mathbf{S}\mathbf{W}_0^k, \mathbf{S}\mathbf{W}_0^v) \quad (6.152)$$

Here the key  $\mathbf{S}\mathbf{W}_0^k$  and value  $\mathbf{S}\mathbf{W}_0^v$  are irrelevant to  $h$ . Hence we need only compute them once rather than computing them several times. As a result, we can make a significant saving in computational cost, especially if the number of heads is large. Multi-query attention has been successfully incorporated into recent large language models, such as Llama 2 [Touvron et al., 2023b] and Falcon<sup>14</sup>.

By extending the idea of sharing to more general situations, any intermediate states can be shared across a neural network. For example, reusing neuron activations allows a sub-model to be applied multiple times. For Transformers, sharing can be considered inside the process of self-attention. It is found that the attention maps of different layers are similar in some NLP tasks [Xiao et al., 2019]. Therefore, it is reasonable to compute the attention map only once and then use it in the following layers.

If we make a further generalization of the sharing mechanism, we can view it as a process by which we use the result produced previously rather than computing it on the fly. It is thus possible to reuse the information across different runs of a neural network. A related example is **reversible residual networks**, in which activations of one layer can be recovered from the activations of the following layer [Gomez et al., 2017]. Hence we only keep the output of the latest layer in the forward pass. Then, in the backward pass of training, we reconstruct the output of each layer from its successor. One advantage of this reversible treatment is that the information produced in the forward pass is shared implicitly, and the model is memory-efficient [Kitaev et al., 2020].

### 6.4.5 Alternatives to Self-Attention

We have seen that the use of self-attention is a primary source of the large computation and memory requirements for Transformer systems. It is natural to wonder if there are efficient alternatives to self-attention models. Here we present briefly some of the Transformer variants in which self-attention sub-layers are not required and we instead replace them with other types of neural networks.

<sup>14</sup><https://falconllm.tii.ae/index.html>

### 1. CNN as A Replacement of Self-Attention

CNNs are simple and widely used neural networks, and are considered as potential alternatives to self-attention models. To apply CNNs to Transformers, all we need is to construct a convolutional sub-layer to replace the self-attention sub-layer in a Transformer block. While a filter of CNNs has a restricted receptive field and thus takes inputs from a “local” context window, large contexts can be easily modeled by stacking multiple convolutional sub-layers. One key advantage of CNNs is that the number of elementary operations required to run CNNs is a linear function of the sequence length  $n$ , compared with the quadratic function for self-attention networks. In practical systems, there have been many highly-optimized implementations for CNNs, making it easier to apply them to sequence modeling. For further improvements to memory efficiency, we can use lightweight CNN variants, for example, depth-wise CNNs [Wu et al., 2018a] <sup>15</sup>.

### 2. Linear Attention

As with many practical approaches to sequence modeling, there is also considerable interest in developing linear models in order to speed up the processing of long sequences. While there are many ways to define a linear model, one general form that is commonly used in sequence models is

$$\mathbf{z}_i = f(a \cdot \mathbf{z}_{i-1} + b \cdot \mathbf{s}_i) \quad (6.154)$$

Here  $\mathbf{s}_i$  represents some intermediate states of the model at step  $i$ , and  $\mathbf{z}_i$  represents the summary of the history states up to step  $i$ . It is easy to see that this is a recurrent model: the output at step  $i$  depends only on the input at the current step and the output at the previous step. As with the popular design choices in neural network-based systems, the linear part is followed by a transformation  $f(\cdot)$  which can be either an activation function or a feedforward neural network. Note that, Eq. (6.154) defines a standard linear model only if  $f(\cdot)$  is a linear function. The use of  $f(\cdot)$  gives greater flexibility in modeling the problem, although the term *linear model* may not be applied if  $f(\cdot)$  chooses a non-linear form.

The above formula describes a linearly structured model which can be seen as an instance of a general family of mathematical models. Typically, it can be represented as a chain structure,

---

<sup>15</sup>Recall from Chapter 2 that in CNNs a filter (or a set of filters) combines the input variables in the receptive field into an output variable (or a set of output variables) via linear mapping. Suppose that the input and output of a problem are represented as sequences of feature vectors. Given a filter having a  $d \times k$  receptive field, we slide it along the sequence. At each step, the filter takes  $d \times k$  input features and produces an output feature. This procedure is typically expressed by

$$y = \text{ReduceSum}(\mathbf{x} \odot \mathbf{W}) \quad (6.153)$$

where  $\mathbf{x} \in \mathbb{R}^{k \times d}$  is the vector representation of the input,  $y \in \mathbb{R}$  is the output feature, and  $\mathbf{W} \in \mathbb{R}^{k \times d}$  is the weight matrix. The function  $\text{ReduceSum}(\cdot)$  computes the sum of all element-wise products between  $\mathbf{x}$  and  $\mathbf{W}$ . If we want the input and output to have the same number of features, we can design  $d$  filters and the number of parameters will be  $d^2 \cdot k$ .

In depth-wise CNNs, we tie the weights across different feature dimensions. More precisely, all the column vectors of  $\mathbf{W}$  are the same. Thus, the number of the unique parameters of the model is reduced to  $d \cdot k$  (each  $\mathbf{W}$  corresponding to a filter having  $k$  unique parameters).

or an ordered set of nodes. The model repeats the same computation process from the first node to the last, each time taking the information from the current and previous steps and producing an output vector that is used in the following time steps. As a result, the space and time cost of the model scales linearly with the length of the chain.

We can extend Eq. (6.154) to a standard RNN model by simply making a linear transformation of the current input and the previous state, that is,  $\mathbf{z}_i = f(\mathbf{z}_{i-1} \cdot \mathbf{W}_z + \mathbf{s}_i \cdot \mathbf{W}_s)$ . It is thus straightforward to apply RNN and its variants to Transformer to obtain a hybrid model. For example, we can use LSTM and GRUs in building some of the Transformer layers to combine the merits of both recurrent models and self-attentive models [Chen et al., 2018b]. As the conventional recurrent models have been discussed at length in Chapter 2, we skip the discussion of them here.

In fact, we may be more interested in developing linear attention models, so that we can obtain an efficient system, while still retaining the benefit of globally attentive sequence modeling. Part of the difficulty in doing this is that the form of self-attention is not linear. Let us take a moment to see how this difficulty arises. Recall that the result of self-attention can be written in the following form

$$\begin{aligned} \text{Att}_{\text{self}} &= \mathbf{A} \cdot \mathbf{V} \\ &= \psi(\mathbf{Q} \cdot \mathbf{K}^T) \cdot \mathbf{V} \end{aligned} \quad (6.155)$$

Here  $\psi(\cdot)$  is a function that is composed by taking the scaling, exponentiating, masking and normalization operations (i.e.,  $\psi(\mathbf{a}) = \text{Normalize}(\text{Mask}(\exp(\frac{\mathbf{a}}{\sqrt{d}})))$ ). Because  $\psi(\cdot)$  is a complex non-linear function, there is no obvious equivalent that simplifies the computation, and we have to calculate the two matrix multiplications separately (one inside  $\psi(\cdot)$  and one outside  $\psi(\cdot)$ ). As a consequence, we need to store all the key-value pairs explicitly, and visit each of them given a query. Not surprisingly, this leads to a model whose computational cost grows quadratically with the sequence length  $n$ .

Although in self-attention keys and values are coupled, they are used in separate steps. An elegant form of this model might be that allows for a direct interaction between the keys and queries, so that we can encode the context information in a way that is irrelevant to the queries. A trick here is that we can remove the non-linearity from  $\psi(\cdot)$  by using a feature space mapping  $\phi(\cdot)$  on the queries and keys, and reformulate  $\psi(\mathbf{Q} \cdot \mathbf{K}^T)$  (i.e.,  $\mathbf{A}$ ) in a form of matrix products. For example, recall from Section 6.4.3 that we can transform  $\mathbf{Q}$  and  $\mathbf{K}$  to  $\mathbf{Q}' = \phi(\mathbf{Q}) \in \mathbb{R}^{n \times d'}$  and  $\mathbf{K}' = \phi(\mathbf{K}) \in \mathbb{R}^{n \times d'}$  through the mapping  $\phi(\cdot)$ . Then, we define the form of the attention model to be

$$\begin{aligned} \text{Att}_{\text{self}} &\equiv \psi'(\mathbf{Q}' \cdot \mathbf{K}'^T) \cdot \mathbf{V} \\ &= \frac{\mathbf{Q}' \cdot \mathbf{K}'^T}{\mathbf{D}} \cdot \mathbf{V} \\ &= \frac{\mathbf{Q}' \cdot (\mathbf{K}'^T \cdot \mathbf{V})}{\mathbf{D}} \end{aligned} \quad (6.156)$$

where  $\psi'(\mathbf{a}) = \frac{\mathbf{a}}{\mathbf{D}}$ . From this definition, we see that, in the case of transformed queries and keys,

the query-key product needs not be normalized via Softmax, but needs only be normalized via a simple factor  $\mathbf{D}$ . Hence the model has a very simple form involving only matrix multiplication and division, allowing us to change the order of the operations using the associativity of matrix multiplication.

This leads to an interesting procedure: keys and values are first encoded via  $\mathbf{K}'^T \cdot \mathbf{V}$ , and then each query attends to this encoding result. Given that  $\mathbf{K}'^T \cdot \mathbf{V} = \sum_{j=1}^n \mathbf{k}'_j{}^T \cdot \mathbf{v}_j$ , we can write  $\mathbf{K}'^T \cdot \mathbf{V}$  in the form of Eq. (6.154), as follows

$$\mu_j = \mu_{j-1} + \mathbf{k}'_j{}^T \cdot \mathbf{v}_j \quad (6.157)$$

Here  $\mu_j \in \mathbb{R}^{d' \times d}$  is a variable that adds  $\mathbf{k}'_j{}^T \cdot \mathbf{v}_j$  at a time. Likewise, we can define another variable  $\nu_j \in \mathbb{R}^d$

$$\nu_j = \nu_{j-1} + \mathbf{k}'_j{}^T \quad (6.158)$$

Then, the output of self-attention for the  $j$ -th query can be written as (see also Eq. (6.147))

$$\text{Att}_{\text{self},j} = \frac{\mathbf{q}'_j \cdot \mu_n}{\mathbf{q}'_j \cdot \nu_n} \quad (6.159)$$

Clearly, this is a linear model, because  $\mu_n$  and  $\nu_n$  are linear with respect to  $n$ . In simple implementations of this model, only  $\mu_j$  and  $\nu_j$  are kept. Each time a new query is encountered, we update  $\mu_j$  and  $\nu_j$  using Eqs. (6.157) and (6.158), and then compute  $\text{Att}_{\text{self},j} = \frac{\mathbf{q}'_j \cdot \mu_j}{\mathbf{q}'_j \cdot \nu_j}$ <sup>16</sup>.

One straightforward extension to the linear attention model is to allow Eqs. (6.157) and (6.158) to combine different terms with different weights. For example, we can redefine  $\mu_j$  and  $\nu_j$  as

$$\mu_j = a \cdot \mu_{j-1} + (1-a) \cdot \mathbf{k}'_j{}^T \cdot \mathbf{v}_j \quad (6.160)$$

$$\nu_j = a \cdot \nu_{j-1} + (1-a) \cdot \mathbf{k}'_j{}^T \quad (6.161)$$

and train the parameter  $a$  as usual. Also, we can treat  $a$  as a gate and use another neural network to compute  $a$  [Peng et al., 2021]. Another model design is to add more terms to Eqs. (6.157) and (6.158) in order to give a more powerful treatment of the linear attention approach [Bello, 2020; Schlag et al., 2021].

We have seen a general idea of designing linear models for the attention mechanism. The key design choice of such models is to remove the Softmax-based normalization, thereby taking linear forms of representations based on various intermediate states of the models. This motivates several recently developed alternatives to self-attention in which efficient inference systems are developed on the basis of recurrent models of sequence modeling [Peng et al., 2023; Sun et al., 2023]. While these systems have different architectures, the underlying models have a similar form, as described in Eq. (6.154). Note that, by using the general formulation of

<sup>16</sup>In autoregressive generation, we generate a sequence from left to right. In this case, we need not consider the keys and values for positions  $> j$ .

recurrent models, we need not restrict the modeling to the standard QKV attention. Instead we may give new meanings and forms to the queries, keys, and values.

The discussion here is also related to the memory models discussed in Section 6.4.2. From the memory viewpoint, the keys and values can be treated as encodings of the context. Therefore, in the linear attention model above we have a memory system in which two simple variables  $\mu_j$  and  $\nu_j$  are used to represent all the context information up to position  $j$ . This results in a fixed-length memory which is very useful in practice. There are also other linear approaches to encoding long sequences. For example, we can view the **moving average** model as an instance of Eq. (6.154), and average a series of state vectors of a Transformer system, either weighted or unweighted.

### 3. State-Space Models

In control systems, **state-space models (SSMs)** are representations of a system whose input and output are related by some **state variables** (or states for short), and whose dynamics is described by first-order differential equations of these states. As a simple example, we consider a continuous time-invariant linear system which is given in the form of the state-space representation

$$\frac{dz(t)}{dt} = \mathbf{z}(t) \cdot \mathbf{A} + \mathbf{s}(t) \cdot \mathbf{B} \quad (6.162)$$

$$\mathbf{o}(t) = \mathbf{z}(t) \cdot \mathbf{C} + \mathbf{s}(t) \cdot \mathbf{D} \quad (6.163)$$

Here  $\mathbf{s}(t)$ ,  $\mathbf{o}(t)$ , and  $\mathbf{z}(t)$  are the values of the input variable, output variable and state variable at time  $t$ <sup>17</sup>. In a general setting,  $\mathbf{s}(t)$ ,  $\mathbf{o}(t)$ , and  $\mathbf{z}(t)$  may have different numbers of dimensions. To simplify the discussion here, we assume that  $\mathbf{s}(t), \mathbf{o}(t) \in \mathbb{R}^d$  and  $\mathbf{z}(t) \in \mathbb{R}^{d_z}$ <sup>18</sup>. Eq. (6.162) is called the **state equation**, where  $\mathbf{A} \in \mathbb{R}^{d_z \times d_z}$  is the state matrix and  $\mathbf{B} \in \mathbb{R}^{d \times d_z}$  is the input matrix. Eq. (6.163) is called the **output equation**, where  $\mathbf{C} \in \mathbb{R}^{d \times d_z}$  is the output matrix and  $\mathbf{D} \in \mathbb{R}^{d \times d}$  is the feedforward matrix.

These equations describe a continuous mapping from the variable  $\mathbf{s}(t)$  to the variable  $\mathbf{o}(t)$  over time. They are, therefore, often used to deal with continuous time series data. To apply this model to the sequence modeling problem discussed in this chapter, we need to modify the above equations to give a discrete form of the state-space representation. Suppose that  $\{\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_n\}$  is a sequence of input data points sampled from  $\mathbf{s}(t)$  with time step  $\Delta t$ . Similarly, we define  $\{\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_n\}$  and  $\{\mathbf{o}_0, \mathbf{o}_1, \dots, \mathbf{o}_n\}$  as sequences of the state and output vectors. Given this notation, we now have a discretized version of the SSM, written as

$$\mathbf{z}_t = \mathbf{z}_{t-1} \cdot \overline{\mathbf{A}} + \mathbf{s}_t \cdot \overline{\mathbf{B}} \quad (6.164)$$

$$\mathbf{o}_t = \mathbf{z}_t \cdot \overline{\mathbf{C}} + \mathbf{s}_t \cdot \overline{\mathbf{D}} \quad (6.165)$$

<sup>17</sup>We use boldface letters to emphasize that the variables are vectors.

<sup>18</sup>In a general state-space model, all these variables are represented as vectors of complex numbers. Because the models defined on the field of complex numbers is applicable to case of real number-based state-spaces, we restrict our discussion to variables in the multi-dimensional real number field.

This formulation of the SSM defines an RNN with a residual connection. To be more precise, Eq. (6.164) describes a recurrent unit that reads the input at step  $t$  and the state at step  $t - 1$ , without using any activation function. Eq. (6.165) describes an output layer that sums both the linear transformations of the state  $\mathbf{z}_t$  and the identity mapping  $s_t$ .

The parameters  $\bar{\mathbf{A}}$ ,  $\bar{\mathbf{B}}$ ,  $\bar{\mathbf{C}}$ , and  $\bar{\mathbf{D}}$  can be induced from  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$  and  $\mathbf{D}$  in several different ways, depending on how Eq. (6.162) is approximated by Eq. (6.164)<sup>19</sup>. One approach to time discretization, called **bilinear transform** or **Tustin's method**, gives a model in which the parameters take the form

$$\bar{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta t}{2} \cdot \mathbf{A}\right) \cdot \left(\mathbf{I} + \frac{\Delta t}{2} \cdot \mathbf{A}\right)^{-1} \quad (6.172)$$

$$\bar{\mathbf{B}} = \Delta t \cdot \mathbf{B} \cdot \left(\mathbf{I} + \frac{\Delta t}{2} \cdot \mathbf{A}\right)^{-1} \quad (6.173)$$

$$\bar{\mathbf{C}} = \mathbf{C} \quad (6.174)$$

$$\bar{\mathbf{D}} = \mathbf{D} \quad (6.175)$$

An alternative approach is to use the Zero-Order-Hold (ZOH) discretization which has the form

$$\bar{\mathbf{A}} = \exp(\Delta t \cdot \mathbf{A}) \quad (6.176)$$

$$\bar{\mathbf{B}} = \Delta t \cdot \mathbf{B} \cdot (\exp(\Delta t \cdot \mathbf{A}) - \mathbf{I}) \cdot (\Delta t \cdot \mathbf{A})^{-1} \quad (6.177)$$

$$\bar{\mathbf{C}} = \mathbf{C} \quad (6.178)$$

$$\bar{\mathbf{D}} = \mathbf{D} \quad (6.179)$$

A detailed discussion of these approaches lies beyond the scope of this book, and we refer the interested reader to standard textbooks on control theory for further details [[Åström and](#)

<sup>19</sup>The discretization process can be interpreted as a numerical method of solving the differential equation. Note that Eq. (6.162) is an ODE

$$\frac{dz(t)}{dt} = g(z(t), t) \quad (6.166)$$

where

$$g(z(t), t) = z(t) \cdot \mathbf{A} + s(t) \cdot \mathbf{B} \quad (6.167)$$

There are many numerical approximations to the solutions to the ODE. For example, the Euler method of solving the ODE can be expressed in the form (see in Section 6.3.3)

$$\mathbf{z}_t = \mathbf{z}_{t-1} + \Delta t \cdot g(\mathbf{z}_{t-1}, t) \quad (6.168)$$

Substituting Eq. (6.167) into Eq. (6.168) yields

$$\begin{aligned} \mathbf{z}_t &= \mathbf{z}_{t-1} + \Delta t(\mathbf{z}_{t-1} \cdot \mathbf{A} + s_t \cdot \mathbf{B}) \\ &= \mathbf{z}_{t-1} \cdot (\mathbf{I} + \Delta t \cdot \mathbf{A}) + s_t \cdot (\Delta t \cdot \mathbf{B}) \end{aligned} \quad (6.169)$$

This gives one of the simplest forms of the discretized state equations [[Gu et al., 2022b](#)], that is,

$$\bar{\mathbf{A}} = \mathbf{I} + \Delta t \cdot \mathbf{A} \quad (6.170)$$

$$\bar{\mathbf{B}} = \Delta t \cdot \mathbf{B} \quad (6.171)$$

Wittenmark, 2013].

The recurrent form of Eq. (6.164) makes it easy to compute the states and outputs over a sequence of discrete time steps. We can unroll  $\mathbf{z}_t$  and  $\mathbf{o}_t$  in a feedforward fashion

$$\begin{array}{l} \mathbf{z}_0 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \\ \mathbf{z}_1 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}} + \mathbf{s}_1 \cdot \bar{\mathbf{B}} \\ \mathbf{z}_2 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^2 + \mathbf{s}_1 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}} + \mathbf{s}_2 \cdot \bar{\mathbf{B}} \\ \dots \end{array} \quad \left| \begin{array}{l} \mathbf{o}_0 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{C}} + \mathbf{s}_0 \cdot \bar{\mathbf{D}} \\ \mathbf{o}_1 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}} \cdot \bar{\mathbf{C}} + \mathbf{s}_1 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{C}} + \mathbf{s}_1 \cdot \bar{\mathbf{D}} \\ \mathbf{o}_2 = \mathbf{s}_0 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^2 \cdot \bar{\mathbf{C}} + \mathbf{s}_1 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}} \cdot \bar{\mathbf{C}} + \\ \quad \mathbf{s}_2 \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{C}} + \mathbf{s}_2 \cdot \bar{\mathbf{D}} \\ \dots \end{array} \right.$$

It is easy to write

$$\mathbf{z}_t = \sum_{i=0}^t \mathbf{s}_i \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{t-i} \quad (6.180)$$

$$\mathbf{o}_t = \sum_{i=0}^t \mathbf{s}_i \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{t-i} \cdot \bar{\mathbf{C}} + \mathbf{s}_t \cdot \bar{\mathbf{D}} \quad (6.181)$$

Clearly, the right-hand side of Eq. (6.181) can be interpreted as a merged output of a convolutional layer and a linear layer. Given that

$$\begin{aligned} \sum_{i=0}^t \mathbf{s}_i \cdot \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{t-i} \cdot \bar{\mathbf{C}} &= \begin{bmatrix} \mathbf{s}_0 & \mathbf{s}_1 & \dots & \mathbf{s}_t \end{bmatrix} \cdot \\ &\quad \begin{bmatrix} \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^t \cdot \bar{\mathbf{C}} & \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{t-1} \cdot \bar{\mathbf{C}} & \dots & \bar{\mathbf{B}} \cdot \bar{\mathbf{C}} \end{bmatrix} \end{aligned} \quad (6.182)$$

we define a filter having the parameters

$$\mathbf{W}_{\text{ssm}} = \begin{bmatrix} \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{n_{\text{max}}} \cdot \bar{\mathbf{C}} & \bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^{n_{\text{max}}-1} \cdot \bar{\mathbf{C}} & \dots & \bar{\mathbf{B}} \cdot \bar{\mathbf{C}} \end{bmatrix} \quad (6.183)$$

where  $n_{\text{max}}$  is the maximum length of the sequence<sup>20</sup>. Then, the output of the state-space

model for a sequence  $\mathbf{S} = \begin{bmatrix} \mathbf{s}_0 \\ \vdots \\ \mathbf{s}_n \end{bmatrix}$  can be expressed as

$$\mathbf{O} = \text{Conv}(\mathbf{S}, \mathbf{W}_{\text{ssm}}) + \text{Linear}(\mathbf{S}, \bar{\mathbf{D}}) \quad (6.184)$$

where  $\text{Conv}(\cdot)$  is the convolution operation, and  $\text{Linear}(\cdot)$  is the linear transformation operation. Such a treatment of the state-space model enables the system to be efficiently implemented using fast parallel convolution algorithms.

Unfortunately, the above model performs poorly in many cases. As with many deep neural networks, careful initialization of the model parameters plays an important role in such models.

<sup>20</sup>Here  $\mathbf{W}_{\text{ssm}}$  can be represented as an  $n_{\text{max}} \times d \times d$  tensor.

For example, restricting the state matrix to particular types of matrices is found to be useful for learning and generalizing on long sequences [Gu et al., 2022a].

Another problem with the basic state-space model is that it involves multiplication of multiple matrices. If the sequence is long (i.e.,  $n$  is a large number), computing  $\bar{\mathbf{A}}^n$  will be computationally expensive and numerically unstable. One of the most popular approaches to developing practical state-space models for sequence modeling is **diagonalization**. The basic idea is that we can transform a state-space model into a new state-space where  $\mathbf{A}$  (or  $\bar{\mathbf{A}}$ ) is diagonalized. Given a state-space model parameterized by  $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ , we can define a new state-space model  $(\mathbf{U}\mathbf{A}\mathbf{U}^{-1}, \mathbf{B}\mathbf{U}^{-1}, \mathbf{U}\mathbf{C}, \mathbf{D})$  by introducing an invertible matrix  $\mathbf{U}$ . It is easy to prove that the two models are equivalent under the state-space transformation  $\mathbf{U}^{21}$ . By using this state-space transformation, and by noting that  $\mathbf{A}$  (or  $\bar{\mathbf{A}}$ ) can be written as a canonical form  $\mathbf{P}^{-1}\Lambda\mathbf{P}^{22}$ , we can enforce the constraint that  $\mathbf{A}$  (or  $\bar{\mathbf{A}}$ ) is a diagonal matrix, giving rise to **diagonal state-space models**. To illustrate, consider the filter used in the convolutional representation of the state-space model (see Eq. (6.182)). Assuming that  $\bar{\mathbf{A}} = \mathbf{P}^{-1}\Lambda\mathbf{P}$ , we can write  $\bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^t \cdot \bar{\mathbf{C}}$  as

$$\begin{aligned}\bar{\mathbf{B}} \cdot \bar{\mathbf{A}}^t \cdot \bar{\mathbf{C}} &= \bar{\mathbf{B}} \cdot (\mathbf{P}^{-1}\Lambda\mathbf{P})^t \cdot \bar{\mathbf{C}} \\ &= \bar{\mathbf{B}} \cdot (\mathbf{P}^{-1}\Lambda\mathbf{P}) \cdot (\mathbf{P}^{-1}\Lambda\mathbf{P}) \cdots (\mathbf{P}^{-1}\Lambda\mathbf{P}) \cdot \bar{\mathbf{C}} \\ &= (\bar{\mathbf{B}} \cdot \mathbf{P}^{-1}) \cdot \Lambda^t \cdot (\mathbf{P} \cdot \bar{\mathbf{C}})\end{aligned}\tag{6.186}$$

Since  $\Lambda$  is a diagonal matrix, we can efficiently compute  $\Lambda^t$  by simply raising all the entries of  $\Lambda$  to the  $t$ -th power. We then have a computationally cheaper model, in which

$$\bar{\mathbf{A}}' = \Lambda \tag{6.187}$$

$$\bar{\mathbf{B}}' = \bar{\mathbf{B}} \cdot \mathbf{P}^{-1} \tag{6.188}$$

$$\bar{\mathbf{C}}' = \mathbf{P} \cdot \bar{\mathbf{C}} \tag{6.189}$$

$$\bar{\mathbf{D}}' = \bar{\mathbf{D}} \tag{6.190}$$

More detailed discussions of diagonal state-space models in sequence modeling can be found in Gu et al. [2021]’s work.

The application of state-space models to Transformer is simple. Each self-attention sub-layer is replaced in this case by an SSM sub-layer as described in Eqs. (6.164) and (6.165). As we have seen there is a close relationship between state-space models and both CNNs and RNNs. For sequence modeling, we can deal with a sequence of tokens either sequentially as in RNNs, or in parallel as in CNNs. This leads to a new paradigm that takes both the sequential view and the parallel view of the sequence modeling problem — for training, the

<sup>21</sup>A state space transformation can be seen as a process of mapping all states from the old space to the new space, by

$$\mathbf{s}'(t) = \mathbf{s}(t) \cdot \mathbf{U} \tag{6.185}$$

<sup>22</sup> $\Lambda$  denotes a diagonal matrix.

system operates like CNNs to make use of fast parallel training algorithms; for prediction, the problem is re-cast as a sequential update problem which can be efficiently solved by using RNN-like models. It should be noted, however, that state-space models are found to underperform Transformer models for NLP problems, such as language modeling, although they have achieved promising results in several other fields. Further refinements are often needed to make them competitive with other widely used sequence models [Fu et al., 2022].

While the formalism of state-space models is different from those we discussed in this chapter, it provides a general framework of sequence modeling in which the problem can be viewed from either of two different perspectives and we choose different ones for different purposes. Several recent sequence models were motivated by this idea, leading to systems exhibiting properties of both parallel training and RNN-style inference [Orvieto et al., 2023; Sun et al., 2023].

### 6.4.6 Conditional Computation

So far in our discussion of efficient Transformer models, we have assumed that the model architecture is given before beginning the training of a model and is then fixed throughout. We now turn to the case of learning efficient model architectures. Without loss of generality, we can write a model in the form

$$\mathbf{y} = \text{Model}(\mathbf{x}, g(\mathbf{x})) \quad (6.191)$$

where  $\mathbf{x}$  and  $\mathbf{y}$  are the input and output of the model.  $g(\mathbf{x})$  is a **model function** that returns the model architecture and corresponding parameters for the given input  $\mathbf{x}$ . In general, we adopt the convention prevalent in learning problems of using a fixed model architecture and learning only the parameters, say,  $g(\mathbf{x}) = \theta$ . In this case, the goal of learning is to find the optimal values of the parameters given the model architecture and training data. On test data, we make predictions using the same model architecture along with the optimized parameters.

A natural extension of this approach is to consider the learning of both the model architecture and parameters. In architecture learning, we would like to find a model function  $\hat{g}(\mathbf{x})$  that produces the optimal model architecture and parameter values given the input  $\mathbf{x}$ . However, searching a hypothesis space of all possible combinations of architectures and parameter choices is extremely difficult, and so we need practical methods to achieve the goal. Two classes of methods can be applied.

- **Neural Architecture Search (NAS).** In **automated machine learning (AutoML)**, neural architecture search is the process of exploring a space of neural networks to find one that best fits some criteria [Zoph and Le, 2016; Elsken et al., 2019b]. Once the optimal neural network is determined, its parameters will be trained as usual, and then be applied to new data. In order to make search tractable, several additional techniques, such as search space pruning and fast search algorithms, are typically used. Applying neural architecture search to the development of efficient neural networks is straightforward [Howard et al., 2019; Tan and Le, 2019]. We need only incorporate efficiency measures into the performance estimation of neural networks, for example, the search can be

guided by a criterion that penalizes neural networks with high latency or excessive memory requirements.

- **Dynamic Neural Networks.** The key idea of dynamic neural networks is to adapt a neural network dynamically to various inputs [Gupta et al., 2004; Han et al., 2021b]. Ideally, we would like to learn  $\hat{g}(\cdot)$ , and then, for any input  $\mathbf{x}_{\text{new}}$ , we apply the model  $\text{Model}(\mathbf{x}_{\text{new}}, \hat{g}(\mathbf{x}_{\text{new}}))$ . As a result, at test time we may have different model structures and/or different parameters for different inputs. However, it is infeasible to develop a function  $\hat{g}(\cdot)$  that can model arbitrary neural networks. In practice,  $\hat{g}(\cdot)$  is often considered to represent a family of sub-networks of a super-network. The problem is therefore reframed as a simpler problem to learn to choose which sub-network is used for a given input.

From a machine learning perspective, the approaches to neural architecture search are general and can be applied to any neural network. On the other hand, from a practical perspective, it is still difficult to find an efficient neural network that is sufficiently powerful and generalizes well. While neural architecture search provides interesting ideas for developing efficient Transformer models, we make no attempt to discuss it here. Instead, the reader can refer to the above papers to have a general idea of it, and refer to So et al. [2019], Wang et al. [2020a], and Hu et al. [2021]’s work for its application to Transformers.

In this subsection, we focus on a particular family of approaches to dynamic neural networks, called **conditional computation**. This concept was originally motivated by the dynamic selection of neurons of a neural network [Bengio et al., 2013; 2015]. More recently, it has often been used to refer to as a process of dynamically selecting parts of a neural network. A narrow view of conditional computation is to see  $g(\cdot)$  as an adaptive neural network which dynamically reduces or grows the number of computation units (such as neurons and layers). As a result, computation can adapt to changing conditions, and we can seek a good accuracy-latency trade-off by this adaptation mechanism.

A common way to achieve this is to learn how to skip some computation steps so that we can work with a necessary sub-set of the network [Xu and Mcauley, 2023]. One of the simplest methods, sometimes called **early stopping**, is to stop the computation at some point during reading or generating a sequence. This technique is often used in practical sequence generation applications where a low latency is required. Suppose  $y_1 \dots y_{n_{\text{max}}}$  is the longest sequence that the system can generate, and  $\mathbf{s}_1 \dots \mathbf{s}_{n_{\text{max}}}$  is the corresponding sequence of the states of the top-most Transformer layer. Then we develop a model  $f_{\text{stop}}(\cdot)$  that takes one hidden state  $\mathbf{s}_i$  at a time and produces a distribution of a binary variable  $c \in \{\text{stop}, \text{nonstop}\}$

$$\Pr(c|\mathbf{s}_i) = f_{\text{stop}}(\mathbf{s}_i) \quad (6.192)$$

The generation process terminates if  $\Pr(\text{stop}|\mathbf{s}_i)$  is sufficiently large, for example

$$\Pr(\text{stop}|\mathbf{s}_i) \geq \Pr(\text{nonstop}|\mathbf{s}_i) + \theta_{\text{stop}} \quad (6.193)$$

where  $\theta_{\text{stop}}$  denotes the minimal margin for distinguishing the two actions<sup>23</sup>. This formulation is also related to the stopping criterion problem that is frequently discussed in search algorithms for sequence generation (see Chapter 5).  $f_{\text{stop}}(\cdot)$  can be designed in several different ways. For example, in many practical applications, the stopping criterion is based on simple heuristics. Alternatively, we can define the function  $f_{\text{stop}}(\cdot)$  as a neural network and train it using labeled data.

The above approach can be easily extended to handle situations in which some of the tokens are skipped. This learning-to-skip approach is typically used in the encoding stage in which all input tokens are given in advance. Let  $\mathbf{h}_1 \dots \mathbf{h}_m$  be low-level representations of a sequence  $x_1 \dots x_m$ . Like Eq. (6.192), we can develop a model  $\Pr(c|\mathbf{s}_i)$  ( $c \in \{\text{skip}, \text{nonskip}\}$ ) to determine whether the token  $x_i$  can be skipped. Figure 6.16 (a) and (b) show illustrations of early stopping and skipping. Note that the learning-to-skip method has overlap with other lines of research on training neural networks. For example, erasing some of the input tokens in training is found to be useful for achieving higher generalization of Transformer models [Shen et al., 2020a; Kim and Cho, 2021]. This method is also related to the downsampling methods which will be discussed in Section 6.4.8.

A second approach to conditional computation is to resort to **sparse expert models**, or its popular instance — MoE [Yuksel et al., 2012]. In deep learning, a model of this kind is typically built from a number of experts which are neural networks having the same structure but with different parameters. In this way, we can construct a big model by simply increasing the number of experts. When running this model, during either training or prediction, we activate only a small number of the experts by some routing algorithms (see Figure 6.16 (c)). An MoE model is an adaptive network since each time we have a new input, the model routes it to different experts. In Section 6.3.4, we presented the basic form of MoE, and showed how Transformer models can be scaled up by this sparse method. For a comprehensive review of the recent advances in MoE, we refer the interested reader to Fedus et al. [2022a]’s work.

A third approach that can be used to adapt a Transformer model to changing input is to dynamically shrink the number of layers. Several methods have been proposed to do this in an attempt to improve inference efficiency. The simplest of these is to exit at some hidden layers by which we can still make accurate predictions for the sample (see Figure 6.16 (d) and (e)). To do this, we can either determine the appropriate depth for the entire sequence (call it a **sentence-level depth-adaptive model**), or use an adaptive depth for each token (call it a **token-level depth-adaptive model**). Here we consider token-level depth-adaptive models but the methods can be easily extended to sequence-level depth-adaptive models.

Suppose there are  $L$  stacked layers at position  $i$ <sup>24</sup>. We would ideally like to find a layer in the stack, which can be used as the last hidden layer for making predictions, and whose depth is as low as possible. However, we cannot simply use the  $L$ -th layer of the stack as the oracle for this problem, because we never know in advance what the last layer generates during inference. Instead, we need to determine whether the network should stop growing at depth  $i$ , considering the layers generated so far.

<sup>23</sup>An equivalent form of Eq. (6.193) is  $\Pr(\text{stop}|\mathbf{s}_i) \geq \frac{1+\theta_{\text{stop}}}{2}$ .

<sup>24</sup>A layer is a standard Transformer block consisting of a few sub-layers.

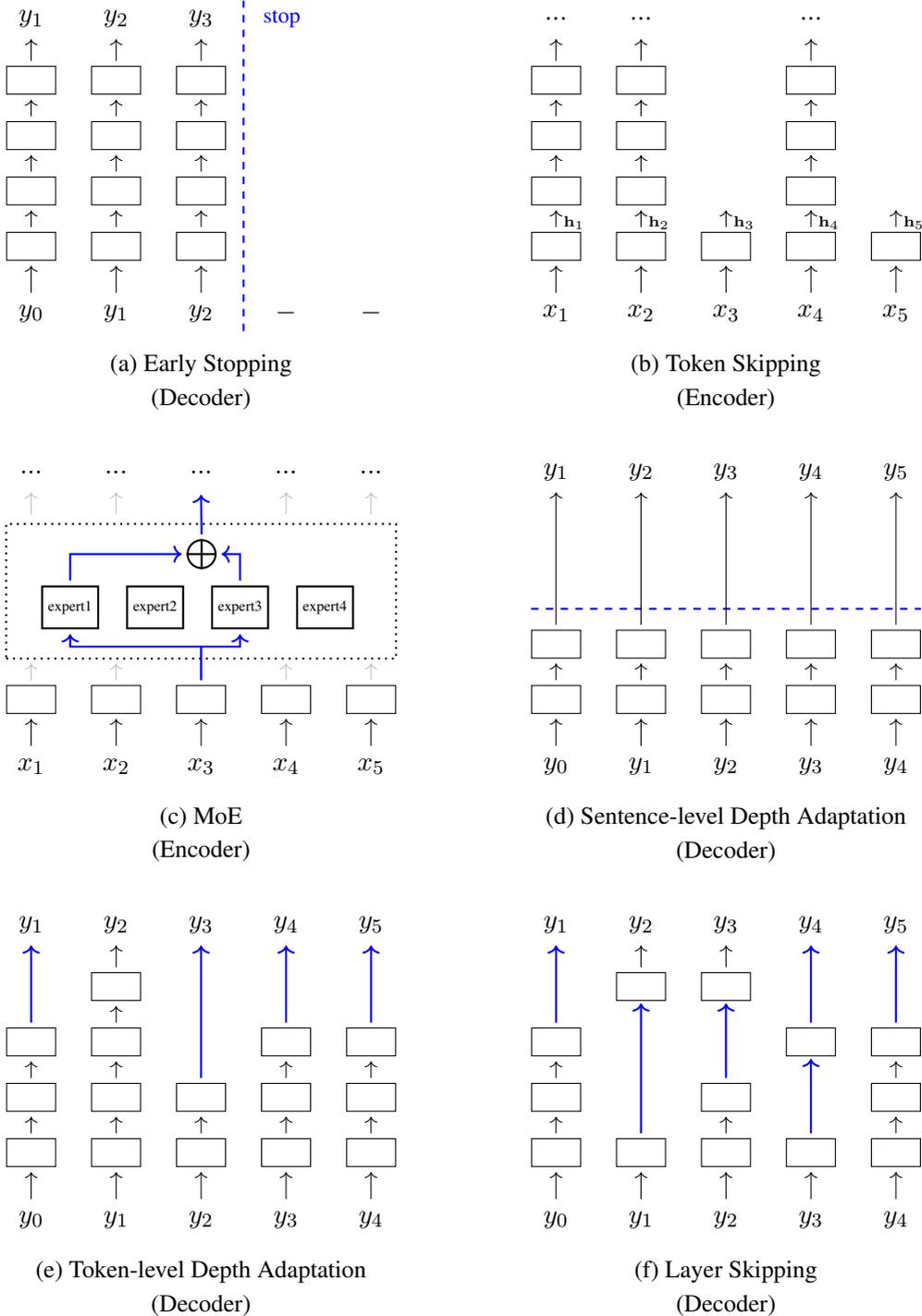


Figure 6.16: Methods of conditional computation, including early stopping, token skipping, MoE, sentence-level depth adaptation, token-level depth adaptation, and layer skipping. While these methods are illustrated using either the encoding or decoding process, most of them can be applied to both Transformer encoders and decoders.

Now suppose we have a Transformer decoder which produces a distribution over a vocabulary  $V$  at each step. As usual, we denote the output of the  $l$ -th layer at step  $i$  by  $\mathbf{s}_i^l$ . For each  $\mathbf{s}_i^l$ , we create an output layer that produces a distribution  $\mathbf{p}_i^l$  over the vocabulary (call it an **early exit classifier**), given by

$$\mathbf{p}_i^l = \text{Softmax}(\mathbf{s}_i^l \cdot \mathbf{W}_o^l) \quad (6.194)$$

where  $\mathbf{W}_o^l \in \mathbb{R}^{d \times |V|}$  is the parameter matrix. Hence we have  $L - 1$  additional output layers, each corresponding to a hidden layer from depth 1 to  $L - 1$ . At training time, we consider the cross-entropy losses of  $\{\mathbf{p}_i^1, \dots, \mathbf{p}_i^{L-1}\}$ , and train these layers together with the Transformer model. At test time, the depth of the network grows as usual, and we use  $\{\mathbf{p}_i^1, \dots, \mathbf{p}_i^L\}$  and/or  $\{\mathbf{s}_i^1, \dots, \mathbf{s}_i^L\}$  to determine whether we should exit at the  $l$ -th layer. There are several exit criteria, for example,

- Common criteria are based on measures of the confidence of predictions. A simple method is to compute the entropy of  $\mathbf{p}_i^l$ , and exit if this entropy is above a pre-defined value.
- Alternatively, one can view the maximum probability of the entries of  $\mathbf{p}_i^l$  as the confidence of the prediction.
- Instead of considering the output of a single layer, we can also examine the change in the outputs or hidden states over a number of layers. For example, we can measure the similarity between  $\mathbf{p}_i^{l-1}$  and  $\mathbf{p}_i^l$  or between  $\mathbf{s}_i^{l-1}$  and  $\mathbf{s}_i^l$ . If the similarity is above a given threshold, then we say that the output of the neural tends to converge and the number of layers can stop growing.
- The above methods can be extended to examine the change in the predictions made by the classifiers associated with the layers. For example, the model can choose to exit if the predictions made by the classifiers remain unchanged for a number of layers.

Discussions of these criteria can be found in the related papers [Xin et al., 2020; Zhou et al., 2020; Schuster et al., 2022]. There are a variety of ways to improve these early exit methods. One is to explore other forms of the prediction for each layer. For example, we can develop a model that directly predicts how many layers we need to model the input [Elbayad et al., 2020]. Another line of research on early exit focuses on better training for these models, for example, we can consider various loss functions for training the classifiers [Schwartz et al., 2020; Schuster et al., 2022]. In addition, there is also interest in learning the combination of the outputs of multiple layers so that we can make predictions by using multiple levels of representation [Zhou et al., 2020; Liao et al., 2021].

A problem with token-level adaptive-depth models is that the representations at certain depths may be absent in the previous steps. In this case, standard self-attention is not directly applicable, because we may not attend to the previous tokens in the same level of representation. For training, this can be addressed by using all the  $L$  layers of the full model. For inference, we can either duplicate the layer from which we exit to fill up the layer stack, or modify the self-attention model to enable it to attend to the representations of the previous tokens at

different depths.

It is also possible to select any sub-set of the layers for constructing a shallow network. The adaptive models therefore can be generalized to skipping models (see Figure 6.16 (f)). As with the early exit problem, the skipping problem can be framed as a learning task, in which a classifier is trained to decide whether a layer should be dropped. The learning-to-skip problem has been studied in the field of computer vision [Wang et al., 2018b; Wu et al., 2018b]. However, learning a skipping model for large-scale, deep neural networks is difficult. For practical systems, it still seems reasonable to use heuristics or cheap models to obtain a neural network having skipped layers, which has been discussed in recent pre-trained NLP models [Wang et al., 2022c; Del Corro et al., 2023].

### 6.4.7 Model Transfer and Pruning

Many large Transformer models have been successfully developed to address NLP problems. A common question is: can we transform a large, well-trained model into a smaller one that allows for more efficient inference? At a high level, this can be thought of as a **transfer learning** problem in which the knowledge is transferred from one model to another. But we will not discuss this general topic, which spans a broad range of issues and models, many outside the scope of this chapter. Instead, we narrow our discussion to two kinds of approaches that are widely used in learning small neural networks from large neural networks.

#### 1. Knowledge Distillation

**Knowledge distillation** is a process of compressing the knowledge in a large neural network (or an ensemble of neural networks) into a small neural network [Hinton et al., 2015]. In supervised learning of neural networks, the objective functions are generally designed to represent some loss of replacing the true answer with the predicted answer. Hence we can minimize this loss so that the models are trained to output the true answer. While models are typically optimized on the training data in this manner, what we really want is to generalize them to new data. This is, however, difficult because we have no information about generalization in training with the ground-truth. In knowledge distillation, instead of forcing a model to stay close to the ground-truth output, we train this model to generalize. To do this, we directly transfer the knowledge (i.e., the generalization ability) of a pre-trained model to the model that we want to train.

A frequently used approach to knowledge distillation is **teacher-student training**. A teacher model is typically a relatively large neural network that has already been trained and can generalize well. A student model is a relatively small neural network, such as a neural network with fewer layers, to which we transfer the knowledge. A simple way to distill the knowledge from the teacher model into the student model is to use the output of the teacher model as the “correct” answer for training the student model. Suppose we have a teacher Transformer model that can generate a sequence of distributions  $\{\Pr(\cdot|y_0, \mathbf{x}), \dots, \Pr(\cdot|y_0 \dots y_{n-1}, \mathbf{x})\}$  for the input  $\mathbf{x}$ . To keep the notation simple, we denote the distribution  $\Pr(\cdot|y_0 \dots y_{i-1}, \mathbf{x})$  as  $\tilde{\mathbf{p}}_i$ . Similarly, we denote the output of the student Transformer model for the same input as  $\mathbf{p}_i$ . As usual, we consider a loss function  $Loss(\tilde{\mathbf{p}}_i, \mathbf{p}_i)$  (such as the cross-entropy function) for computing some

distance between  $\tilde{\mathbf{p}}_i$  and  $\mathbf{p}_i$ . Then, we can define the loss over the entire sequence as

$$L(\mathbf{x}, \theta) = \frac{1}{n} \sum_{i=1}^n \text{Loss}(\tilde{\mathbf{p}}_i, \mathbf{p}_i) \quad (6.195)$$

where  $\theta$  denotes the parameters of the student model<sup>25</sup>. Using this loss, we can optimize  $\theta$ , for any given set of source sequences  $\{\mathbf{x}_1, \dots, \mathbf{x}_K\}$ , in such a way as to minimize the quality  $\sum_{k=1}^N L(\mathbf{x}_k, \theta)$ .

Several different extensions to this basic method have been developed to model the problem of knowledge transfer between two models. A simple way is to use the hidden states instead of the output probabilities as the training targets [Romero et al., 2014]. In this case, the objective is to minimize the difference between some hidden states of the teacher model and the corresponding states of the student model. Rather than using the outputs of various layers as the targets for training the student model, another technique is to model the relations between samples and train the student model by minimizing some differences between the relation encodings of the teacher and student models [Park et al., 2019; Peng et al., 2019]. For example, we can develop a relation encoding model based on the Transformer architecture. The goal is then to optimize the student model so that its corresponding relation encoding of a group of samples is as close as possible to that of the teacher model.

For sequence generation problems, a special case of knowledge distillation, which can be viewed as a means of **data augmentation**, is often used for developing lightweight models [Kim and Rush, 2016]. For example, consider the problem of transferring the translation ability of a well-developed machine translation model (i.e., the teacher model) to a new model (i.e., the student model). Given a set of source-side sentences  $\{\mathbf{x}_1, \dots, \mathbf{x}_K\}$ , we can use the teacher model to translate each  $\mathbf{x}_k$  to a target-side sentence  $\tilde{\mathbf{y}}_k$ . Then, by treating  $\mathbf{x}_k$  and  $\tilde{\mathbf{y}}_k$  as paired sentences, we obtain a bilingual dataset consisting of  $\{(\mathbf{x}_1, \tilde{\mathbf{y}}_1), \dots, (\mathbf{x}_K, \tilde{\mathbf{y}}_K)\}$ . We can use this bilingual dataset as the labeled dataset to train the student model as usual. One advantage of this data argumentation method is that it is architecture free, and we do not even need to understand the internal architectures of the teacher and student models. Hence we can apply this method if we have a black-box teacher model. More detailed discussions of knowledge distillation can be found in Gou et al. [2021] and Wang and Yoon [2021]'s surveys.

## 2. Structured Pruning

Pruning is among the most popular of the model compression methods and has been applied to a broad range of systems. One common approach to pruning is **unstructured pruning**, by which we activate only some of the connections between neurons. However, as with most sparse models, models pruned in this way typically require special implementations and hardware support, which in turn reduces their efficiency in some applications. A simple but more aggressive way to do pruning is to use **structured pruning**. In deep learning, structured pruning is a technique that removes a group of neurons or connections together. For example, we can remove an entire layer of neuron from a neural network to obtain a shallower model.

<sup>25</sup>We omit the parameters of the teacher model because they are fixed throughout the training process.

As multi-layer, multi-head neural networks, Transformers are naturally suited to structured pruning, and we can prune a Transformer network in several different ways. For example, we can prune some of the heads in multi-head attention [Voita et al., 2019; Michel et al., 2019], or some of the layers in the layer stack [Hou et al., 2020; Kim and Awadalla, 2020].

Formally, we can represent a neural network as a set of parameter groups  $\{\theta_1, \dots, \theta_R\}$ , each corresponding to a component or sub-model of the model. Our goal is to find a sub-set of  $\{\theta_1, \dots, \theta_R\}$  by which we can build a model that yields good performance, while having a lower model complexity. However, a simple search of such a model is infeasible because there are a combinatorially large number of possible model candidates and evaluating all of these models is computationally expensive.

One approach to structured pruning is to randomly prune components of a model. One can run the random pruning process a number of times to generate a pool of model candidates and select the best one from the pool. Another approach is to use heuristics to decide which components are not important and can be removed. Common measures of the importance of a parameter group  $\theta_r$  include various qualities based on norms of the weights or gradients of  $\theta_r$  [Santacrose et al., 2023]. We can prune  $\theta_r$  if the values of these measures are below (or above) given thresholds. A third approach is to frame the pruning problem as an optimization task by introducing trainable gates indicating the presence of different components [McCarley et al., 2019; Wang et al., 2020d; Lagunas et al., 2021]. The pruned model can be induced by using the trained gates. Note that, in many cases, pruning is not a post-processing step for a given trained model, but part of the training.

### 6.4.8 Sequence Compression

In sequence modeling and generation problems, the time and space complexities are strongly influenced by the length of the input or output sequence, and we prefer the sequence to be short. This is particularly important for Transformer models, as their time and space complexities are quadratic with the sequence length, and the memory footprint and latency can be heavy burdens if the sequence is very long. In the previous subsections, we have discussed modifications to the Transformer architecture for dealing with long sequences. Here we instead consider methods for compressing the sequences into ones with acceptable lengths.

One simple approach is to map the input sequence to a fixed-size representation. For example, using the recurrent models discussed in Section 6.4.2, we can encode a sequence of vectors into a single vector. This method can be easily extended to generate a “larger” representation so that this representation can retain more information of the original input. For example, we can select a fixed number of the hidden states over the sequence to form a new sequence of fixed-length. Another way to represent a variable-length sequence as a fixed-length sequence is to attend the input vectors to some hidden states, usually a fixed number of learnable hidden representations. In Jaegle et al. [2021]’s work, this is done by introducing  $r$  hidden representations  $\{\mathbf{u}_1, \dots, \mathbf{u}_r\}$ , and then attending the input vectors  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  to these hidden representations. The attention model can be a standard QKV attention model in which we view  $\{\mathbf{u}_1, \dots, \mathbf{u}_r\}$  as queries and  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  as keys and values. The output of this model is a sequence of  $r$  vectors, which can be used as fixed-length input to downstream systems.

A second approach is to use downsampling to compress the sequence into a shorter one. A typical method of downsampling is strided convolution, which has been widely used in computer vision and speech processing. For example, suppose there is a sequence of  $m$  vectors  $\in \mathbb{R}^d$ . We can develop a filter with a width of 2 and a stride of 2. By taking the sequence as input, the filter produces a sequence of  $\frac{m}{2}$  new vectors  $\in \mathbb{R}^d$ , and so we have a reduction rate of 2. Also, we can stack multiple convolutional layers or pooling layers to achieve a desired level of length reduction, called **progressive downsampling**. However, it seems inevitable that downsampling will lead to information loss [Han et al., 2020; Burchi and Vielzeuf, 2021]. We need to consider a trade-off between the compressed sequence length and the performance of downstream systems [Xu et al., 2023b].

In NLP, the problem of sequence compression is also closely related to the problem of tokenizing input strings. Therefore, tokenization is a practical approach that can be taken to address the length issue. Segmenting a string into small tokens (such as characters) generally reduces the sparsity of the data, which makes it easier to learn the embeddings of these tokens, but such approaches often lead to a long sequence. By contrast, we will have a shorter sequence if we segment the input string into larger units, but this will suffer from sparse data. In deterministic tokenization methods, which produce tokenization results using statistics collected from the entire dataset, the sequence length can be somehow controlled by adjusting some hyper-parameter, for example, in byte pair encoding [Sennrich et al., 2016b], increasing the size of the vocabulary generally reduces the number of the resulting tokens. Another way to obtain an appropriate sequence of tokens is to use a model for choosing among tokenization candidates [Kudo, 2018; Provilkov et al., 2020]. As with many probabilistic models for text generation, in this case, we can add priors to the criterion for tokenization selection so that we can express a preference for shorter sequences over longer sequences.

A fourth approach to sequence compression is to drop some of the tokens in the sequence. For example, in many practical applications, we chop the sequence when its length exceeds a threshold. We can relate this to the early stopping and skipping approaches in conditional computation. Thus the methods discussed in Section 6.4.6 are directly applied. The token dropping methods can also be viewed as pruning methods, called **token pruning**. By discarding tokens that are less important for representing the entire sequence, token pruning can significantly reduce the sequence length while maintaining the performance of NLP systems on downstream tasks [Kim et al., 2023].

### 6.4.9 High Performance Computing Methods

So far in this section, we have discussed efficient Transformer models from the perspectives of deep learning and NLP. However, we have not considered their efficiency on hardware. As modern hardware provides a variety of modes for running a program, the practical time and memory footprint savings generally depend on the specifications of hardware systems. One line of research on efficient use of computing resources explores methods of parallel computing. There have been many attempts to develop large-scale Transformer models by using a cluster of machines. Typically, scaling Transformers to models with billions or even tens of billions of parameters requires a careful design of parallelism strategies for sharding

the big networks. More efficient implementations of such systems also need considerations of networking and communication in the cluster, as well as the utilization of sparse models that activate only a small sub-set of the parameters for each sample, enabling the use of very large models. Most of these methods have been studied in an extensive literature on how to scale up the training of deep neural networks like Transformers efficiently [Lepikhin et al., 2021; Barham et al., 2022; Fedus et al., 2022b]. The results of these studies were foundational to many follow-on works on investigating the **scaling laws** for large language models [Brown et al., 2020; Chowdhery et al., 2022]. Since large-scale distributed models are generic and not specialized to the case of Transformers, we skip the discussion of them here. The interested readers can refer to the above papers for more detailed discussions.

In this subsection, we consider hardware-aware methods to seek greater computational efficiency for Transformer models. We first consider a simple but widely used method that aims to store and execute neural networks using lower or mixed-precision number representations [Gholami et al., 2022]. Conventional neural networks are typically based on single-precision and/or double-precision floating-point representations of data. While single-precision floating-point data types provide a sufficiently precise way to represent parameters and intermediate states in most cases, in some applications, they are not essential. As an alternative, one can use half-precision (or even lower-precision) formats in storing floating-point numbers for neural networks. The size of the resulting model is thus half the size of the original model. One advantage of using half-precision floating-point representations is that, although processing such data types requires new APIs of linear algebra operations and hardware support, it does not change the model architecture, and so we need only a slight modification to the systems. For example, half-precision floating-point representations can be applied to either training or inference of Transformers, or both.

Recently, the deployment of large Transformer models has been further improved by quantizing these models. In signal processing, quantization is a process of mapping continuous values (i.e., floating-point representations) to a set of discrete values (i.e., fix-point representations). This process is in general implemented using a system called quantizer. In the context of neural networks, a quantizer involves two functions — the quantization function and the de-quantization function. The quantization function maps a floating point number to a (lower-bit) integer. A simple quantization function is given by

$$Q(x) = \lfloor \frac{x}{s} \rfloor \quad (6.196)$$

where  $\lfloor \cdot \rfloor$  is a rounding function<sup>26</sup>,  $x$  is the real-valued input, and  $s$  is the quantization step size that controls the level of quantization. The quantization function is coupled with a de-quantization function

$$D(r) = s \cdot r \quad (6.197)$$

With this notation, the quantizer can be expressed as  $D(Q(x)) = s \cdot \lfloor \frac{x}{s} \rfloor$ . The difference

---

<sup>26</sup> $\lfloor a \rfloor$  returns the integer closest to  $a$ .

between  $D(Q(x))$  and  $x$  is called quantization error. A smaller value of  $s$  typically reduces the quantization error. In practice, however, we wish to choose an appropriate value of  $s$  in order to spread possible values of  $Q(r)$  evenly across values of an integer, for example,  $s = \frac{\max\{D(r)\}}{2^p - 1}$  where  $p$  is the number of bits used to represent an integer and  $\max\{D(r)\}$  is the maximum value for  $D(r)$ . The above equations show one of the simplest cases of quantization. More general discussions of quantization can be found in books on digital signal processing and related surveys [Oppenheim and Schaffer, 1975; Rabiner and Gold, 1975; Gray, 1998].

Applying quantization to Transformers is relatively straightforward. The idea is that we quantize the inputs and model parameters using  $Q(x)$ , and feed them to a quantized Transformer model in which all the layers operate on integer-valued tensors. In other words, we implement the model using integer-only arithmetic. However, the price to be paid for this compressed model, as with many approximation approaches to deep learning, is that its prediction is not as accurate as that of the standard Transformer model. Using integer operations to approximate continuous-valued operations generally leads to approximation errors. These errors will be accumulated if the quantized neural network is deep. Furthermore, Transformer models involve components (such as self-attention sub-layers) that require relatively complex linear algebra operations. Simply applying quantization to these sub-models will lead to high accuracy loss. One solution is to simplify the model architecture and develop new sub-models that is more feasible for quantization. Alternatively, a more common paradigm in quantized neural networks is to add de-quantization functions to the neural networks so that the output of a layer is floating-point tensors and can be used as usual in the following steps. Consider a simple example where we multiply a real-valued input matrix  $\mathbf{a}$  with a real-valued parameter matrix  $\mathbf{A}$ . We first quantize  $\mathbf{a}$  and  $\mathbf{A}$ , and multiply them using integer-based matrix multiplication. The result is then de-quantized to a real-valued matrix. In this way, we obtain an approximation  $D(Q(\mathbf{a}) \cdot Q(\mathbf{A}))$  to  $\mathbf{a} \cdot \mathbf{A}$  in a very cheap way.

However, sandwiching each layer between  $Q(\cdot)$  and  $D(\cdot)$  will lead to additional cost of running  $Q(\cdot)$  and  $D(\cdot)$ . In some practical applications, the computational overhead introduced by  $Q(\cdot)$  and  $D(\cdot)$  is even bigger than the time saving of performing integer-based operations. In general, the benefit of quantizing neural networks would be larger than its cost if the neural networks are large. Therefore, in practice it is common to perform quantized computation only for operations whose computational costs are high. For example, in recent large language models, quantization is primarily applied to the multiplication of large matrices, yielding significant time and memory savings.

While the quantization approaches can be used in both training and inference, a widely-used approach is to get Transformer models quantized after training (call it **post-training quantization**). In this approach, quantization is performed on well-trained floating-point-based neural networks and there will be fewer quantization-related errors. However, these errors cannot be compensated for because they exist after training. A more promising idea is to involve quantization in training so that the model can learn to compensate for quantization-related errors [Jacob et al., 2018; Nagel et al., 2021]. There have been several attempts to apply quantization-aware training to Transformers [Bondarenko et al., 2021; Stock et al., 2021; Yang et al., 2023b]. In addition to computational efficiency, another important consideration for

high-performance systems is the restrictions of the memory hierarchy. In general, better system design requires considering the speeds and sizes of different levels of memory. The problem is even more complicated when we train large Transformer models on modern hardware where both GPUs and CPUs are used. A general principle of system design is that memory transfer between different memory levels should be minimized. While we would ideally like to have a large high-level memory on which we can store all the data that we need to process, in many practical situations the size of the fast, on-chip memory is orders of magnitude smaller than the size of data. In this case, we can re-order the memory access in the algorithms so that the data used in nearby computation steps can be loaded into the high-speed memory at one time. This idea motivates the development of many fast linear algebra libraries. For example, there are matrix multiplication algorithms that are highly optimized for different shapes of input matrices.

It is relatively straightforward to use these optimized linear algebra algorithms to build a Transformer system. But the modules of this system are not optimized as a whole for efficiency improvement. For example, a self-attention sub-layer involves a series of operations of scaling, normalization, and matrix multiplication. Although each of these operations has been implemented in several supported and efficient libraries of linear algebra, successive calls to them still require multiple times of memory transfer when we switch from one operation to another. In practice, a better approach would be that we keep some of the intermediate states in the on-chip memory, and reuse them in the following computation steps instead of fetching them again from the slow memory. For example, on modern GPUs, a simple way to achieve this is to merge multiple operations into a single operation, known as **kernel fusion**. For Transformer models, a general idea is to design data partitioning and layout strategies by which we maximize the computation on each data block loaded into the high-performance memory, while at the same time minimizing the memory transfer. There have been several attempts to use these strategies to improve the attention models in Transformers [Ivanov et al., 2021; Pope et al., 2023]. Some of these methods, such as flash attention and paged attention, have been successfully incorporated into recent large language models [Dao et al., 2022; Kwon et al., 2023].

## 6.5 Applications

Transformers have a wide range of applications, covering numerous NLP problems. While the Transformer model introduced by Vaswani et al. [2017] is based on a standard encoder-decoder architecture, it is mainly used in three different ways.

- **Decoder-only Models.** By removing the cross-attention sub-layers from a Transformer decoder, the decoder becomes a standard language model. Hence this decoder-only model can be applied to text generation problems. For example, given a sequence of left-context tokens, we use the model to predict the next and following tokens.
- **Encoder-only Models.** Transformer encoders can be treated as sequence models that take a sequence of tokens at once and produce a sequence of representations, each of

which corresponds to an input token. These representations can be seen as some sort of encoding of the input sequence, and are often taken as input to a prediction model. This encoder+predictor architecture forms the basis of many NLP systems, for example, systems of sentence classification, sequence labeling, and so on. Pre-trained Transformer encoders can also be used to map texts into the same vector space so that we can compute the distance or similarity between any two texts.

- **Encoder-Decoder Models.** Encoder-decoder models are typically used to model sequence-to-sequence problems. Applications include many tasks in NLP and related fields, such as machine translation and image captioning.

Note that while most Transformer-based systems can fall into the above three categories, the same NLP problem can generally be addressed using different types of models. For example, recent decoder-only models have demonstrated good performance on a broad range of problems by framing them as text generation tasks, though some of these problems were often addressed by using encoder-decoder or encoder-only models. To illustrate how the above models are applied, this section considers a few applications where Transformers are chosen as the backbone models.

### 6.5.1 Language Modeling

Language modeling is an NLP task in which we predict the next token given its preceding tokens. This is generally formulated as a problem of estimating the distribution of tokens at position  $i + 1$  given tokens at positions  $0 \sim i$  (denoted by  $\Pr(\cdot|x_0, \dots, x_i)$  where  $\{x_0, \dots, x_i\}$  denote the tokens up to position  $i$ ). The best predicted token is the one which maximizes the probability, given by

$$\hat{x}_{i+1} = \arg \max_{x_{i+1} \in V} \Pr(x_{i+1}|x_0, \dots, x_i) \quad (6.198)$$

where  $V$  is the vocabulary. The prediction  $\hat{x}_{i+1}$  can be extended to the tokens following  $\hat{x}_{i+1}$

$$\hat{x}_{k+1} = \arg \max_{x_{k+1} \in V} \Pr(x_{k+1}|x_0, \dots, x_i, \hat{x}_{i+1}, \dots, \hat{x}_k) \quad (6.199)$$

This model forms the basis of many systems for text generation: given the context tokens  $x_1 \dots x_i$ , we generate the remaining tokens  $\hat{x}_{i+1} \dots \hat{x}_{k+1}$  to make the sequence complete and coherent.

As discussed in Section 6.1.1, Transformer decoders are essentially language models. The only difference between the problem of decoding in an encoder-decoder Transformer and the problem of language modeling is that the Transformer decoder makes predictions conditioned on the “context” tokens on both the encoder and decoder sides, rather than being conditioned on preceding tokens solely on one side. To modify the Transformer decoder to implement a standard language model, the cross-attention sub-layers are simply removed and a Transformer

decoding block can be expressed as

$$\mathbf{S}^l = \text{Layer}_{\text{ffn}}(\mathbf{S}_{\text{self}}^l) \quad (6.200)$$

$$\mathbf{S}_{\text{self}}^l = \text{Layer}_{\text{self}}(\mathbf{S}^{l-1}) \quad (6.201)$$

Here  $\mathbf{S}^l$  denotes the output of the block at depth  $l$ .  $\text{Layer}_{\text{self}}(\cdot)$  denotes the self-attention sub-layer, and  $\text{Layer}_{\text{ffn}}(\cdot)$  denotes the FFN sub-layer. We see that this decoding block has the same form as an encoding block. The difference between the decoding and encoding blocks arises from the masking strategies adopted in training, because the former masks the attention from a position  $i$  to any right-context position  $k > i$  whereas the latter has no such restriction. A Softmax layer is stacked on the top of the last block, and is used to produce the distribution over the vocabulary at each position. For inference, the Transformer decoder works in an auto-regressive manner, as described in Eq. (6.199).

The training of this model is standard. We learn the model by repeatedly updating the parameters, based on the gradients of the loss on the training samples. This paradigm can be extended to the training of large Transformer-based language models, which have been widely applied in generative AI. However, training Transformer models at scale, including decoder-only, encoder-only, and encoder-decoder models, may lead to new difficulties, such as training instabilities. We will discuss these issues further in the following chapters, where large-scale pre-training is the primary focus.

### 6.5.2 Text Encoding

For many NLP problems, a widely used paradigm is to first represent an input sequence in some form, and then make predictions for downstream tasks based on this representation. As a result, we separate sequence modeling or sequence representation from NLP tasks. One of the advantages of this paradigm is that we can train a sequence model that is not specialized to particular tasks, thereby generalizing well.

Clearly, Transformer encoders are a type of sequence model, and can be used as text encoders. Consider a Transformer encoder with  $L$  encoding blocks. The output of the last encoding block can be seen as the encoding result. Here add a special token  $x_0$  to any sequence, indicating the beginning of a sequence (written as  $\langle \text{SOS} \rangle$  or  $[\text{CLS}]$ ). If there is a sequence of  $m + 1$  input tokens  $x_0 x_1 \dots x_m$ , the output of the encoder will be a sequence of  $m + 1$  vectors  $\mathbf{h}_0^L \mathbf{h}_1^L \dots \mathbf{h}_m^L$ . Since  $x_0$  is not a real token and has a fixed positional embedding, it serves as a tag for collecting information from other positions using the self-attention mechanism. Hence  $\mathbf{h}_0^L$  is a representation of the entire sequence, with no biases for any specific tokens or positions. In many cases, we need a single representation of a sequence and take it as input to downstream components of the system, for example, we can construct a sentence classification system based on a single vector generated from  $\{\mathbf{h}_0^L, \dots, \mathbf{h}_m^L\}$ . In this case, we can simply use  $\mathbf{h}_0^L$  as the representation of the sequence. A more general approach is to add a pooling layer to the encoder. This allows us to explore various pooling methods to generate the sequence embedding from  $\{\mathbf{h}_0^L, \dots, \mathbf{h}_m^L\}$ .

In text encoding, token sequences are represented by real-valued vectors, often referred to

as sentence representations or sentence embeddings, which can be seen as points in a multi-dimensional space [Hill et al., 2016]. Another way to make use of text encoding, therefore, is to obtain semantic or syntactic similarities of token sequences based on their relative positions or proximity in this space. A straightforward method for this is to compute the Euclidean distances between sequence embeddings. The shorter the distance between two sequences, the more similar they are considered to be. There are many distance metrics we can choose, and it is possible to combine them to obtain a better measure of sequence similarity. Such similarity computations are applied in areas such as text entailment, information retrieval, translation evaluation, among others [Cer et al., 2018; Reimers and Gurevych, 2019]. Additionally, they are often used to assess the quality of text encoding models.

Text encoding is also a crucial component of sequence-to-sequence models. Given this, we can develop a separate Transformer encoder for source-side sequence modeling in an encoder-decoder system (see Figure 6.17). For example, we can pre-train a Transformer encoder on large-scale source-side texts, and use it as the encoder in a downstream encoder-decoder model. It is worth noting that while the encoder is designed based on the Transformer architecture, the decoder is not confined to just Transformers. Such flexibility enables us to incorporate pre-trained Transformer encoders into hybrid sequence-to-sequence architectures, such as systems that combine a Transformer encoder with an LSTM decoder.

In supervised learning scenarios, training a Transformer encoder is straightforward. We can treat it as a regular component of the target model and train this model on task-specific labeled data. However, such a method requires the encoder to be optimized on each task, and the resulting encoder might not always generalize well to other tasks, especially given that labeled data is scarce in most cases. A more prevalent approach is to frame the training of text encoders as an independent task in which supervision signals are derived solely from raw text. This led researchers to develop self-supervised Transformer encoders, such as BERT, which make use of large-scale unlabeled text, and these encoders were found to generalize well across many downstream tasks. Further discussions of pre-trained Transformer encoders can be found in Chapter 7.

### 6.5.3 Speech Translation

As illustrated in Section 6.1, the standard encoder-decoder Transformer model was proposed to model sequence-to-sequence problems. Here we consider the problem of translating speech in one language to text in another language — a problem that is conventionally addressed using both automatic speech recognition (ASR) and machine translation techniques. Instead of cascading an automatic speech recognition system and a machine translation system, we can use Transformer models to build an end-to-end speech-to-text (S2T) translation system to directly translate the input speech to the output text.

To simplify the discussion, we assume that the input of an S2T translation system is a sequence of source-side acoustic feature vectors, denoted by  $\mathbf{a}_1 \dots \mathbf{a}_m$ , and the output of the system is a sequence of target-side tokens, denoted by  $y_1 \dots y_n$ .<sup>27</sup> Mapping  $\mathbf{a}_1 \dots \mathbf{a}_m$  to  $y_1 \dots y_n$  is a sequence-to-sequence problem. Thus it is straightforward to model the problem using an

<sup>27</sup>In order to obtain the input sequence to the system, we need to discretize continuous speech into signals

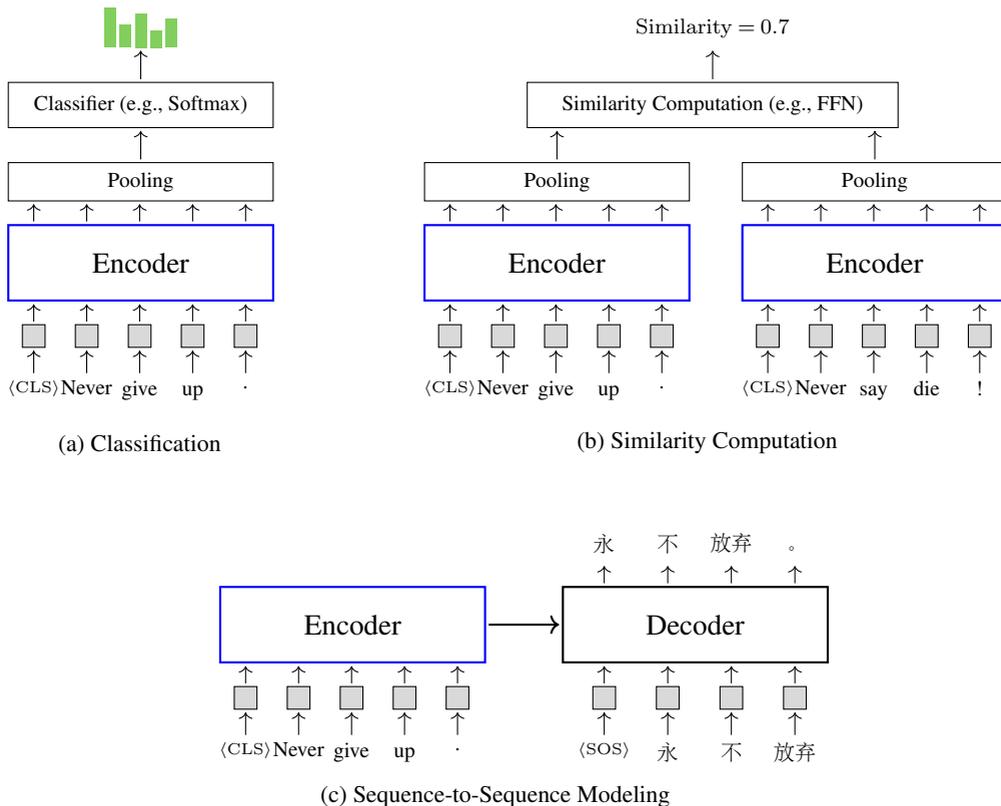


Figure 6.17: Integrating Transformer encoders as components of different systems. A common approach is to feed the output of the encoder (with pooling) into a classifier to obtain a sequence classification system. Another way to utilize Transformer encoders is to compute the similarity between two sequences. We use the same encoder to represent the two sequences, and then construct a neural network on top of the two representations for producing a similarity score between them. As usual, Transformer encoders can also be used in encoder-decoder systems to model sequence-to-sequence problems.

encoder-decoder Transformer model, and the training and inference of this model are standard, like in neural machine translation.

In S2T translation, however, we have to deal with sequence mappings between modalities and between languages simultaneously. This poses new challenges compared with conventional machine translation problems and influences the design of S2T translation models. There have been several improvements to Transformer models for adapting them better to S2T translation tasks. Some of the improvements concern the design of Transformer blocks [Di Gangi et al., 2019]. For example, in Gulati et al. [2020]’s system, a CNN sub-layer and relative positional embeddings are integrated into each Transformer block, enabling the model to efficiently

---

represented by feature vectors. This process is typically nontrivial, requiring either a feature extractor based on a variety of signal processing operations or a neural network that learns feature mappings in an end-to-end manner. But we will not dive into the details of these methods and simply treat the input feature extractor as an upstream system.

capture both local and global features.

Another line of research on S2T translation focuses on improving the encoder-decoder architecture. This involves modifications to either encoders or decoders, or both. To illustrate, Figure 6.18 shows the architectures of three S2T translation models. All of them are based on Transformers, but have different encoder architectures. As shown in the figure, the standard encoder-decoder architecture has one Transformer encoder for reading the source-side input  $a_1 \dots a_m$  and one Transformer decoder for producing the target-side output  $y_1 \dots y_n$ . By contrast, the decoupled encoder model separates the encoder into two stacked encoders — one for acoustic modeling (call it the **speech encoder**), and one for textual modeling (call it the **text encoder**) [Liu et al., 2020d; Xu et al., 2021a]. This design reflects a modeling hierarchy in which representations in different levels of the network are concerned with different aspects of the problem, for example, the speech encoder models low-level features in mapping acoustic embeddings into larger language units, and the text encoder models the semantic or syntactic features in representing the entire input sequence. An advantage of separating out the text encoder is that the encoding process follows our prior knowledge that we need to first transcribe the speech input and then translate the transcript into the target language. Therefore, we can train the speech encoder in some way we train an ASR system. This enables us to pre-train the speech encoder and the text encoder on unlabeled data, and incorporate the pre-trained encoders into S2T translation systems.

An alternative encoding architecture is the two-stream architecture, as shown in Figure 6.18 (c). Like the decoupled encoder architecture, this architecture has a speech encoder and a text encoder, but the two encoders work in parallel rather than in sequence [Ye et al., 2021]. The speech encoder takes acoustic features as input and the text encoder takes tokens (or their embeddings) as input. A third encoder, called **shared encoder**, integrates the outputs from both the speech and text encoders, merging the representations from the two modalities. This two-stream architecture is flexible because it provides multiple ways to train S2T translation models. A common approach is to train each branch individually. For example, if we mask the speech encoder, then the model will transform into a machine translation model which can be trained using bilingual texts. Conversely, if we mask the text encoder, then we can train the model as a standard S2T translation model. For inference, the text encoder can be dropped, and the speech input is modeled using the speech encoder and the shared encoder.

In deep learning, training is often related to architecture design. Here, we have data in two modalities and two languages, and so we can develop multiple supervision signals for multi-task learning of S2T translation models. A widely used method is to introduce ASR-related loss into the training of speech encoders. For example, in the decoupled encoder model, a classifier can be constructed based on the output from the speech encoder. By minimizing the connectionist temporal classification (CTC) loss for this classifier, the speech encoder can be optimized in a manner similar to ASR. In general, training S2T translation models is challenging because speech-to-text aligned data is scarce. Among typical responses to this challenge are data augmentation, pre-training, knowledge distillation with machine translation, and so on. However, an in-depth discussion of these methods goes beyond the scope of this discussion on Transformers. The interested reader can refer to a recent survey on speech

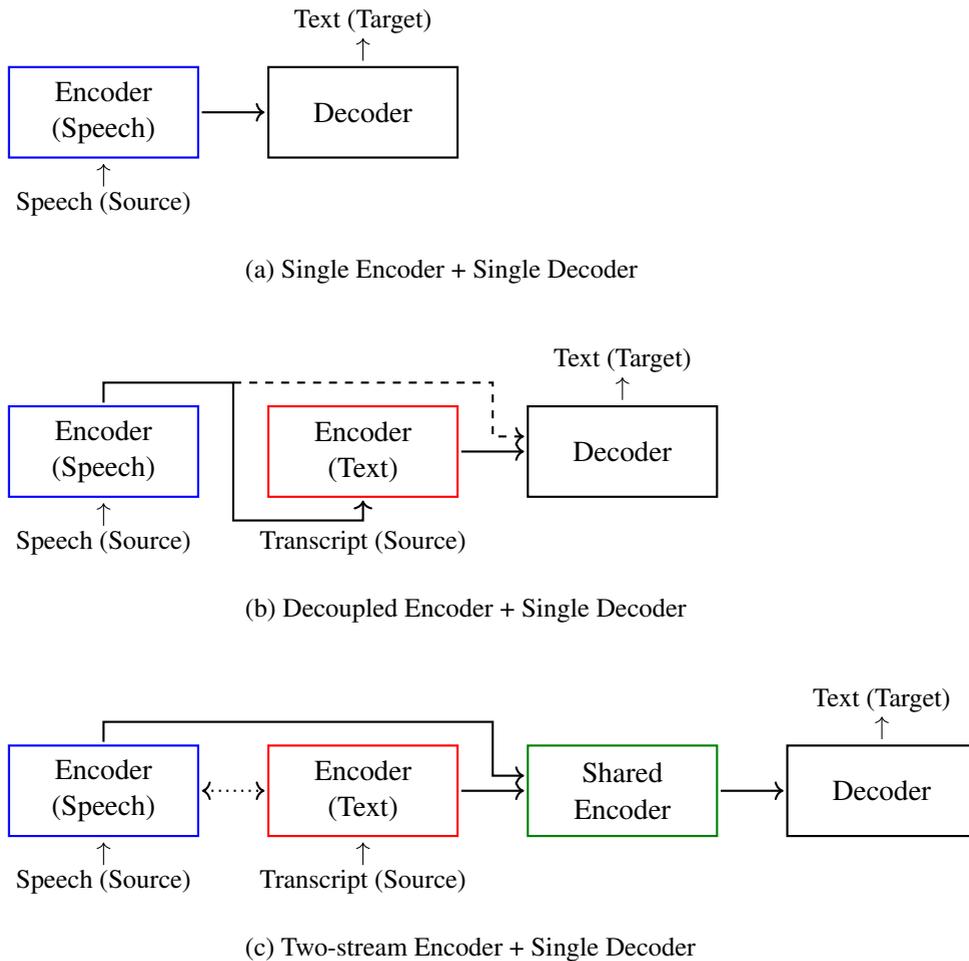


Figure 6.18: Architectures of speech-to-text translation models based on Transformers. In addition to the standard encoder-decoder architecture, we can explicitly model the acoustic and textual (semantic) information using two separate encoders, called the speech encoder and the text encoder. In the decoupled encoder architecture, the two encoders are stacked, that is, text encoding is a subsequent process after speech encoding. In the two-stream encoder architecture, the two encoders work in parallel, and their outputs are merged using an additional encoder, called the shared encoder. The dotted line indicates the potential for interaction between the two encoders. For example, we could define a loss function to minimize the difference between their outputs, thereby guiding the model towards more aligned representations.

translation for more information [Xu et al., 2023a].

#### 6.5.4 Vision Models

While Transformers were first used in NLP, their application to other domains has been a prominent research topic. In computer vision, for instance, there is a notable trend of shifting from CNNs to Transformers as the backbone models. In this subsection, we consider **Vision Transformer (ViT)** - an interesting application of Transformers to image classification

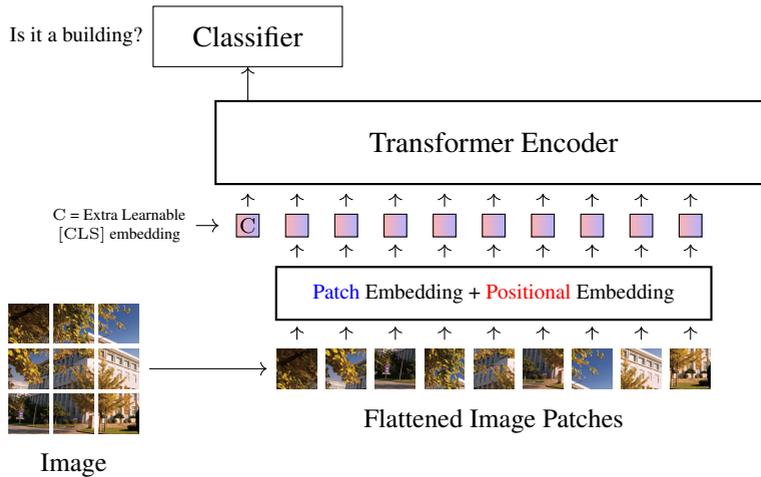


Figure 6.19: Illustration of Vision Transformer for image classification [Dosovitskiy et al., 2021]. There are three steps. In the first step, the input image is segmented into patches, which are then flattened and mapped into embeddings. In the second step, a Transformer encoder is employed to process the sequence of embeddings, representing the image as a real-valued vector (e.g., the output of the encoder at the first position). In the last step, a classifier is built on top of this image representation.

[Dosovitskiy et al., 2021]. Vision Transformer is a milestone model which opens the door to purely Transformer-based vision models. Here we consider the basic structure of Vision transformer to make this section concentrated and coherent, although there has been an extensive literature on Vision transformer and its variants. More detailed discussions of vision transformer can be found in recent surveys [Han et al., 2022; Liu et al., 2023e].

The core idea behind Vision Transformer is to transform an image into a sequence of visual tokens, and input this sequence into a Transformer encoder to generate a representation of the image. The Transformer encoder is standard, and so we will not discuss it here, given the introduction to Transformers we have presented so far in this chapter. Mapping a 2D image into a sequence of tokens needs some additional work. Suppose we have an image represented as an  $H \times W \times C$  feature map, where  $H$  is the height of the image,  $W$  is the width of the image, and  $C$  is the number of channels. The first step is to segment this image into a number of **patches**. Suppose all patches are squares of side length  $P$ . Then the resulting patches can be represented by feature maps of shape  $P \times P \times C$ . By ordering these patches in some way, we obtain a sequence of  $\frac{HW}{P^2}$  patches, with each patch being treated as a “token”.

Given this patch sequence, the subsequent steps are straightforward. For the patch at each position, we obtain a  $d$ -dimensional embedding by a linear transformation of the input feature map. The input of the Transformer encoder is a sequence of  $d$ -dimensional vectors, each of which is the sum of the corresponding patch and positional embeddings. Figure 6.19 illustrates the patching and embedding steps in Vision Transformer.

Once we have a sequence of vectors for representing the image, we can employ the Transformer encoder to encode the sequence. The encoding process is exactly the same as that

in text encoding as discussed in Section 6.5.2. For classification problems, we need only a single representation of the input. It is convenient to take the output of the encoder at position 0 (denoted by  $\mathbf{h}_0^L$ ) and feed it into a classifier. Given that the first token [CLS] serves as a special token that would be attended to by all other tokens,  $\mathbf{h}_0^L$  provides an unbiased representation of the entire sequence.

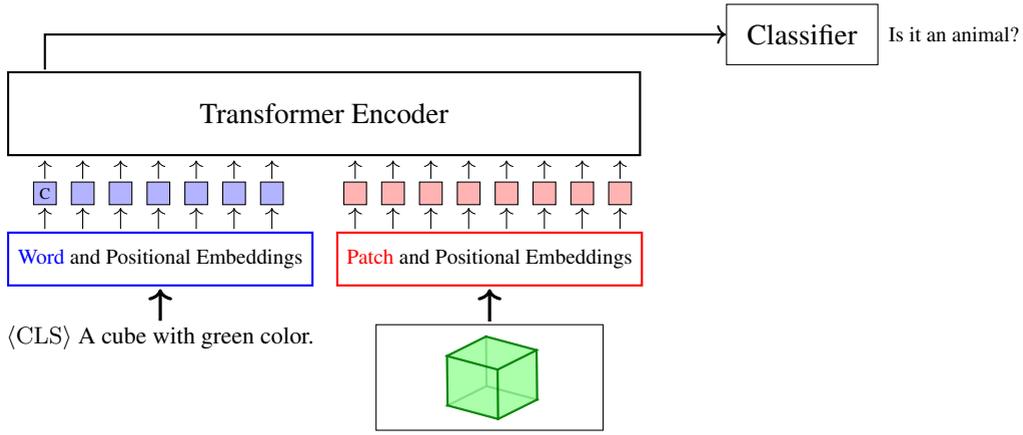
Typically, a standard way to train Vision Transformer is to minimize some loss on labeled data, such as ImageNet. More recently, inspired by self-supervised learning in BERT-like models, there have been successful attempts to train Transformer-based image encoders on large-scale unlabeled data [Caron et al., 2021; Bao et al., 2021; He et al., 2022]. Note that one of the most significant contributions of Vision Transformer is that it unifies the representation models for different modalities. This suggests that if an object, whether an image or text, is represented as a sequence of embeddings, it can be easily modeled using the Transformer architecture.

### 6.5.5 Multimodal Models

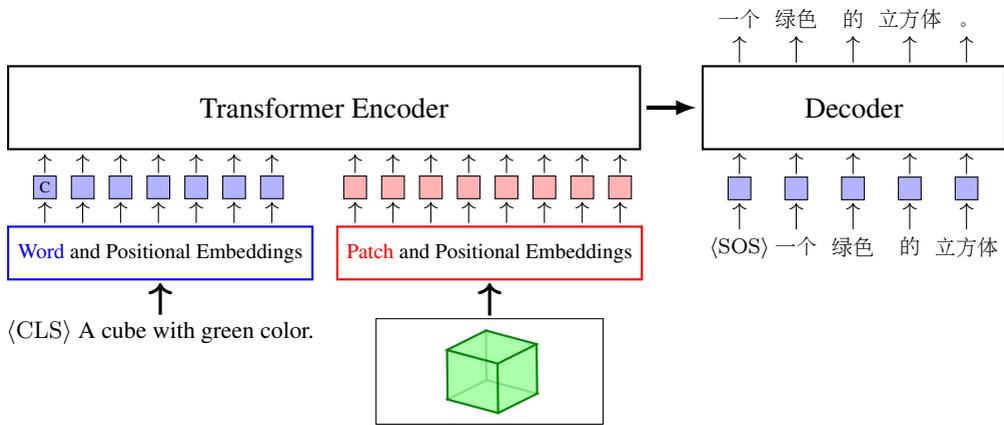
The above discussion of Vision Transformer offers the possibility of unifying the representations from multiple modalities using the same Transformer architecture. In fact, many recent multimodal systems draw inspiration largely from Transformers [Xu et al., 2023c]. Such systems convert objects from different modalities into vector sequences and feed these vectors into a single Transformer model. The output is a fused representation of all inputs, which can then be used in downstream systems.

As a simple example, consider the task of encoding a pair consisting of text and its corresponding image. First, we represent both the text and the image as sequences of embeddings that have the same dimensionality. This is a common step in sequence modeling, which we have confronted many times so far. We can do this by using either a simple embedding model (e.g., a word or patch embedding model) or a well-trained sequence model (e.g., a vision model). Then, these two sequences are concatenated into a long sequence involving both textual and visual embeddings. The follow-on step is standard: a Transformer encoder takes the concatenated sequence of embeddings as input and produces representations of the text and image as output. Note that concatenating textual and visual sequences is one of the simplest methods for vision-text modeling. There are several alternative ways to merge information from different modalities, for example, we can feed visual representations into the attention layers of a text encoder or decoder [Li et al., 2022e; Alayrac et al., 2022].

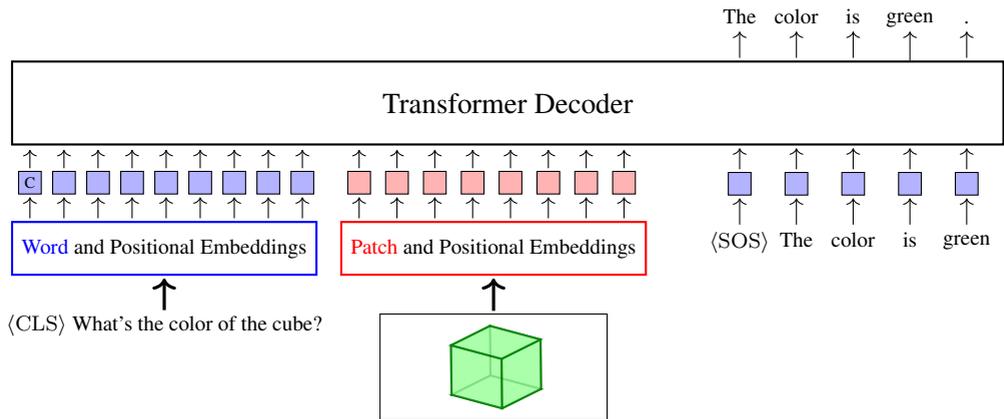
The above multimodal encoder can be used in both encoder-only and encoder-decoder systems. For encoder-only systems, consider an example where, given an image and a description of it, we predict the class of the image using a classifier built on top of the encoder [Kim et al., 2021]. For encoder-decoder systems, we pair the encoder with a decoder, as in sequence-to-sequence modeling [Cho et al., 2021]. For example, we might employ a Transformer decoder to generate text based on the output of the encoder. A common application of this architecture is **visual question answering (VQA)**, where an image and a question about the image are provided, and the system is tasked with generating an answer [Antol et al., 2015]. The architectures of these models are illustrated in Figure 6.20 (a-b).



(a) Multi-modal Encoder + Classifier



(b) Multi-modal Encoder + Text Decoder (Translation)



(c) Multi-modal Decoder (Language Modeling)

Figure 6.20: Vision-text models. Blue boxes represent word+position embeddings, and red boxes represent image patch+position embeddings.

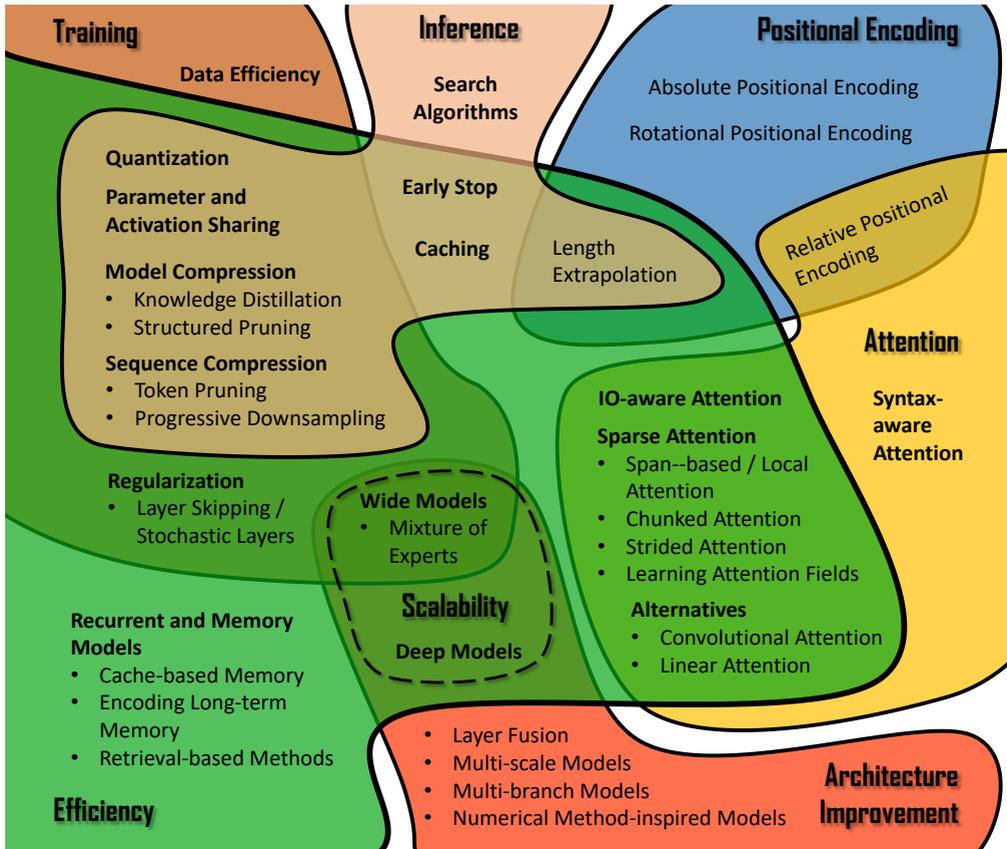
More recently, NLP has seen new advances by using large language models to deal with both textual and other forms of data, such as images, videos, and audio, leading to new breakthroughs in multimodal processing [Liu et al., 2023a; Yin et al., 2023]. By representing all inputs as a sequence of token embeddings, the problem will be simple: we predict the next token given its context. This can be done by using decoder-only systems, as shown in Figure 6.20 (c).

## 6.6 Summary

Transformer models have achieved widespread use over the past few years since the concept of *Transformer* was proposed by Vaswani et al. [2017]. This has accelerated the development of these models, leading to a great variety of new algorithms, systems and concepts. A thorough discussion of Transformers requires a broad scope, and so it is impossible to cover every problem and to provide a complete list of the corresponding references. While this chapter has presented a detailed introduction to Transformers, there are still topics that we did not mention, such as the theoretical aspects of these models. Figure 6.21 shows an overview of Transformer models, where we attempt to give a big picture. Note that these models and related techniques can be classified in many different ways, and we just show one of them. To summarize, we would like to highlight the following points.

- **Foundations of Transformers.** Although the impact of Transformers has been revolutionary, they are not completely "new" models. From a deep learning perspective, Transformers are composed of common building blocks, including word and positional embeddings [Bengio et al., 2003a; Mikolov et al., 2013c; Gehring et al., 2017b], attention mechanisms [Bahdanau et al., 2014; Luong et al., 2015], residual connections [He et al., 2016b], layer-normalization [Ba et al., 2016], and so on. Many of these components were presented in earlier systems, for example, similar ideas with QKV attention can be found in memory networks [Sukhbaatar et al., 2015] and hierarchical attention networks [Yang et al., 2016]. Transformers offer a novel approach to integrating these components, resulting in a unique architecture. For example, in Transformers, the combination of multi-head attention and dot-product QKV attention, along with the incorporation of layer-normalization and residual connections, gives rise to a distinctive neural network block, specifically a self-attention sub-layer. This design has since become a de facto standard in many follow-on sequence modeling systems.
- **Attention Models.** The success of Transformers on NLP tasks has largely been attributed to the use of multi-head self-attention for sequence modeling. This has led to a surge of interest in enhancing the attention mechanisms within Transformers. While it is impossible to detail every attention model, there are several notable research directions. One prominent direction involves modifying the forms of QKV attention and multi-head attention for improved performance. The scope of this direction is vast, as there are numerous aspects to consider when enhancing Transformers [Lin et al., 2022a]. For example, one may add new components to self-attention sub-layers to adapt them

<p><b>Applications</b></p> <p><b>Self-supervised Transformers in NLP</b></p> <p><b>Natural Language Understanding:</b> BERT, RoBERTa, ALBERT, SpanBERT, ERNIE, XLM, XLNet, etc.</p> <p><b>Natural Language Generation:</b> GPT Series, T5, BART, MASS, PaLM, LaMDA, Megatron-Turing NLG, BLOOM, LLaMA, etc.</p>		<p><b>Supervised Transformers in NLP</b> Neural Machine Translation, Summarization, Sentiment Analysis, Question Answering, Named Entity Recognition, Syntactic Analysis, etc.</p>	<p><b>Vision</b> ViT, Swin, MAE, BEiT, DETR, iGPT, etc.</p> <p><b>Speech</b> Wav2vec 2.0, Whisper, HuBERT, Conformer, etc.</p>
		<p><b>Multi-modal</b> CLIP, ViLBERT, VisualBERT, VL-BERT, UNITER, LXMERT, ViLT, VLTS, Data2vec, etc.</p>	<p><b>Bioinformatics</b>   <b>Robotics</b></p> <p><b>Recommendation Systems</b></p> <p><b>Time Series Analysis</b></p>



**Foundations of Transformers**

<p><b>Transformer Sub-models</b></p> <ul style="list-style-type: none"> <li>Word Encoding and Positional Encoding</li> <li>Multi-head Self-attention</li> <li>Feed-forward Networks</li> <li>Layer Normalization and Residual Connections</li> </ul>	<p><b>Architectures</b></p> <ul style="list-style-type: none"> <li>Encoder Only</li> <li>Decoder Only</li> <li>Encoder-Decoder</li> </ul>	<p><b>Theoretical Analysis</b></p> <ul style="list-style-type: none"> <li>Linguistics</li> <li>Machine Learning</li> <li>Formal Systems</li> </ul>
--	---	--

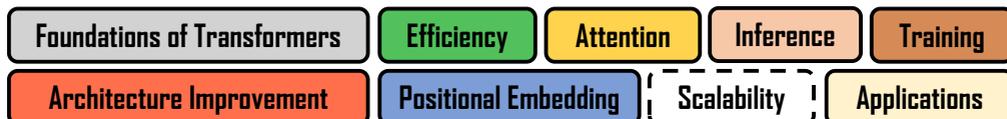


Figure 6.21: An overview of Transformers.

to specific tasks, resulting in various Transformer variants. A second direction is to incorporate prior knowledge into the design of attention models. This makes sense, because much of the emphasis in traditional NLP has been on using linguistic insights to guide system design, and we generally want NLP systems to be linguistically explainable. For example, many Transformer-based systems take syntactic parses as input in various forms and make use of syntax in sequence modeling. A third direction is to develop efficient attention models [Tay et al., 2020b]. Self-attention has long been criticized for its quadratic time complexity and dependency on all previous tokens for each new token. In response, many researchers have focused on simplifying the structure of self-attention, or on approximating it using sparse or recurrent models. This concern for efficiency also motivates the development of alternatives to self-attention, such as attention models with linear time complexity. In addition to exploring stronger and more efficient attention models, it is natural to examine what knowledge is learned by such models. Interestingly, researchers have found that the underlying structure of languages can be learned by multi-head self-attention models, although these models are not trained to represent such knowledge [Manning et al., 2020].

- **Word and Positional Embeddings.** Transformers represent each input word as a word embedding, along with its positional embedding. Learning these word embeddings is not a specific problem for Transformers. We can either resort to well-trained word embeddings, such as the Word2Vec or GloVe embeddings, or treat them as learnable parameters of Transformers. A related issue is tokenization of the input sequences. In general, tokenization impacts the number of resulting tokens and the difficulty of learning the corresponding embeddings. In many applications, therefore, one needs to carefully choose a tokenization method. Furthermore, positional embedding plays an important role in Transformers, as the attention mechanisms are order-insensitive by design [Dufter et al., 2022]. Although positional embedding is a general problem, much of the research is focused on improving Transformers, leading to modifications to Transformer models [Shaw et al., 2018; Huang et al., 2018]. Additionally, studies show that, when we deal with sequences that are much longer than those in training data, extrapolation can be achieved by replacing sinusoidal positional embeddings with rotary positional embeddings or simply scaling attention weights with a positional scalar [Raffel et al., 2020; Su et al., 2021; Press et al., 2021].
- **Training and Model Scaling.** In the era of deep learning, powerful systems are typically obtained by using large neural networks. A simple approach to increasing the model capacity of Transformers is to stack more layers and/or enlarge the size of each representation. We can see many cases where deep and wide Transformer models consistently outperform small models. However, challenges arise when we attempt to train extremely large Transformer models, especially when gradient descent is applied over vast amounts of data, demanding substantial computational resources. An engineering solution is to distribute the training across a cluster of computers [Lepikhin et al., 2021; Chowdhery et al., 2022]. While distributed training is a very general

method and is not restricted to Transformers, it indeed influences the design of model architectures, for example, sparse expert models can ease the training with distributed parameters, serving as the foundation for many expansive Transformer-based systems. Scaling up the training of Transformers allows us to study the scaling law of large neural networks: how model performance relates to model size, training data size, and training cost [Hestness et al., 2017; Kaplan et al., 2020]. This is sometimes accompanied by an interesting behavior, known as emergence [Wei et al., 2022b]. In recent NLP research, the acquisition of emergent abilities has been considered one of the prerequisites for developing strong language models.

- **Efficient Models.** There are different goals for efficiency. For example, one may wish a system to be memory efficient when the problem is memory bound, or one may wish it to be speed efficient when latency is an important consideration. In general, we need to seek a balance between these goals, resulting in different efficiency optimizations. In the context of Transformers, many of these optimizations are achieved by modifying the attention models, as mentioned above. For example, several variants of the self-attention models are proposed to reduce the memory footprint when processing long sequences [Tay et al., 2020b]. Similarly, other variants aim to reduce computation and thus give lower latency. Furthermore, being a type of neural network, Transformers can be optimized in ways independent of model architectures. Typical methods include but are not limited to conditional computation, knowledge distillation, structured pruning, and sequence compression. Efficiency optimizations can also be considered from the perspective of computer architecture [Kim et al., 2023]. For example, when applying Transformers to sequence-to-sequence problems, the encoding and decoding processes are generally compute-intensive and IO-intensive, respectively. Therefore, we can employ different optimization methods for different components of Transformers.
- **Inference.** The inference problem is commonly discussed in sequence generation. In NLP, we often need to find the “best” hypothesis in a space involving sequences of tens or even hundreds of tokens over a vocabulary. Considering this an instance of the search problem in artificial intelligence, many algorithms can be applied, such as breadth-first search, depth-first search and A\* search. In many practical applications of NLP, the efficiency of the search systems is an important consideration. As a result, optimized search algorithms are required. Most of these algorithms have been explored in machine translation and ASR, and are directly applicable to neural text generation models like Transformer. There are also optimizations of conventional decoding methods tailored to Transformers [Leviathan et al., 2023]. Moreover, the above-mentioned efficient approaches, such as the efficient attention models, are also in widespread use, with many successful examples in deploying neural machine translation systems and large language models [Heafield et al., 2021; Dao et al., 2023].
- **Applications.** Applications of Transformers cover a wide variety of NLP problems. During the development of Transformers, they were at first used to build supervised models that perform particular tasks. Later, a greater success was achieved by using

them as backbone networks for large scale self-supervised learning of foundation models [Bommasani et al., 2021]. This markedly changed the paradigm in NLP. We need only pre-train a model to obtain general knowledge of languages on huge amounts of text. Then, we adapt this model to downstream tasks using methods with little effort, such as fine-tuning or prompting. Over the past few years, we have also seen an explosion of applications for Transformers in fields other than NLP, such as computer vision, speech processing, and bioinformatics. The idea behind these applications is that we can represent any input data as a sequence of tokens and directly employ Transformers to model this sequence. This approach extends Transformers to general representation models across different modalities, making it easier to use Transformers for handling multi-modal data.

- **Large Language Models as Foundation Models.** Transformers form the basis of recent large language models, such as the GPT series, which show surprising breakthroughs in NLP, and even in **artificial general intelligence (AGI)** [Bubeck et al., 2023; Yang et al., 2023a]. Much of the research in large language models is more or less related to Transformers. For example, as discussed in Section 6.5.1, the problem of training these language models is the same as that of training Transformer decoders. And the modifications to Transformer decoders can be directly applied to large language models. On the other hand, the rapid development of large language models has also driven further improvements in various techniques for Transformers, such as efficient and low-cost adaptation of large Transformers to different tasks.
- **Theoretical Analysis.** Although Transformers have shown strong empirical results in various fields, their theoretical aspects have received relatively less attention compared to the extensive research on model improvement and engineering. This is not a specific problem for Transformers, but a common problem for the NLP and machine learning communities. In response, researchers have made attempts to analyze Transformers more deeply. One way is to view Transformers as deep neural networks and interpret them via mathematical tools. For example, the residual networks in Transformers are mathematically equivalent to the Euler solvers for ODEs. This equivalence suggests that we can leverage insights from numerical ODE methods to inform model design. Another promising avenue of research aims to develop a theoretical understanding of the self-attention mechanism, which distinguishes Transformers from other deep learning models. For example, there have been studies on interpreting self-attention and Transformers from machine learning perspectives, such as data compression [Yu et al., 2023a], optimization [Li et al., 2022c], and function approximation [Yun et al., 2019]. Moreover, Transformers can also be related to formal systems, including Turing machines [Pérez et al., 2018], counter machines [Bhattamishra et al., 2020], regular and context-free languages [Hahn, 2020], Boolean circuits [Hao et al., 2022; Merrill et al., 2022], programming languages [Weiss et al., 2021], first-order logic [Chiang et al., 2023a], and so on. These provide tools to study the expressivity of Transformers. It is, however, worth noting that, while we can understand Transformers in several different

ways, there are no general theories to explain the nature of these models. Perhaps this is a challenge for the field of machine learning, and many researchers are working on this issue. But it is indeed an important issue, as the development of the theories behind complex neural networks like Transformers can help develop systems with explainable and predictable behaviors.



# Large Language Models

<b>7</b>	<b>Pre-training</b> .....	<b>365</b>
7.1	Pre-training NLP Models	
7.2	Self-supervised Pre-training Tasks	
7.3	Example: BERT	
7.4	Applying BERT Models	
7.5	Summary	
<b>8</b>	<b>Generative Models</b> .....	<b>403</b>
8.1	A Brief Introduction to LLMs	
8.2	Training at Scale	
8.3	Long Sequence Modeling	
8.4	Summary	
<b>9</b>	<b>Prompting</b> .....	<b>467</b>
9.1	General Prompt Design	
9.2	Advanced Prompting Methods	
9.3	Learning to Prompt	
9.4	Summary	
<b>10</b>	<b>Alignment</b> .....	<b>533</b>
10.1	An Overview of LLM Alignment	
10.2	Instruction Alignment	
10.3	Human Preference Alignment: RLHF	
10.4	Improved Human Preference Alignment	
10.5	Summary	
<b>11</b>	<b>Inference</b> .....	<b>587</b>
11.1	Prefilling and Decoding	
11.2	Efficient Inference Techniques	
11.3	Inference-time Scaling	
11.4	Summary	



# Chapter 7

## Pre-training

The development of neural sequence models, such as Transformers, along with the improvements in large-scale self-supervised learning, has opened the door to universal language understanding and generation. This achievement is largely motivated by pre-training: we separate common components from many neural network-based systems, and then train them on huge amounts of unlabeled data using self-supervision. These pre-trained models serve as foundation models that can be easily adapted to different tasks via fine-tuning or prompting. As a result, the paradigm of NLP has been enormously changed. In many cases, large-scale supervised learning for specific tasks is no longer required, and instead, we only need to adapt pre-trained foundation models.

While pre-training has gained popularity in recent NLP research, this concept dates back decades to the early days of deep learning. For example, early attempts to pre-train deep learning systems include unsupervised learning for RNNs, deep feedforward networks, autoencoders, and others [Schmidhuber, 2015]. In the modern era of deep learning, we experienced a resurgence of pre-training, caused in part by the large-scale unsupervised learning of various word embedding models [Mikolov et al., 2013c; Pennington et al., 2014]. During the same period, pre-training also attracted significant interest in computer vision, where the backbone models were trained on relatively large labeled datasets such as ImageNet, and then applied to different downstream tasks [He et al., 2019; Zoph et al., 2020]. Large-scale research on pre-training in NLP began with the development of language models using self-supervised learning. This family of models covers several well-known examples like **BERT** [Devlin et al., 2019] and **GPT** [Brown et al., 2020], all with a similar idea that general language understanding and generation can be achieved by training the models to predict masked words in a huge amount of text. Despite the simple nature of this approach, the resulting models show remarkable capability in modeling linguistic structure, though they are not explicitly trained to achieve this. The generality of the pre-training tasks leads to systems that exhibit strong performance in a large variety of NLP problems, even outperforming previously well-developed supervised systems. More recently, pre-trained large language models have achieved greater success, showing the exciting prospects for more general artificial intelligence [Bubeck et al., 2023].

This chapter discusses the concept of pre-training in the context of NLP. It begins with a general introduction to pre-training methods and their applications. BERT is then used as an example to illustrate how a sequence model is trained via a self-supervised task, called **masked language modeling**. This is followed by a discussion of methods for adapting pre-trained sequence models for various NLP tasks. Note that in this chapter, we will focus primarily on the pre-training paradigm in NLP, and therefore, we do not intend to cover details about generative large language models. A detailed discussion of these models will be left to subsequent chapters.

## 7.1 Pre-training NLP Models

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation. While these problems have different forms, for simplicity, we describe them using a single model defined as follows:

$$\begin{aligned} \mathbf{o} &= g(x_0, x_1, \dots, x_m; \theta) \\ &= g_\theta(x_0, x_1, \dots, x_m) \end{aligned} \quad (7.1)$$

where  $\{x_0, x_1, \dots, x_m\}$  denotes a sequence of input tokens<sup>1</sup>,  $x_0$  denotes a special symbol ( $\langle s \rangle$  or [CLS]) attached to the beginning of a sequence,  $g(\cdot; \theta)$  (also written as  $g_\theta(\cdot)$ ) denotes a neural network with parameters  $\theta$ , and  $\mathbf{o}$  denotes the output of the neural network. Different problems can vary based on the form of the output  $\mathbf{o}$ . For example, in token prediction problems (as in language modeling),  $\mathbf{o}$  is a distribution over a vocabulary; in sequence encoding problems,  $\mathbf{o}$  is a representation of the input sequence, often expressed as a real-valued vector sequence.

There are two fundamental issues here.

- Optimizing  $\theta$  on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.
- Applying the pre-trained model  $g_\theta(\cdot)$  to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters  $\hat{\theta}$  slightly using labeled data or prompt the model with task descriptions.

In this section, we discuss the basic ideas in addressing these issues.

### 7.1.1 Unsupervised, Supervised and Self-supervised Pre-training

In deep learning, pre-training refers to the process of optimizing a neural network before it is further trained/tuned and applied to the tasks of interest. This approach is based on an assumption that a model pre-trained on one task can be adapted to perform another task. As a result, we do not need to train a deep, complex neural network from scratch on tasks with

---

<sup>1</sup>Here we assume that tokens are basic units of text that are separated through tokenization. Sometimes, we will use the terms *token* and *word* interchangeably, though they have closely related but slightly different meanings in NLP.

limited labeled data. Instead, we can make use of tasks where supervision signals are easier to obtain. This reduces the reliance on task-specific labeled data, enabling the development of more general models that are not confined to particular problems.

During the resurgence of neural networks through deep learning, many early attempts to achieve pre-training were focused on **unsupervised learning**. In these methods, the parameters of a neural network are optimized using a criterion that is not directly related to specific tasks. For example, we can minimize the reconstruction cross-entropy of the input vector for each layer [Bengio et al., 2006]. Unsupervised pre-training is commonly employed as a preliminary step before supervised learning, offering several advantages, such as aiding in the discovery of better local minima and adding a regularization effect to the training process [Erhan et al., 2010]. These benefits make the subsequent supervised learning phase easier and more stable.

A second approach to pre-training is to pre-train a neural network on **supervised learning** tasks. For example, consider a sequence model designed to encode input sequences into some representations. In pre-training, this model is combined with a classification layer to form a classification system. This system is then trained on a pre-training task, such as classifying sentences based on sentiment (e.g., determining if a sentence conveys a positive or negative sentiment). Then, we adapt the sequence model to a downstream task. We build a new classification system based on this pre-trained sequence model and a new classification layer (e.g., determining if a sequence is subjective or objective). Typically, we need to fine-tune the parameters of the new model using task-specific labeled data, ensuring the model is optimally adjusted to perform well on this new type of data. The fine-tuned model is then employed to classify new sequences for this task. An advantage of supervised pre-training is that the training process, either in the pre-training or fine-tuning phase, is straightforward, as it follows the well-studied general paradigm of supervised learning in machine learning. However, as the complexity of the neural network increases, the demand for more labeled data also grows. This, in turn, makes the pre-training task more difficult, especially when large-scale labeled data is not available.

A third approach to pre-training is **self-supervised learning**. In this approach, a neural network is trained using the supervision signals generated by itself, rather than those provided by humans. This is generally done by constructing its own training tasks directly from unlabeled data, such as having the system create pseudo labels. While self-supervised learning has recently emerged as a very popular method in NLP, it is not a new concept. In machine learning, a related concept is **self-training** where a model is iteratively improved by learning from the pseudo labels assigned to a dataset. To do this, we need some seed data to build an initial model. This model then generates pseudo labels for unlabeled data, and these pseudo labels are subsequently used to iteratively refine and bootstrap the model itself. Such a method has been successfully used in several NLP areas, such as word sense disambiguation [Yarowsky, 1995] and document classification [Blum and Mitchell, 1998]. Unlike the standard self-training method, self-supervised pre-training in NLP does not rely on an initial model for annotating the data. Instead, all the supervision signals are created from the text, and the entire model is trained from scratch. A well-known example of this is training sequence models by successively predicting a masked word given its preceding or surrounding words in a text. This

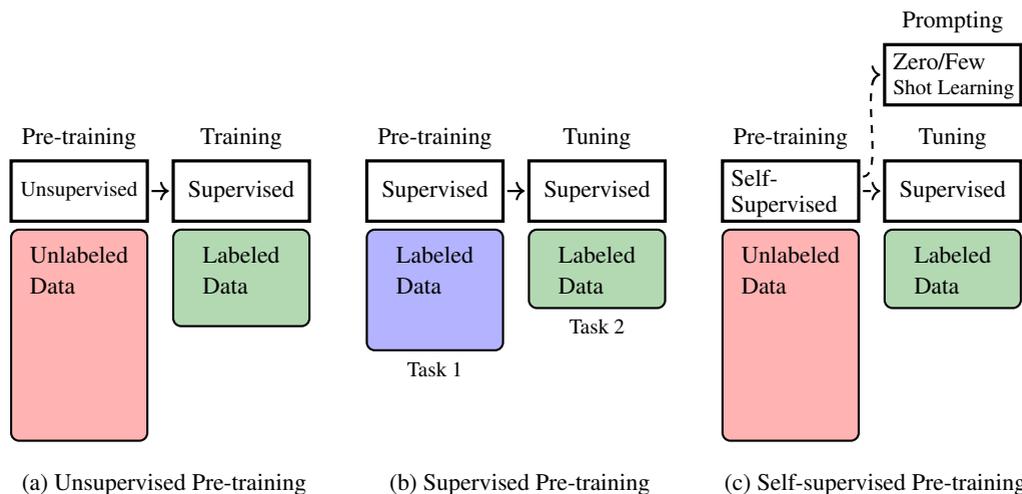


Figure 7.1: Illustration of unsupervised, supervised, and self-supervised pre-training. In unsupervised pre-training, the pre-training is performed on large-scale unlabeled data. It can be viewed as a preliminary step to have a good starting point for the subsequent optimization process, though considerable effort is still required to further train the model with labeled data after pre-training. In supervised pre-training, the underlying assumption is that different (supervised) learning tasks are related. So we can first train the model on one task, and transfer the resulting model to another task with some training or tuning effort. In self-supervised pre-training, a model is pre-trained on large-scale unlabeled data via self-supervision. The model can be well trained in this way, and we can efficiently adapt it to new tasks through fine-tuning or prompting.

enables large-scale self-supervised learning for deep neural networks, leading to the success of pre-training in many understanding, writing, and reasoning tasks.

Figure 7.1 shows a comparison of the above three pre-training approaches. Self-supervised pre-training is so successful that most current state-of-the-art NLP models are based on this paradigm. Therefore, in this chapter and throughout this book, we will focus on self-supervised pre-training. We will show how sequence models are pre-trained via self-supervision and how the pre-trained models are applied.

## 7.1.2 Adapting Pre-trained Models

As mentioned above, two major types of models are widely used in NLP pre-training.

- **Sequence Encoding Models.** Given a sequence of words or tokens, a sequence encoding model represents this sequence as either a real-valued vector or a sequence of vectors, and obtains a representation of the sequence. This representation is typically used as input to another model, such as a sentence classification system.
- **Sequence Generation Models.** In NLP, sequence generation generally refers to the problem of generating a sequence of tokens based on a given context. The term *context* has different meanings across applications. For example, it refers to the preceding

tokens in language modeling, and refers to the source-language sequence in machine translation<sup>2</sup>.

We need different techniques for applying these models to downstream tasks after pre-training. Here we are interested in the following two methods.

### 1. Fine-tuning of Pre-trained Models

For sequence encoding pre-training, a common method of adapting pre-trained models is fine-tuning. Let  $\text{Encode}_\theta(\cdot)$  denote an encoder with parameters  $\theta$ , for example,  $\text{Encode}_\theta(\cdot)$  can be a standard Transformer encoder. Provided we have pre-trained this model in some way and obtained the optimal parameters  $\hat{\theta}$ , we can employ it to model any sequence and generate the corresponding representation, like this

$$\mathbf{H} = \text{Encode}_{\hat{\theta}}(\mathbf{x}) \quad (7.2)$$

where  $\mathbf{x}$  is the input sequence  $\{x_0, x_1, \dots, x_m\}$ , and  $\mathbf{H}$  is the output representation which is a sequence of real-valued vectors  $\{\mathbf{h}_0, \mathbf{h}_1, \dots, \mathbf{h}_m\}$ . Because the encoder does not work as a standalone NLP system, it is often integrated as a component into a bigger system. Consider, for example, a text classification problem in which we identify the polarity (i.e., positive, negative, and neutral) of a given text. We can build a text classification system by stacking a classifier on top of the encoder. Let  $\text{Classify}_\omega(\cdot)$  be a neural network with parameters  $\omega$ . Then, the text classification model can be expressed in the form

$$\begin{aligned} \text{Pr}_{\omega, \hat{\theta}}(\cdot | \mathbf{x}) &= \text{Classify}_\omega(\mathbf{H}) \\ &= \text{Classify}_\omega(\text{Encode}_{\hat{\theta}}(\mathbf{x})) \end{aligned} \quad (7.3)$$

Here  $\text{Pr}_{\omega, \hat{\theta}}(\cdot | \mathbf{x})$  is a probability distribution over the label set  $\{\text{positive}, \text{negative}, \text{neutral}\}$ , and the label with the highest probability in this distribution is selected as output. To keep the notation uncluttered, we will use  $F_{\omega, \hat{\theta}}(\cdot)$  to denote  $\text{Classify}_\omega(\text{Encode}_{\hat{\theta}}(\cdot))$ .

Because the model parameters  $\omega$  and  $\hat{\theta}$  are not optimized for the classification task, we cannot directly use this model. Instead, we must use a modified version of the model that is adapted to the task. A typical way is to fine-tune the model by giving explicit labeling in downstream tasks. We can train  $F_{\omega, \hat{\theta}}(\cdot)$  on a labeled dataset, treating it as a common supervised learning task. The outcome of the fine-tuning is the parameters  $\tilde{\omega}$  and  $\tilde{\theta}$  that are further optimized. Alternatively, we can freeze the encoder parameters  $\hat{\theta}$  to maintain their pre-trained state, and focus solely on optimizing  $\omega$ . This allows the classifier to be efficiently adapted to work in tandem with the pre-trained encoder.

Once we have obtained a fine-tuned model, we can use it to classify a new text. For example, suppose we have a comment posted on a travel website:

I love the food here. It's amazing!

<sup>2</sup>More precisely, in auto-regressive decoding of machine translation, each target-language token is generated based on both its preceding tokens and source-language sequence.

We first tokenize this text into tokens<sup>3</sup>, and then feed the token sequence  $\mathbf{x}_{\text{new}}$  into the fine-tuned model  $F_{\tilde{\omega}, \tilde{\theta}}(\cdot)$ . The model generates a distribution over classes by

$$F_{\tilde{\omega}, \tilde{\theta}}(\mathbf{x}_{\text{new}}) = \left[ \Pr(\text{positive}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{negative}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{neutral}|\mathbf{x}_{\text{new}}) \right] \quad (7.4)$$

And we select the label of the entry with the maximum value as output. In this example it is positive.

In general, the amount of labeled data used in fine-tuning is small compared to that of the pre-training data, and so fine-tuning is less computationally expensive. This makes the adaptation of pre-trained models very efficient in practice: given a pre-trained model and a downstream task, we just need to collect some labeled data, and slightly adjust the model parameters on this data. A more detailed discussion of fine-tuning can be found in Section 7.4.

## 2. Prompting of Pre-trained Models

Unlike sequence encoding models, sequence generation models are often employed independently to address language generation problems, such as question answering and machine translation, without the need for additional modules. It is therefore straightforward to fine-tune these models as complete systems on downstream tasks. For example, we can fine-tune a pre-trained encoder-decoder multilingual model on some bilingual data to improve its performance on a specific translation task.

Among various sequence generation models, a notable example is the large language models trained on very large amounts of data. These language models are trained to simply predict the next token given its preceding tokens. Although token prediction is such a simple task that it has long been restricted to “language modeling” only, it has been found to enable the learning of the general knowledge of languages by repeating the task a large number of times. The result is that the pre-trained large language models exhibit remarkably good abilities in token prediction, making it possible to transform numerous NLP problems into simple text generation problems through prompting the large language models. For example, we can frame the above text classification problem as a text generation task

I love the food here. It’s amazing! I’m \_\_\_\_\_

Here `__` indicates the word or phrase we want to predict (call it the **completion**). If the predicted word is *happy*, or *glad*, or *satisfied* or a related positive word, we can classify the text as positive. This example shows a simple prompting method in which we concatenate the input text with *I’m* to form a prompt. Then, the completion helps decide which label is assigned to the original text.

Given the strong performance of language understanding and generation of large language models, a prompt can instruct the models to perform more complex tasks. Here is a prompt where we prompt the LLM to perform polarity classification with an instruction.

<sup>3</sup>The text can be tokenized in many different ways. One of the simplest is to segment the text into English words and punctuations {I, love, the, food, here, ., It, ’s, amazing, !}

Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.

**Input:** I love the food here. It's amazing!

**Polarity:** \_\_\_\_\_

The first two sentences are a description of the task. **Input** and **Polarity** are indicators of the input and output, respectively. We expect the model to complete the text and at the same time give the correct polarity label. By using instruction-based prompts, we can adapt large language models to solve NLP problems without the need for additional training.

This example also demonstrates the zero-shot learning capability of large language models, which can perform tasks that were not observed during the training phase. Another method for enabling new capabilities in a neural network is few-shot learning. This is typically achieved through **in-context learning (ICT)**. More specifically, we add some samples that demonstrate how an input corresponds to an output. These samples, known as **demonstrations**, are used to teach large language models how to perform the task. Below is an example involving demonstrations

Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.

**Input:** The traffic is terrible during rush hours, making it difficult to reach the airport on time.

**Polarity:** Negative

**Input:** The weather here is wonderful.

**Polarity:** Positive

**Input:** I love the food here. It's amazing!

**Polarity:** \_\_\_\_\_

Prompting and in-context learning play important roles in the recent rise of large language models. We will discuss these issues more deeply in Chapter 9. However, it is worth noting that while prompting is a powerful way to adapt large language models, some tuning efforts are still needed to ensure the models can follow instructions accurately. Additionally, the fine-tuning process is crucial for aligning the values of these models with human values. More detailed discussions of fine-tuning can be found in Chapter 10.

## 7.2 Self-supervised Pre-training Tasks

In this section, we consider self-supervised pre-training approaches for different neural architectures, including decoder-only, encoder-only, and encoder-decoder architectures. We restrict our discussion to Transformers since they form the basis of most pre-trained models in NLP. However, pre-training is a broad concept, and so we just give a brief introduction to basic approaches in order to make this section concise.

### 7.2.1 Decoder-only Pre-training

The decoder-only architecture has been widely used in developing language models [Radford et al., 2018]. For example, we can use a Transformer decoder as a language model by simply removing cross-attention sub-layers from it. Such a model predicts the distribution of tokens at a position given its preceding tokens, and the output is the token with the maximum probability. The standard way to train this model, as in the language modeling problem, is to minimize a loss function over a collection of token sequences. Let  $\text{Decoder}_\theta(\cdot)$  denote a decoder with parameters  $\theta$ . At each position  $i$ , the decoder generates a distribution of the next tokens based on its preceding tokens  $\{x_0, \dots, x_i\}$ , denoted by  $\text{Pr}_\theta(\cdot|x_0, \dots, x_i)$  (or  $\mathbf{p}_{i+1}^\theta$  for short). Suppose we have the gold-standard distribution at the same position, denoted by  $\mathbf{p}_{i+1}^{\text{gold}}$ . For language modeling, we can think of  $\mathbf{p}_{i+1}^{\text{gold}}$  as a one-hot representation of the correct predicted word. We then define a loss function  $\mathcal{L}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}})$  to measure the difference between the model prediction and the true prediction. In NLP, the log-scale cross-entropy loss is typically used.

Given a sequence of  $m$  tokens  $\{x_0, \dots, x_m\}$ , the loss on this sequence is the sum of the loss over the positions  $\{0, \dots, m-1\}$ , given by

$$\begin{aligned} \text{Loss}_\theta(x_0, \dots, x_m) &= \sum_{i=0}^{m-1} \mathcal{L}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \\ &= \sum_{i=0}^{m-1} \text{LogCrossEntropy}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \end{aligned} \quad (7.5)$$

where  $\text{LogCrossEntropy}(\cdot)$  is the log-scale cross-entropy, and  $\mathbf{p}_{i+1}^{\text{gold}}$  is the one-hot representation of  $x_{i+1}$ .

This loss function can be extended to a set of sequences  $\mathcal{D}$ . In this case, the objective of pre-training is to find the best parameters that minimize the loss on  $\mathcal{D}$

$$\hat{\theta} = \arg \min_{\theta} \sum_{\mathbf{x} \in \mathcal{D}} \text{Loss}_\theta(\mathbf{x}) \quad (7.6)$$

Note that this objective is mathematically equivalent to maximum likelihood estimation, and

can be re-expressed as

$$\begin{aligned}\hat{\theta} &= \arg \max_{\theta} \sum_{\mathbf{x} \in \mathcal{D}} \log \Pr_{\theta}(\mathbf{x}) \\ &= \arg \max_{\theta} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=0}^{i-1} \log \Pr_{\theta}(x_{i+1} | x_0, \dots, x_i)\end{aligned}\quad (7.7)$$

With these optimized parameters  $\hat{\theta}$ , we can use the pre-trained language model Decoder $_{\hat{\theta}}(\cdot)$  to compute the probability  $\Pr_{\hat{\theta}}(x_{i+1} | x_0, \dots, x_i)$  at each position of a given sequence.

## 7.2.2 Encoder-only Pre-training

As defined in Section 7.1.2, an encoder  $\text{Encoder}_{\theta}(\cdot)$  is a function that reads a sequence of tokens  $\mathbf{x} = x_0 \dots x_m$  and produces a sequence of vectors  $\mathbf{H} = \mathbf{h}_0 \dots \mathbf{h}_m$ <sup>4</sup>. Training this model is not straightforward, as we do not have gold-standard data for measuring how good the output of the real-valued function is. A typical approach to encoder pre-training is to combine the encoder with some output layers to receive supervision signals that are easier to obtain. Figure 7.2 shows a common architecture for pre-training Transformer encoders, where we add a Softmax layer on top of the Transformer encoder. Clearly, this architecture is the same as that of the decoder-based language model, and the output is a sequence of probability distributions

$$\begin{bmatrix} \mathbf{p}_1^{\mathbf{W}, \theta} \\ \vdots \\ \mathbf{p}_m^{\mathbf{W}, \theta} \end{bmatrix} = \text{Softmax}_{\mathbf{W}}(\text{Encoder}_{\theta}(\mathbf{x}))\quad (7.9)$$

Here  $\mathbf{p}_i^{\mathbf{W}, \theta}$  is the output distribution  $\Pr(\cdot | \mathbf{x})$  at position  $i$ . We use  $\text{Softmax}_{\mathbf{W}}(\cdot)$  to denote that the Softmax layer is parameterized by  $\mathbf{W}$ , that is,  $\text{Softmax}_{\mathbf{W}}(\mathbf{H}) = \text{Softmax}(\mathbf{H} \cdot \mathbf{W})$ . For notation simplicity, we will sometimes drop the superscripts  $\mathbf{W}$  and  $\theta$  affixed to each probability distribution.

The difference between this model and standard language models is that the output  $\mathbf{p}_i$  has different meanings in encoder pre-training and language modeling. In language modeling,  $\mathbf{p}_i$  is the probability distribution of predicting the next word. This follows an auto-regressive decoding process: a language model only observes the words up to position  $i$  and predicts the next. By contrast, in encoder pre-training, the entire sequence can be observed at once, and so it makes no sense to predict any of the tokens in this sequence.

<sup>4</sup>If we view  $\mathbf{h}_i$  as a row vector,  $\mathbf{H}$  can be written as

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_m \end{bmatrix}\quad (7.8)$$

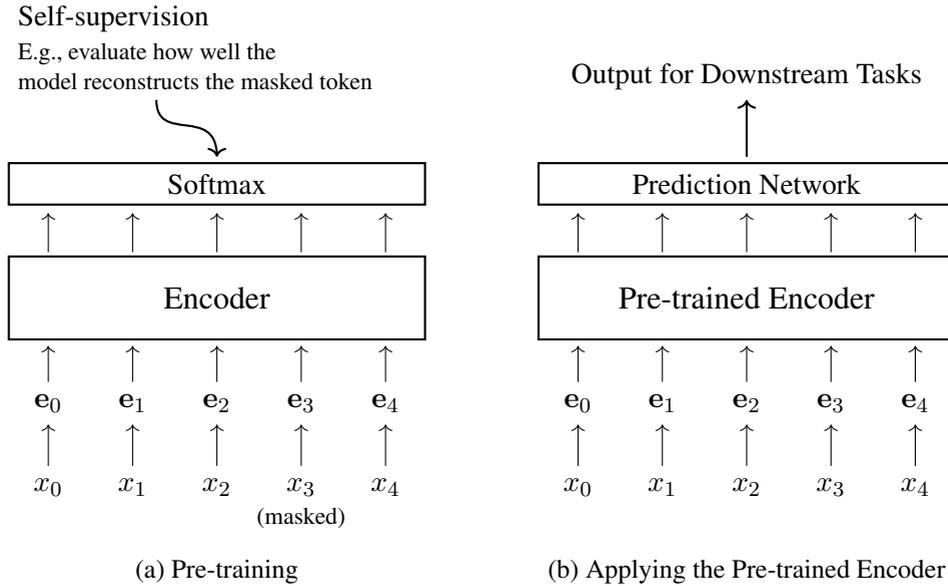


Figure 7.2: Pre-training a Transformer encoder (left) and then applying the pre-trained encoder (right). In the pre-training phase, the encoder, together with a Softmax layer, is trained via self-supervision. In the application phase, the Softmax layer is removed, and the pre-trained encoder is combined with a prediction network to address specific problems. In general, for better adaptation to these tasks, the system is fine-tuned using labeled data.

## 1. Masked Language Modeling

One of the most popular methods of encoder pre-training is **masked language modeling**, which forms the basis of the well-known BERT model [Devlin et al., 2019]. The idea of masked language modeling is to create prediction challenges by masking out some of the tokens in the input sequence and training a model to predict the masked tokens. In this sense, the conventional language modeling problem, which is sometimes called **causal language modeling**, is a special case of masked language modeling: at each position, we mask the tokens in the right-context, and predict the token at this position using its left-context. However, in causal language modeling we only make use of the left-context in word prediction, while the prediction may depend on tokens in the right-context. By contrast, in masked language modeling, all the unmasked tokens are used for word prediction, leading to a bidirectional model that makes predictions based on both left and right-contexts.

More formally, for an input sequence  $\mathbf{x} = x_0 \dots x_m$ , suppose that we mask the tokens at positions  $\mathcal{A}(\mathbf{x}) = \{i_1, \dots, i_u\}$ . Hence we obtain a masked token sequence  $\bar{\mathbf{x}}$  where the token at each position in  $\mathcal{A}(\mathbf{x})$  is replaced with a special symbol [MASK]. For example, for the following sequence

The early bird catches the worm

we may have a masked token sequence like this

The [MASK] bird catches the [MASK]

where we mask the tokens *early* and *worm* (i.e.,  $i_1 = 2$  and  $i_2 = 6$ ).

Now we have two sequences  $\mathbf{x}$  and  $\bar{\mathbf{x}}$ . The model is then optimized so that we can correctly predict  $\mathbf{x}$  based on  $\bar{\mathbf{x}}$ . This can be thought of as an autoencoding-like process, and the training objective is to maximize the reconstruction probability  $\Pr(\mathbf{x}|\bar{\mathbf{x}})$ . Note that there is a simple position-wise alignment between  $\mathbf{x}$  and  $\bar{\mathbf{x}}$ . Because an unmasked token in  $\bar{\mathbf{x}}$  is the same as the token in  $\mathbf{x}$  at the same position, there is no need to consider the prediction for this unmasked token. This leads to a simplified training objective which only maximizes the probabilities for masked tokens. We can express this objective in a maximum likelihood estimation fashion

$$(\widehat{\mathbf{W}}, \hat{\theta}) = \arg \max_{\mathbf{W}, \theta} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i^{\mathbf{W}, \theta}(x_i | \bar{\mathbf{x}}) \quad (7.10)$$

or alternatively express it using the cross-entropy loss

$$(\widehat{\mathbf{W}}, \hat{\theta}) = \arg \min_{\mathbf{W}, \theta} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i \in \mathcal{A}(\mathbf{x})} \text{LogCrossEntropy}(\mathbf{p}_i^{\mathbf{W}, \theta}, \mathbf{p}_i^{\text{gold}}) \quad (7.11)$$

where  $\Pr_k^{\mathbf{W}, \theta}(x_k | \bar{\mathbf{x}})$  is the probability of the true token  $x_k$  at position  $k$  given the corrupted input  $\bar{\mathbf{x}}$ , and  $\mathbf{p}_k^{\mathbf{W}, \theta}$  is the probability distribution at position  $k$  given the corrupted input  $\bar{\mathbf{x}}$ . To illustrate, consider the above example where two tokens of the sequence “*the early bird catches the worm*” are masked. For this example, the objective is to maximize the sum of log-scale probabilities

$$\begin{aligned} \text{Loss} = & \log \Pr(x_2 = \text{early} | \bar{\mathbf{x}} = [\text{CLS}] \text{ The } \underbrace{[\text{MASK}]}_{\bar{x}_2} \text{ bird catches the } \underbrace{[\text{MASK}]}_{\bar{x}_6}) + \\ & \log \Pr(x_6 = \text{worm} | \bar{\mathbf{x}} = [\text{CLS}] \text{ The } \underbrace{[\text{MASK}]}_{\bar{x}_2} \text{ bird catches the } \underbrace{[\text{MASK}]}_{\bar{x}_6}) \end{aligned} \quad (7.12)$$

Once we obtain the optimized parameters  $\widehat{\mathbf{W}}$  and  $\hat{\theta}$ , we can drop  $\widehat{\mathbf{W}}$ . Then, we can further fine-tune the pre-trained encoder  $\text{Encoder}_{\hat{\theta}}(\cdot)$  or directly apply it to downstream tasks.

## 2. Permuted Language Modeling

While masked language modeling is simple and widely applied, it introduces new issues. One drawback is the use of a special token, [MASK], which is employed only during training but not at test time. This leads to a discrepancy between training and inference. Moreover, the auto-encoding process overlooks the dependencies between masked tokens. For example, in the above example, the prediction of  $x_2$  (i.e., the first masked token) is made independently of  $x_6$  (i.e., the second masked token), though  $x_6$  should be considered in the context of  $x_2$ .

These issues can be addressed using the **permuted language modeling** approach to pre-training [Yang et al., 2019]. Similar to causal language modeling, permuted language modeling involves making sequential predictions of tokens. However, unlike causal modeling where

predictions follow the natural sequence of the text (like left-to-right or right-to-left), permuted language modeling allows for predictions in any order. The approach is straightforward: we determine an order for token predictions and then train the model in a standard language modeling manner, as described in Section 7.2.1. Note that in this approach, the actual order of tokens in the text remains unchanged, and only the order in which we predict these tokens differs from standard language modeling. For example, consider a sequence of 5 tokens  $x_0x_1x_2x_3x_4$ . Let  $\mathbf{e}_i$  represent the embedding of  $x_i$  (i.e., combination of the token embedding and positional embedding). In standard language modeling, we would generate this sequence in the order of  $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$ . The probability of the sequence can be modeled via a generation process.

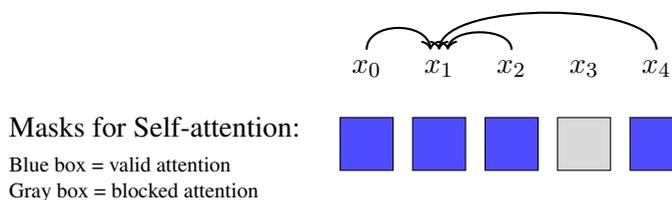
$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdot \Pr(x_3|x_0, x_1, x_2) \cdot \\ &\quad \Pr(x_4|x_0, x_1, x_2, x_3) \\ &= \Pr(x_0) \cdot \Pr(x_1|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_1) \cdot \Pr(x_3|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_4|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3) \end{aligned} \quad (7.13)$$

Now, let us consider a different order for token prediction:  $x_0 \rightarrow x_4 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3$ . The sequence generation process can then be expressed as follows:

$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_4|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_4) \cdot \Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_3|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2, \mathbf{e}_1) \end{aligned} \quad (7.14)$$

This new prediction order allows for the generation of some tokens to be conditioned on a broader context, rather than being limited to just the preceding tokens as in standard language models. For example, in generating  $x_3$ , the model considers both its left-context (i.e.,  $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2$ ) and right-context (i.e.,  $\mathbf{e}_4$ ). The embeddings  $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_4$  incorporate the positional information of  $x_0, x_1, x_2, x_4$ , preserving the original order of the tokens. As a result, this approach is somewhat akin to masked language modeling: we mask out  $x_3$  and use its surrounding tokens  $x_0, x_1, x_2, x_4$  to predict this token.

The implementation of permuted language models is relatively easy for Transformers. Because the self-attention model is insensitive to the order of inputs, we do not need to explicitly reorder the sequence to have a factorization like Eq. (7.14). Instead, permutation can be done by setting appropriate masks for self-attention. For example, consider the case of computing  $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2)$ . We can place  $x_0, x_1, x_2, x_3, x_4$  in order and block the attention from  $x_3$  to  $x_1$  in self-attention, as illustrated below



For a more illustrative example, we compare the self-attention masking results of causal language modeling, masked language modeling and permuted language modeling in Figure 7.3.

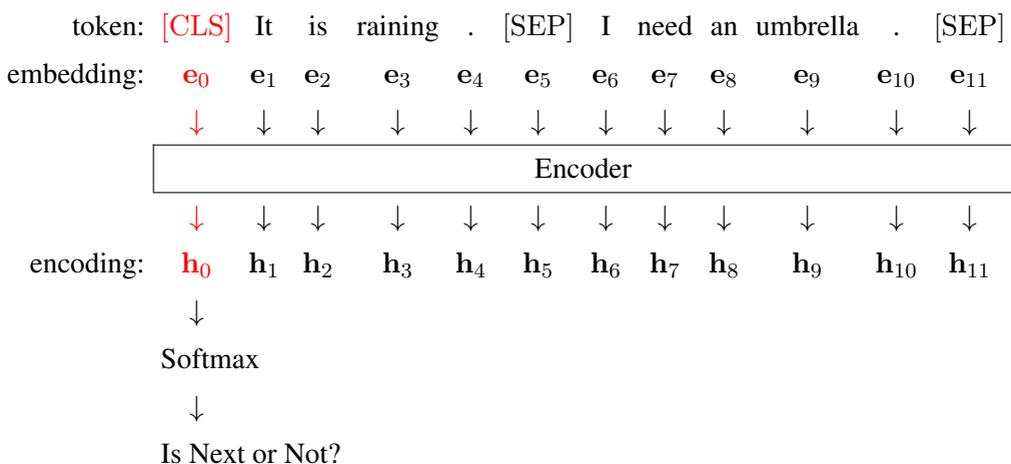
### 3. Pre-training Encoders as Classifiers

Another commonly-used idea to train an encoder is to consider classification tasks. In self-supervised learning, this is typically done by creating new classification challenges from the unlabeled text. There are many different ways to design the classification tasks. Here we present two popular tasks.

A simple method, called **next sentence prediction (NSP)**, is presented in BERT's original paper [Devlin et al., 2019]. The assumption of NSP is that a good text encoder should capture the relationship between two sentences. To model such a relationship, in NSP we can use the output of encoding two consecutive sentences  $Sent_A$  and  $Sent_B$  to determine whether  $Sent_B$  is the next sentence following  $Sent_A$ . For example, suppose  $Sent_A = 'It is raining.'$  and  $Sent_B = 'I need an umbrella.'$  The input sequence of the encoder could be

[CLS] It is raining . [SEP] I need an umbrella . [SEP]

where [CLS] is the start symbol (i.e.,  $x_0$ ) which is commonly used in encoder pre-training, and [SEP] is a separator that separates the two sentences. The processing of this sequence follows a standard procedure of Transformer encoding: we first represent each token  $x_i$  as its corresponding embedding  $e_i$ , and then feed the embedding sequence  $\{e_0, \dots, e_m\}$  into the encoder to obtain the output sequence  $\{h_0, \dots, h_m\}$ . Since  $h_0$  is generally considered as the representation of the entire sequence, we add a Softmax layer on top of it to construct a binary classification system. This process is illustrated as follows



In order to generate training samples, we need two sentences each time, one for  $Sent_A$  and the other for  $Sent_B$ . A simple way to do this is to utilize the natural sequence of two consecutive sentences in the text. For example, we obtain a positive sample by using actual consecutive

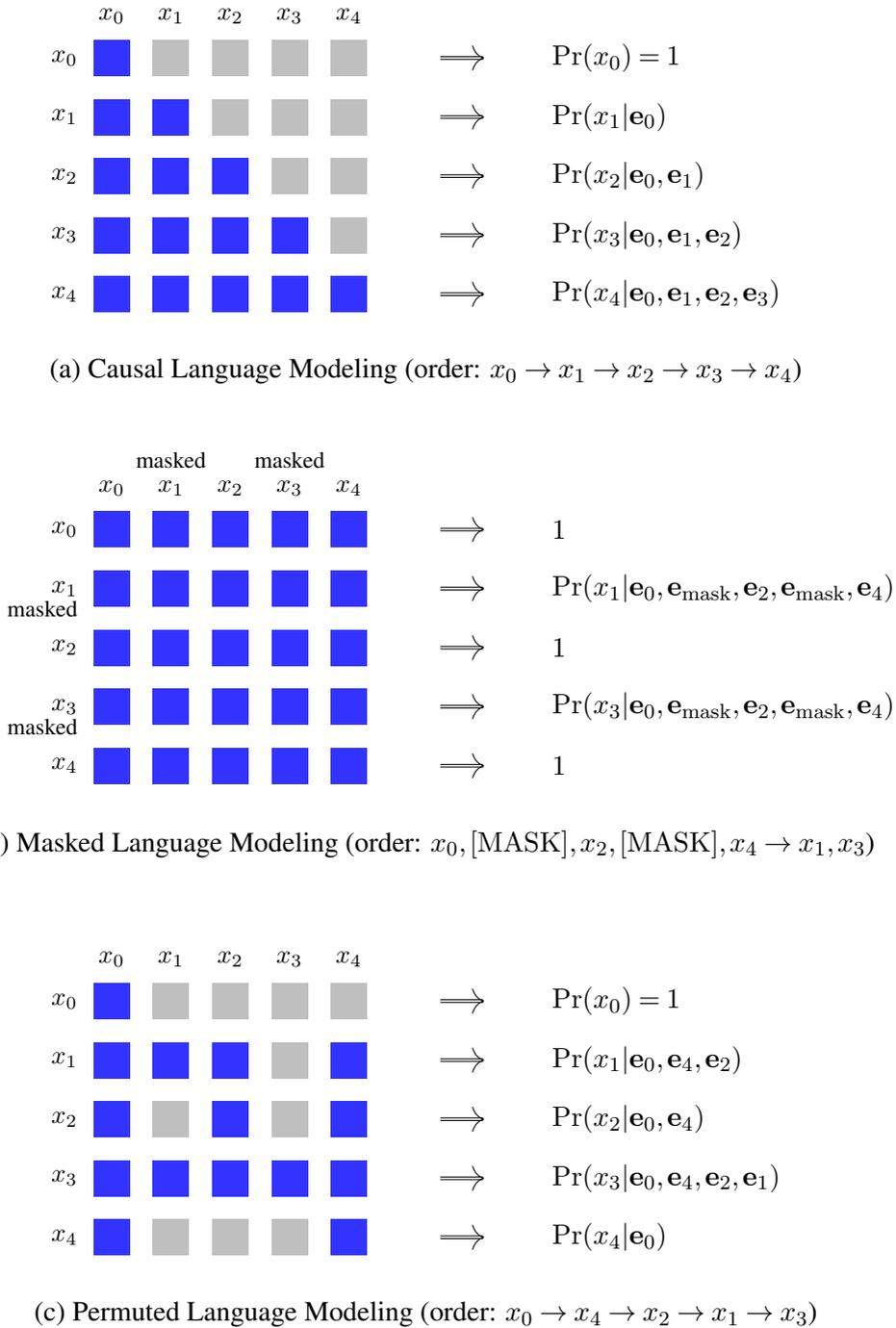
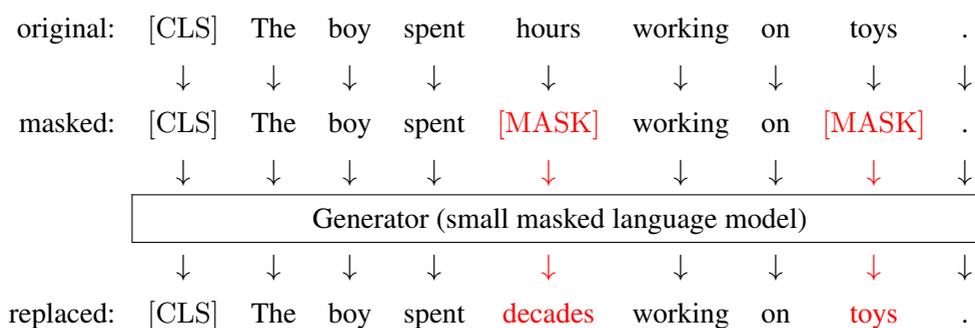


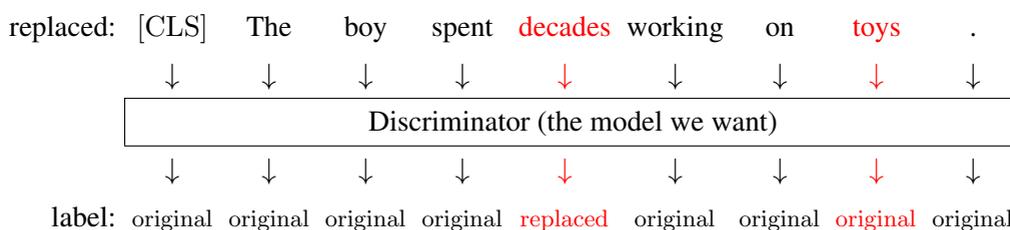
Figure 7.3: Comparison of self-attention masking results of causal language modeling, masked language modeling and permuted language modeling. The gray cell denotes the token at position  $j$  does not attend to the token at position  $i$ . The blue cell  $(i, j)$  denotes that the token at position  $j$  attends to the token at position  $i$ .  $\mathbf{e}_{\text{mask}}$  represents the embedding of the symbol [MASK], which is a combination of the token embedding and the positional embedding.

sentences, and a negative sample by using randomly sampled sentences. Consequently, training this model is the same as training a classifier. Typically, NSP is used as an additional training loss function for pre-training based on masked language modeling.

A second example of training Transformer encoders as classifiers is to apply classification-based supervision signals to each output of an encoder. For example, [Clark et al. \[2019b\]](#) in their ELECTRA model, propose training a Transformer encoder to identify whether each input token is identical to the original input or has been altered in some manner. The first step of this method is to generate a new sequence from a given sequence of tokens, where some of the tokens are altered. To do this, a small masked language model (call it the generator) is applied: we randomly mask some of the tokens, and train this model to predict the masked tokens. For each training sample, this masked language model outputs a token at each masked position, which might be different from the original token. At the same time, we train another Transformer encoder (call it the discriminator) to determine whether each predicted token is the same as the original token or altered. More specifically, we use the generator to generate a sequence where some of the tokens are replaced. Below is an illustration.



Then, we use the discriminator to label each of these tokens as original or replaced, as follows



For training, the generator is optimized as a masked language model with maximum likelihood estimation, and the discriminator is optimized as a classifier using a classification-based loss. In ELECTRA, the maximum likelihood-based loss and the classification-based loss are combined for jointly training both the generator and discriminator. An alternative approach is to use generative adversarial networks (GANs), that is, the generator is trained to fool the discriminator, and the discriminator is trained to distinguish the output of the generator from the true distribution. However, GAN-style training complicates the training task and is more

difficult to scale up. Nevertheless, once training is complete, the generator is discarded, and the encoding part of the discriminator is applied as the pre-trained model for downstream tasks.

### 7.2.3 Encoder-Decoder Pre-training

In NLP, encoder-decoder architectures are often used to model sequence-to-sequence problems, such as machine translation and question answering. In addition to these typical sequence-to-sequence problems in NLP, encoder-decoder models can be extended to deal with many other problems. A simple idea is to consider text as both the input and output of a problem, and so we can directly apply encoder-decoder models. For example, given a text, we can ask a model to output a text describing the sentiment of the input text, such as *positive*, *negative*, and *neutral*.

Such an idea allows us to develop a single text-to-text system to address any NLP problem. We can formulate different problems into the same text-to-text format. We first train an encoder-decoder model to gain general-purpose knowledge of language via self-supervision. This model is then fine-tuned for specific downstream tasks using targeted text-to-text data.

#### 1. Masked Encoder-Decoder Pre-training

In Raffel et al. [2020]’s **T5** model, many different tasks are framed as the same text-to-text task. Each sample in T5 follows the format

Source Text → Target Text

Here → separates the source text, which consists of a task description or instruction and the input given to the system, from the target text, which is the response to the input task. As an example, consider a task of translating from Chinese to English. A training sample can be expressed as

[CLS] Translate from Chinese to English: 你好! → ⟨s⟩ Hello!

where [CLS] and ⟨s⟩ are the start symbols on the source and target sides, respectively<sup>5</sup>.

---

<sup>5</sup>We could use the same start symbol for different sequences. Here we use different symbols to distinguish the sequences on the encoder and decoder-sides.

Likewise, we can express other tasks in the same way. For example

[CLS] **Answer:** when was Albert Einstein born?

→  $\langle s \rangle$  He was born on March 14, 1879.

[CLS] **Simplify:** the professor, who has published numerous papers in his field, will be giving a lecture on the topic next week.

→  $\langle s \rangle$  The experienced professor will give a lecture next week.

[CLS] **Text:** John bought a new car. Hypothesis: John has a car.

→  $\langle s \rangle$  Entailment

[CLS] **Score the translation from English to Chinese.** English: when in Rome, do as the Romans do. Chinese: 人在罗马就像罗马人一样做事。

→  $\langle s \rangle$  0.81

where instructions are highlighted in gray. An interesting case is that in the last example we reframe the scoring problem as the text generation problem. Our goal is to generate a text representing the number 0.81, rather than outputting it as a numerical value.

The approach described above provides a new framework of universal language understanding and generation. Both the task instructions and the problem inputs are provided to the system in text form. The system then follows the instructions to complete the task. This method puts different problems together, with the benefit of training a single model that can perform many tasks simultaneously.

In general, fine-tuning is necessary for adapting the pre-trained model to a specific downstream task. In this process, one can use different ways to instruct the model for the task, such as using a short name of the task as the prefix to the actual input sequence or providing a detailed description of the task. Since the task instructions are expressed in text form and involved as part of the input, the general knowledge of instruction can be gained through learning the language understanding models in the pre-training phase. This may help enable zero-shot learning. For example, pre-trained models can generalize to address new problems where the task instructions have never been encountered.

There have been several powerful methods of self-supervised learning for either Transformer encoders or decoders. Applying these methods to pre-train encoder-decoder models is relatively straightforward. One common choice is to train encoder-decoder models as language models. For example, the encoder receives a sequence prefix, while the decoder generates the remaining sequence. However, this differs from standard causal language modeling, where the entire sequence is autoregressively generated from the first token. In our case, the encoder processes the prefix at once, and then the decoder predicts subsequent tokens in the manner of causal language modeling. Put more precisely, this is a **prefix language modeling** problem: a

language model predicts the subsequent sequence given a prefix, which serves as the context for prediction.

Consider the following example

$$\underbrace{[\text{CLS}] \text{ The puppies are frolicking}}_{\text{Prefix}} \rightarrow \underbrace{\langle s \rangle \text{ outside the house}}_{\text{Subsequent Sequence}} .$$

We can directly train an encoder-decoder model using examples like this. Then, the encoder learns to understand the prefix, and the decoder learns to continue writing based on this understanding. For large-scale pre-training, it is easy to create a large number of training examples from unlabeled text.

It is worth noting that for pre-trained encoder-decoder models to be effective in multi-lingual and cross-lingual tasks, such as machine translation, they should be trained with multi-lingual data. This typically requires that the vocabulary includes tokens from all the languages. By doing so, the models can learn shared representations across different languages, thereby enabling capabilities in both language understanding and generation in a multi-lingual and cross-lingual context.

A second approach to pre-training encoder-decoder models is masked language modeling. In this approach, as discussed in Section 7.2.2, tokens in a sequence are randomly replaced with a mask symbol, and the model is then trained to predict these masked tokens based on the entire masked sequence.

As an illustration, consider the task of masking and reconstructing the sentence

The puppies are frolicking outside the house .

By masking two tokens (say, *frolicking* and *the*), we have the BERT-style input and output of the model, as follows

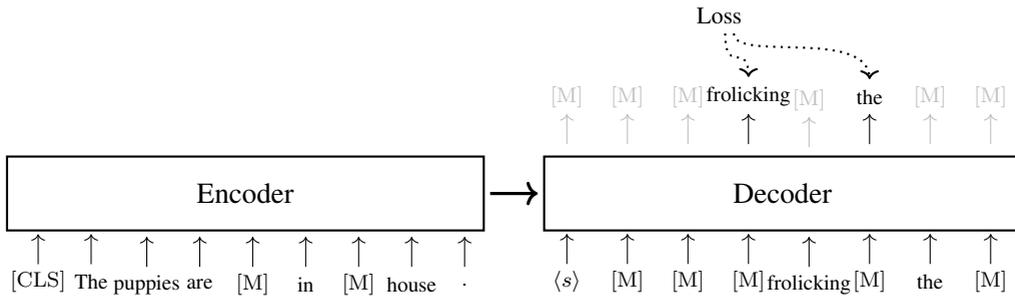
$$\begin{aligned} & [\text{CLS}] \text{ The puppies are } [\text{MASK}] \text{ outside } [\text{MASK}] \text{ house} . \\ \rightarrow & \langle s \rangle \text{ \_\_ \_\_ \_\_ frolicking \_\_ the \_\_ \_\_} \end{aligned}$$

Here `\_\_` denotes the masked position at which we do not make token predictions. By varying the percentage of the tokens in the text, this approach can be generalized towards either BERT-style training or language modeling-style training [Song et al., 2019]. For example, if we mask out all the tokens, then the model is trained to generate the entire sequence

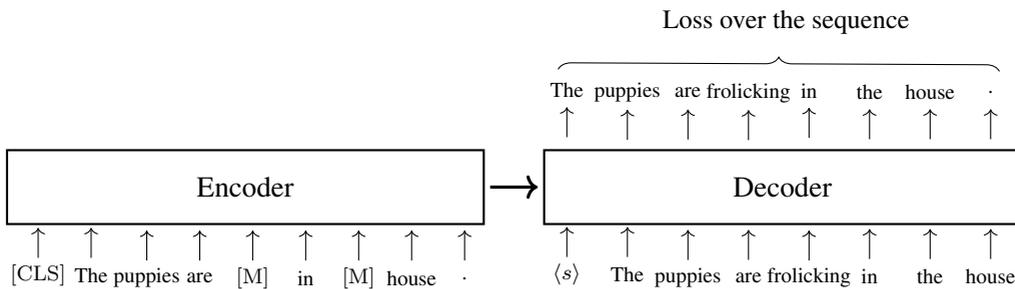
$$\begin{aligned} & [\text{CLS}] [\text{MASK}] [\text{MASK}] [\text{MASK}] [\text{MASK}] [\text{MASK}] [\text{MASK}] [\text{MASK}] [\text{MASK}] \\ \rightarrow & \langle s \rangle \text{ The puppies are frolicking outside the house} . \end{aligned}$$

In this case, we train the decoder as a language model.

Note that, in the context of the encoder-decoder architecture, we can use the encoder to read the masked sequence, and use the decoder to predict the original sequence. With this objective, we essentially have a denoising autoencoder: the encoder transforms a corrupted



(a) Training an encoder-decoder model with BERT-style masked language modeling



(b) Training an encoder-decoder model with denoising autoencoding

Figure 7.4: Training an encoder-decoder model using BERT-style and denoising autoencoding methods. In both methods, the input to the encoder is a corrupted token sequence where some tokens are masked and replaced with [MASK] (or [M] for short). The decoder predicts these masked tokens, but in different ways. In BERT-style training, the decoder only needs to compute the loss for the masked tokens, while the remaining tokens in the sequence can be simply treated as [MASK] tokens. In denoising autoencoding, the decoder predicts the sequence of all tokens in an autoregressive manner. As a result, the loss is obtained by accumulating the losses of all these tokens, as in standard language modeling.

input into some hidden representation, and the decoder reconstructs the uncorrupted input from this hidden representation. Here is an example of input and output for denoising training.

[CLS] The puppies are [MASK] outside [MASK] house .  
 → ⟨s⟩ The puppies are frolicking outside the house .

By learning to map from this corrupted sequence to its uncorrupted counterpart, the model gains the ability to understand on the encoder side and to generate on the decoder side. See Figure 7.4 for an illustration of how an encoder-decoder model is trained with BERT-style and denoising autoencoding objectives.

As we randomly select tokens for masking, we can certainly mask consecutive tokens

[Joshi et al., 2020]. Here is an example.

[CLS] The puppies are [MASK] outside [MASK] [MASK] .  
 → ⟨s⟩ The puppies are frolicking outside the house .

Another way to consider consecutive masked tokens is to represent them as spans. Here we follow Raffel et al. [2020]’s work, and use [X], [Y] and [Z] to denote sentinel tokens that cover one or more consecutive masked tokens. Using this notation, we can re-express the above training example as

[CLS] The puppies are [X] outside [Y] .  
 → ⟨s⟩ [X] frolicking [Y] the house [Z]

The idea is that we represent the corrupted sequence as a sequence containing placeholder slots. The training task is to fill these slots with the correct tokens using the surrounding context. An advantage of this approach is that the sequences used in training would be shorter, making the training more efficient. Note that masked language modeling provides a very general framework for training encoder-decoder models. Various settings can be adjusted to have different training versions, such as altering the percentage of tokens masked and the maximum length of the masked spans.

## 2. Denoising Training

If we view the problem of training encoder-decoder models as a problem of training denoising autoencoders, there will typically be many different methods for introducing input corruption and reconstructing the input. For instance, beyond randomly masking tokens, we can also alter some of them or rearrange their order.

Suppose we have an encoder-decoder model that can map an input sequence  $\mathbf{x}$  to an output sequence  $\mathbf{y}$

$$\begin{aligned} \mathbf{y} &= \text{Decode}_{\omega}(\text{Encode}_{\theta}(\mathbf{x})) \\ &= \text{Model}_{\theta, \omega}(\mathbf{x}) \end{aligned} \quad (7.15)$$

where  $\theta$  and  $\omega$  are the parameters of the encoder and the decoder, respectively. In denoising autoencoding problems, we add some noise to  $\mathbf{x}$  to obtain a noisy, corrupted input  $\mathbf{x}_{\text{noise}}$ . By feeding  $\mathbf{x}_{\text{noise}}$  into the encoder, we wish the decoder to output the original input. The training objective can be defined as

$$(\hat{\theta}, \hat{\omega}) = \underset{\theta, \omega}{\text{arg min}} \text{Loss}(\text{Model}_{\theta, \omega}(\mathbf{x}_{\text{noise}}), \mathbf{x}) \quad (7.16)$$

Here the loss function  $\text{Loss}(\text{Model}_{\theta, \omega}(\mathbf{x}_{\text{noise}}), \mathbf{x})$  evaluates how well the model  $\text{Model}_{\theta, \omega}(\mathbf{x}_{\text{noise}})$  reconstructs the original input  $\mathbf{x}$ . We can choose the cross-entropy loss as usual.

As the model architecture and the training approach have been developed, the remaining

issue is the corruption of the input. Lewis et al. [2020a], in their **BART** model, propose corrupting the input sequence in several different ways.

- **Token Masking.** This is the same masking method that we used in masked language modeling. The tokens in the input sequence are randomly selected and masked.
- **Token Deletion.** This method is similar to token masking. However, rather than replacing the selected tokens with a special symbol [MASK], these tokens are removed from the sequence. See the following example for a comparison of the token masking and token deletion methods.

Original ( $\mathbf{x}$ ): The puppies are frolicking outside the house .  
 Token Masking ( $\mathbf{x}_{\text{noise}}$ ): The puppies are [MASK] outside [MASK] house .  
 Token Deletion ( $\mathbf{x}_{\text{noise}}$ ): The puppies are ~~frolicking~~ outside ~~the~~ house .

where the underlined tokens in the original sequence are masked or deleted.

- **Span Masking.** Non-overlapping spans are randomly sampled over the sequence. Each span is masked by [MASK]. We also consider spans of length 0, and, in such cases, [MASK] is simply inserted at a position in the sequence. For example, we can use span masking to corrupt the above sequence as

Original ( $\mathbf{x}$ ): The 0 puppies are frolicking outside the house .  
 Span Masking ( $\mathbf{x}_{\text{noise}}$ ): The [MASK] puppies are [MASK] house .

Here the span *frolicking outside the* is replaced with a single [MASK]. 0 indicates a length-0 span, and so we insert an [MASK] between *The* and *puppies*. Span masking introduces new prediction challenges in which the model needs to know how many tokens are generated from a span. This problem is very similar to fertility modeling in machine translation [Brown et al., 1993].

If we consider a sequence consisting of multiple sentences, additional methods of corruption can be applied. In the BART model, there are two such methods.

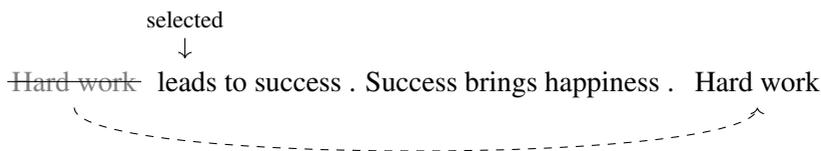
- **Sentence Reordering.** This method randomly permutes the sentences so that the model can learn to reorder sentences in a document. Consider, for example, two consecutive sentences

Hard work leads to success . Success brings happiness .

We can reorder the two sentences to have a corrupted input sequence

Success brings happiness . Hard work leads to success .

- **Document Rotation.** The goal of this task is to identify the start token of the sequence. First, a token is randomly selected from the sequence. Then, the sequence is rotated so that the selected token is the first token. For example, suppose we select the token *leads* from the above sequence. The rotated sequence is



where the subsequence *Hard work* before *leads* is appended to the end of the sequence.

For pre-training, we can apply multiple corruption methods to learn robust models, for example, we randomly choose one of them for each training sample. In practice, the outcome of encoder-decoder pre-training depends heavily on the input corruption methods used, and so we typically need to choose appropriate training objectives through careful experimentation.

## 7.2.4 Comparison of Pre-training Tasks

So far, we have discussed a number of pre-training tasks. Since the same training objective can apply to different architectures (e.g., using masked language modeling for both encoder-only and encoder-decoder pre-training), categorizing pre-training tasks based solely on model architecture does not seem ideal. Instead, we summarize these tasks based on the training objectives.

- **Language Modeling.** Typically, this approach refers to an auto-regressive generation procedure of sequences. At one time, it predicts the next token based on its previous context.
- **Masked Language Modeling.** Masked Language Modeling belongs to a general mask-predict framework. It randomly masks tokens in a sequence and predicts these tokens using the entire masked sequence.
- **Permuted Language Modeling.** Permuted language modeling follows a similar idea to masked language modeling, but considers the order of (masked) token prediction. It reorders the input sequence and predicts the tokens sequentially. Each prediction is based on some context tokens that are randomly selected.
- **Discriminative Training.** In discriminative training, supervision signals are created from classification tasks. Models for pre-training are integrated into classifiers and trained together with the remaining parts of the classifiers to enhance their classification performance.
- **Denoising Autoencoding.** This approach is applied to the pre-training of encoder-decoder models. The input is a corrupted sequence and the encoder-decoder models are trained to reconstruct the original sequence.

Table 7.1 illustrates these methods and their variants using examples. The use of these examples does not distinguish between models, but we mark the model architectures where the pre-training tasks can be applied. In each example, the input consists of a token sequence, and the output is either a token sequence or some probabilities. For generation tasks, such as language modeling, superscripts are used to indicate the generation order on the target side. If the superscripts are omitted, it indicates that the output sequence can be generated

Method	Enc	Dec	E-D	Input	Output
Causal LM		•	•		The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup>
Prefix LM		•	•	[C] The kitten is	chasing <sup>1</sup> the <sup>2</sup> ball <sup>3</sup> . <sup>4</sup>
Masked LM	•		•	[C] The kitten [M] chasing the [M] .	__ __ is __ __ ball __
MASS-style	•		•	[C] The kitten [M] [M] [M] ball .	__ __ is chasing the __ __
BERT-style	•		•	[C] The kitten [M] playing the [M] .	__ kitten is chasing __ ball __
Permuted LM	•			[C] The kitten is chasing the ball .	The <sup>5</sup> kitten <sup>7</sup> is <sup>6</sup> chasing <sup>1</sup> the <sup>4</sup> ball <sup>2</sup> . <sup>3</sup>
Next Sentence Prediction	•			[C] The kitten is chasing the ball . Birds eat worms .	$\Pr(\text{IsNext} \mid \text{representation-of-[C]})$
Sentence Comparison	•			Encode a sentence as $\mathbf{h}_a$ and another sentence as $\mathbf{h}_b$	$\text{Score}(\mathbf{h}_a, \mathbf{h}_b)$
Token Classification	•			[C] The kitten is chasing the ball .	$\Pr(\cdot \mid \text{The}) \Pr(\cdot \mid \text{kitten}) \dots \Pr(\cdot \mid \cdot)$
Token Reordering			•	[C] . kitten the chasing The is ball	The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup>
Token Deletion			•	[C] The kitten is <del>chasing</del> the ball .	The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup>
Span Masking			•	[C] The kitten [M] is [M] .	The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup>
Sentinel Masking			•	[C] The kitten [X] the [Y]	[X] <sup>1</sup> is <sup>2</sup> chasing <sup>3</sup> [Y] <sup>4</sup> ball <sup>5</sup> . <sup>6</sup>
Sentence Reordering			•	[C] The ball rolls away swiftly . The kitten is chasing the ball .	The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup> The <sup>8</sup> ball <sup>9</sup> rolls <sup>10</sup> away <sup>11</sup> swiftly <sup>12</sup> . <sup>13</sup>
Document Rotation			•	[C] chasing the ball . The ball rolls away swiftly . The kitten is	The <sup>1</sup> kitten <sup>2</sup> is <sup>3</sup> chasing <sup>4</sup> the <sup>5</sup> ball <sup>6</sup> . <sup>7</sup> The <sup>8</sup> ball <sup>9</sup> rolls <sup>10</sup> away <sup>11</sup> swiftly <sup>12</sup> . <sup>13</sup>

Table 7.1: Comparison of pre-training tasks, including **language modeling**, **masked language modeling**, **permuted language modeling**, **discriminative training**, and **denoising autoencoding**. [C] = [CLS], [M] = [MASK], [X], [Y] = sentinel tokens. Enc, Dec and E-D indicate whether the approach can be applied to encoder-only, decoder-only, encoder-decoder models, respectively. For generation tasks, superscripts are used to represent the order of the tokens.

either autoregressively or simultaneously. On the source side, we assume that the sequence undergoes a standard Transformer encoding process, meaning that each token can see the entire sequence in self-attention. The only exception is in permuted language modeling, where an autoregressive generation process is implemented by setting attention masks on the encoder side. To simplify the discussion, we remove the token  $\langle s \rangle$  from the target-side of each example.

While these pre-training tasks are different, it is possible to compare them in the same framework and experimental setup [Dong et al., 2019; Raffel et al., 2020; Lewis et al., 2020a]. Note that we cannot list all the pre-training tasks here as there are many of them. For more discussions on pre-training tasks, the interested reader may refer to some surveys on this topic [Qiu et al., 2020b; Han et al., 2021a].

## 7.3 Example: BERT

In this section, we introduce BERT models, which are among the most popular and widely used pre-trained sequence encoding models in NLP.

### 7.3.1 The Standard Model

The standard BERT model, which is proposed in [Devlin et al. \[2019\]](#)'s work, is a Transformer encoder trained using both masked language modeling and next sentence prediction tasks. The loss used in training this model is a sum of the loss of the two tasks.

$$\text{Loss}_{\text{BERT}} = \text{Loss}_{\text{MLM}} + \text{Loss}_{\text{NSP}} \quad (7.17)$$

As is regular in training deep neural networks, we optimize the model parameters by minimizing this loss. To do this, a number of training samples are collected. During training, a batch of training samples is randomly selected from this collection at a time, and  $\text{Loss}_{\text{BERT}}$  is accumulated over these training samples. Then, the model parameters are updated via gradient descent or its variants. This process is repeated many times until some stopping criterion is satisfied, such as when the training loss converges.

#### 1. Loss Functions

In general, BERT models are used to represent a single sentence or a pair of sentences, and thus can handle various downstream language understanding problems. In this section we assume that the input representation is a sequence containing two sentences  $\text{Sent}_A$  and  $\text{Sent}_B$ , expressed as

$$[\text{CLS}] \text{ Sent}_A [\text{SEP}] \text{ Sent}_B [\text{SEP}]$$

Here we follow the notation in BERT's paper and use  $[\text{SEP}]$  to denote the separator.

Given this sequence, we can obtain  $\text{Loss}_{\text{MLM}}$  and  $\text{Loss}_{\text{NSP}}$  separately. For masked language modeling, we predict a subset of the tokens in the sequence. Typically, a certain percentage of the tokens are randomly selected, for example, in the standard BERT model, 15% of the tokens in each sequence are selected. Then the sequence is modified in three ways

- **Token Masking.** 80% of the selected tokens are masked and replaced with the symbol  $[\text{MASK}]$ . For example

Original:      $[\text{CLS}]$  It is raining .  $[\text{SEP}]$  I need an umbrella .  $[\text{SEP}]$   
 Masked:      $[\text{CLS}]$  It is  $[\text{MASK}]$  .  $[\text{SEP}]$  I need  $[\text{MASK}]$  umbrella .  $[\text{SEP}]$

where the selected tokens are underlined. Predicting masked tokens makes the model learn to represent tokens from their surrounding context.

- **Random Replacement.** 10% of the selected tokens are changed to a random token. For

example

Original: [CLS] It is raining . [SEP] I need an umbrella . [SEP]  
 Random Token: [CLS] It is raining . [SEP] I need an **hat** . [SEP]

This helps the model learn to recover a token from a noisy input.

- **Unchanged.** 10% of the selected tokens are kept unchanged. For example,

Original: [CLS] It is raining . [SEP] I need an umbrella . [SEP]  
 Unchanged Token: [CLS] It is raining . [SEP] **I** need an umbrella . [SEP]

This is not a difficult prediction task, but can guide the model to use easier evidence for prediction.

Let  $\mathcal{A}(\mathbf{x})$  be the set of selected positions of a given token sequence  $\mathbf{x}$ , and  $\bar{\mathbf{x}}$  be the modified sequence of  $\mathbf{x}$ . The loss function of masked language modeling can be defined as

$$\text{Loss}_{\text{MLM}} = - \sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i(x_i | \bar{\mathbf{x}}) \quad (7.18)$$

where  $\Pr_i(x_i | \bar{\mathbf{x}})$  is the probability of predicting  $x_i$  at the position  $i$  given  $\bar{\mathbf{x}}$ . Figure 7.5 shows a running example of computing  $\text{Loss}_{\text{MLM}}$ .

For next sentence prediction, we follow the method described in Section 7.2.2. Each training sample is classified into a label set  $\{\text{IsNext}, \text{NotNext}\}$ , for example,

Sequence: [CLS] It is raining . [SEP] I need an umbrella . [SEP]  
 Label: IsNext

Sequence: [CLS] The cat sleeps on the windowsill . [SEP] Apples grow on trees . [SEP]  
 Label: NotNext

The output vector of the encoder for the first token [CLS] is viewed as the sequence representation, denoted by  $\mathbf{h}_{\text{cls}}$  (or  $\mathbf{h}_0$ ). A classifier is built on top of  $\mathbf{h}_{\text{cls}}$ . Then, we can compute the probability of a label  $c$  given  $\mathbf{h}_{\text{cls}}$ , i.e.,  $\Pr(c | \mathbf{h}_{\text{cls}})$ . There are many loss functions one can choose for classification problems. For example, in maximum likelihood training, we can define  $\text{Loss}_{\text{NSP}}$  as

$$\text{Loss}_{\text{NSP}} = -\log \Pr(c_{\text{gold}} | \mathbf{h}_{\text{cls}}) \quad (7.19)$$

where  $c_{\text{gold}}$  is the correct label for this sample.

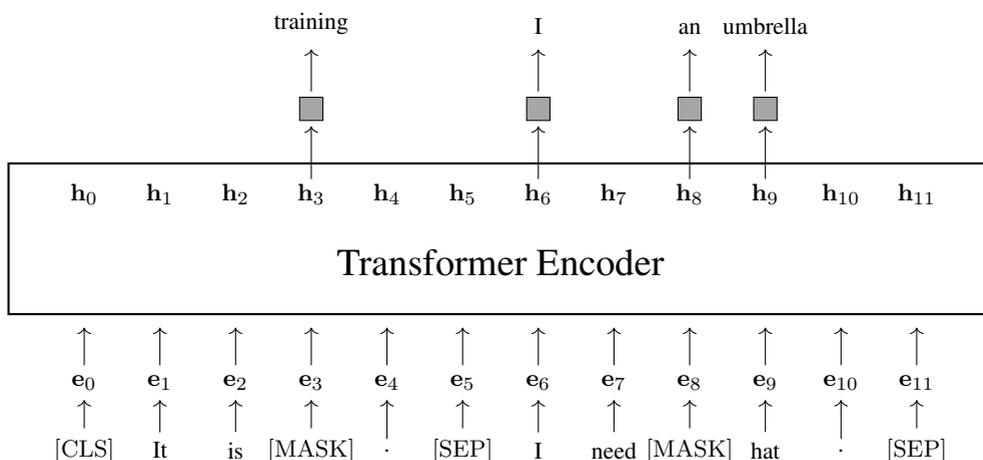
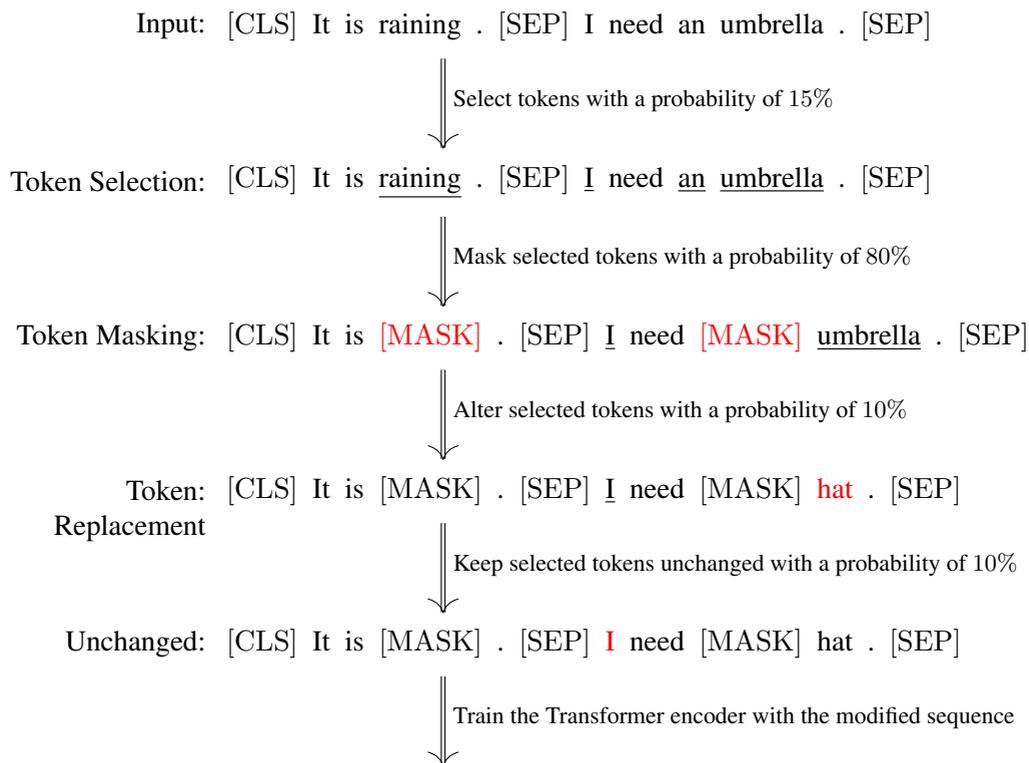


Figure 7.5: A running example of BERT-style masked language modeling. First, 15% of the tokens are randomly selected. These selected tokens are then processed in one of three ways: replaced with a [MASK] token (80% of the time), replaced with a random token (10% of the time), or kept unchanged (10% of the time). The model is trained to predict these selected tokens based on the modified sequence.  $e_i$  represents the embedding of the token at the position  $i$ . Gray boxes represent the Softmax layers.

## 2. Model Setup

As shown in Figure ??, BERT models are based on the standard Transformer encoder architecture. The input is a sequence of embeddings, each being the sum of the token embedding, the

positional embedding, and the segment embedding.

$$\mathbf{e} = \mathbf{x} + \mathbf{e}_{\text{pos}} + \mathbf{e}_{\text{seg}} \quad (7.20)$$

Both the token embedding ( $\mathbf{x}$ ) and positional embedding ( $\mathbf{e}_{\text{pos}}$ ) are regular, as in Transformer models. The segment embedding ( $\mathbf{e}_{\text{seg}}$ ) is a new type of embedding that indicates whether a token belongs to  $\text{Sent}_A$  or  $\text{Sent}_B$ . This can be illustrated by the following example.

Token	[CLS]	It	is	raining	.	[SEP]	I	need	an	umbrella	.	[SEP]
$\mathbf{x}$	$\mathbf{x}_0$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$	$\mathbf{x}_9$	$\mathbf{x}_{10}$	$\mathbf{x}_{11}$
$\mathbf{e}_{\text{pos}}$	PE(0)	PE(1)	PE(2)	PE(3)	PE(4)	PE(5)	PE(6)	PE(7)	PE(8)	PE(9)	PE(10)	PE(11)
$\mathbf{e}_{\text{seg}}$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$

The main part of BERT models is a multi-layer Transformer network. A Transformer layer consists of a self-attention sub-layer and an FFN sub-layer. Both of them follow the post-norm architecture:  $\text{output} = \text{LNorm}(F(\text{input}) + \text{input})$ , where  $F(\cdot)$  is the core function of the sub-layer (either a self-attention model or an FFN), and  $\text{LNorm}(\cdot)$  is the layer normalization unit. Typically, a number of Transformer layers are stacked to form a deep network. At each position of the sequence, the output representation is a real-valued vector which is produced by the last layer of the network.

There are several aspects one may consider in developing BERT models.

- **Vocabulary Size** ( $|V|$ ). In Transformers, each input token is represented as an entry in a vocabulary  $V$ . Large vocabularies can cover more surface form variants of words, but may lead to increased storage requirements.
- **Embedding Size** ( $d_e$ ). Every token is represented as a  $d_e$ -dimensional real-valued vector. As presented above, this vector is the sum of the token embedding, positional embedding, and segment embedding, all of which are also  $d_e$ -dimensional real-valued vectors.
- **Hidden Size** ( $d$ ). The input and output of a sub-layer are of  $d$  dimensions. Besides, most of the hidden states of a sub-layer are  $d$ -dimensional vectors. In general,  $d$  can be roughly viewed as the width of the network.
- **Number of Heads** ( $n_{\text{head}}$ ). In self-attention sub-layers, one needs to specify the number of heads used in multi-head self-attention. The larger this number is, the more sub-spaces in which attention is performed. In practical systems, we often set  $n_{\text{head}} \geq 4$ .
- **FFN Hidden Size** ( $d_{\text{ffn}}$ ). The size of the hidden layer of the FFNs used in Transformers is typically larger than  $d$ . For example, a typical setting is  $d_{\text{ffn}} = 4d$ . For larger Transformers, such as recent large models,  $d_{\text{ffn}}$  may be set to a very large value.
- **Model Depth** ( $L$ ). Using deep networks is an effective way to improve the expressive power of Transformers. For BERT models,  $L$  is typically set to 12 or 24. However, networks with even greater depth are also feasible and can be applied for further enhancements.

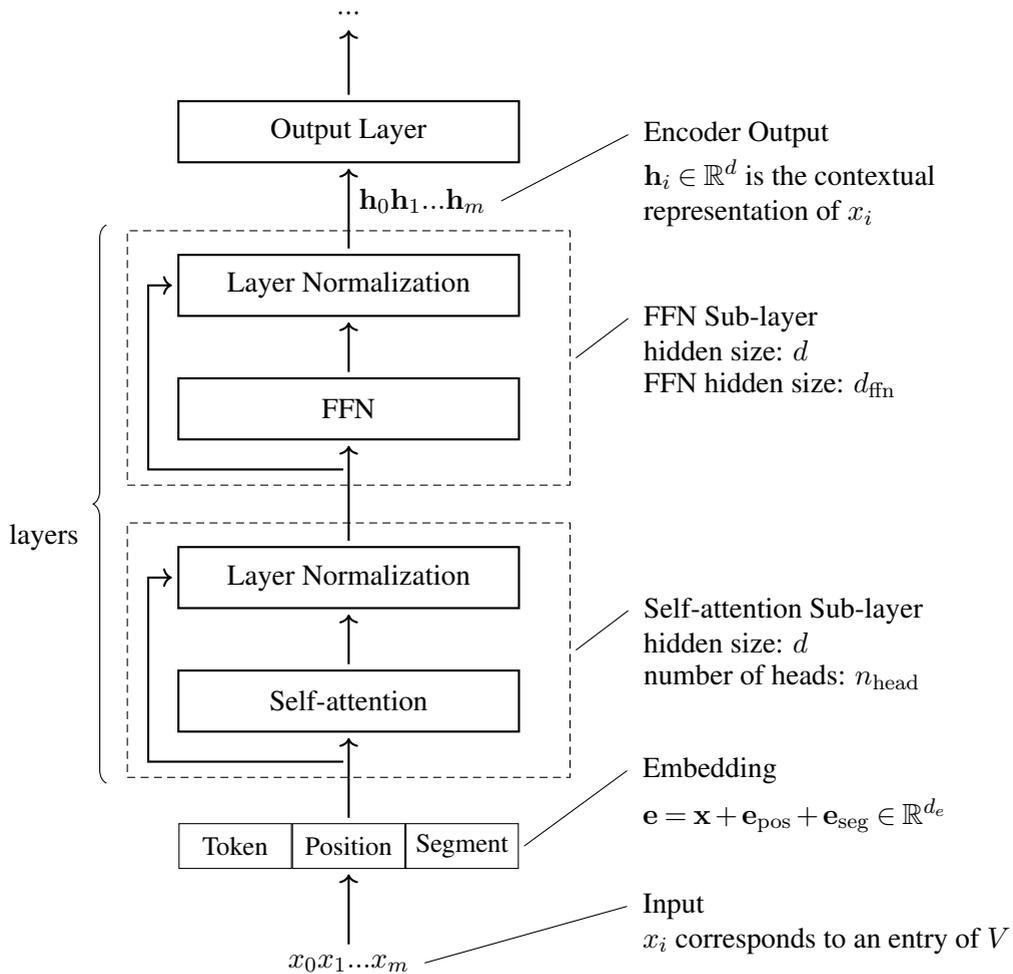


Figure 7.6: The model architecture of BERT (Transformer encoder). The input tokens are first represented as embeddings, each of which is the sum of the corresponding token embedding, positional embedding and segment embedding. Then, the embedding sequence is processed by a stack of Transformer layers. Each layer in this stack includes a self-attention sub-layer and a FFN sub-layer. The output of the BERT model is a sequence of vectors produced by the final Transformer layer.

Different settings of these hyper-parameters lead to different model sizes. There are two widely-used BERT models.

- BERT<sub>base</sub>:  $d = 768$ ,  $L = 12$ ,  $n_{\text{head}} = 12$ , total number of parameters = 110M.
- BERT<sub>large</sub>:  $d = 1,024$ ,  $L = 24$ ,  $n_{\text{head}} = 16$ , total number of parameters = 340M.

Training BERT models follows the standard training process of Transformers. Training larger models such as BERT<sub>large</sub> requires more training effort and time. This is a common problem for pre-training, especially when a model is trained on a very large amount of data. In practice, there are often considerations of training efficiency. For example, a practice is to first train a BERT model on relatively short sequences for a large number of training steps, and

then continue training it on full-length sequences for the remaining training steps.

### 7.3.2 More Training and Larger Models

BERT is a milestone model in NLP, sparking many subsequent efforts to improve it. One direction is to scale up the model itself, including increasing training data and developing larger models.

**RoBERTa**, an extension of the standard BERT model, is an example of such efforts [Liu et al., 2019]. It introduces two major improvements. First, simply using more training data and more compute can improve BERT models without need of changing the model architectures. Second, removing the NSP loss does not decrease the performance on downstream tasks if the training is scaled up. These findings suggest exploring a general direction of pre-training: we can continue to improve pre-training by scaling it up on simple pre-training tasks.

A second approach to improving BERT models is to increase the number of model parameters. For example, in He et al. [2021]’s work, a 1.5 billion-parameter BERT-like model is built by increasing both the model depth and hidden size. However, scaling up BERT and various other pre-trained models introduces new challenges in training, for example, training very large models often becomes unstable and difficult to converge. This makes the problem more complicated, and requires careful consideration of various aspects, including model architecture, parallel computation, parameter initialization, and so on. In another example, Shoeybi et al. [2019] successfully trained a 3.9 billion-parameter BERT-like model, where hundreds of GPUs were used to manage the increased computational demands.

### 7.3.3 More Efficient Models

Compared to its predecessors, BERT is a relatively large model for the time it was proposed. This increase in model size results in larger memory requirements and a consequent slowdown in system performance. Developing smaller and faster BERT models is part of the broader challenge of building efficient Transformers, which has been extensively discussed in Chapter 6. However, a deeper discussion of this general topic is beyond the scope of our current discussion. Here we instead consider a few efficient variants of BERT.

Several threads of research are of interest to NLP researchers in developing efficient BERT models. First, work on knowledge distillation, such as training student models with the output of well-trained teacher models, shows that smaller BERT models can be obtained by transferring knowledge from larger BERT models. Given that BERT models are multi-layer networks with several different types of layers, knowledge distillation can be applied at different levels of representation. For example, beyond distilling knowledge from the output layers, it is also possible to incorporate training loss that measures the difference in output of hidden layers between teacher models and student models [Sun et al., 2020b; Jiao et al., 2020]. Indeed, knowledge distillation has been one of the most widely-used techniques for learning small pre-trained models.

Second, conventional model compression methods can be directly applied to compress BERT models. One common approach is to use general-purpose pruning methods to prune the Transformer encoding networks [Gale et al., 2019]. This generally involves removing entire

layers [Fan et al., 2019] or a certain percentage of parameters in the networks [Sanh et al., 2020; Chen et al., 2020b]. Pruning is also applicable to multi-head attention models. For example, Michel et al. [2019] show that removing some of the heads does not significantly decrease the performance of BERT models, but speeds up the inference of these models. Another approach to compressing BERT models is quantization [Shen et al., 2020b]. By representing model parameters as low-precision numbers, the models can be greatly compressed. While this method is not specific to BERT models, it proves effective for large Transformer-based architectures.

Third, considering that BERT models are relatively deep and large networks, another thread of research uses dynamic networks to adapt these models for efficient inference. An idea in this paradigm is to dynamically choose the layers for processing a token, for example, in depth-adaptive models we exit at some optimal depth and thus skip the rest of the layers in the layer stack [Xin et al., 2020; Zhou et al., 2020]. Similarly, we can develop length-adaptive models in which the length of the input sequence is dynamically adjusted. For example, we can skip some of the tokens in the input sequence so that the model can reduce computational load on less important tokens, enhancing overall efficiency.

Fourth, it is also possible to share parameters across layers to reduce the size of BERT models. A simple way to do this is to share the parameters of a whole Transformer layer across the layer stack [Dehghani et al., 2018; Lan et al., 2020]. In addition to the reduced number of parameters, this enables reuse of the same layer in a multi-layer Transformer network, leading to savings of memory footprint at test time.

### 7.3.4 Multi-lingual Models

The initial BERT model was primarily focused on English. Soon after this model was proposed, it was extended to many languages. One simple way to do this is to develop a separate model for each language. Another approach, which has become more popular in recent work on large language models, is to train multi-lingual models directly on data from all the languages. In response, **multi-lingual BERT (mBERT)** models were developed by training them on text from 104 languages<sup>6</sup>. The primary difference from monolingual BERT models is that mBERT models use larger vocabularies to cover tokens from multiple languages. As a result, the representations of tokens from different languages are mapped into the same space, allowing for the sharing of knowledge across languages via this universal representation model.

One important application of multi-lingual pre-trained models is cross-lingual learning. In the cross-lingual setting, we learn a model on tasks in one language, and apply it to the same tasks in another language. In cross-lingual text classification, for example, we fine-tune a multi-lingual pre-trained model on English annotated documents. Then, we use the fine-tuned model to classify Chinese documents.

An improvement to multi-lingual pre-trained models like mBERT is to introduce bilingual data into pre-training. Rather than training solely on monolingual data from multiple languages, bilingual training explicitly models the relationship between tokens in two languages. The

---

<sup>6</sup><https://github.com/google-research/bert/>

resulting model will have innate cross-lingual transfer abilities, and thus can be easily adapted to different languages. [Lample and Conneau \[2019\]](#) propose an approach to pre-training **cross-lingual language models (XLMs)**. In their work, a cross-lingual language model can be trained in either the causal language modeling or masked language modeling manner. For masked language modeling pre-training, the model is treated as an encoder. The training objective is the same as BERT: we maximize the probabilities of some randomly selected tokens which are either masked, replaced with random tokens, or kept unchanged in the input. If we consider bilingual data in pre-training, we sample a pair of aligned sentences each time. Then, the two sentences are packed together to form a single sequence used for training. For example, consider an English-Chinese sentence pair

鲸鱼 是 哺乳 动物 。 ↔ Whales are mammals .

We can pack them to obtain a sequence, like this

[CLS] 鲸鱼 是 哺乳 动物 。 [SEP] Whales are mammals . [SEP]

We then select a certain percentage of the tokens and replace them with [MASK].

[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]

The goal of pre-training is to maximize the product of the probabilities of the masked tokens given the above sequence. By performing training in this way, the model can learn to represent both the English and Chinese sequences, as well as to capture the correspondences between tokens in the two languages. For example, predicting the Chinese token 鲸鱼 may require the information from the English token *Whales*. Aligning the representations of the two languages essentially transforms the model into a “translation” model. So this training objective is also called **translation language modeling**. [Figure 7.7](#) shows an illustration of this approach.

A benefit of multi-lingual pre-trained models is their inherent capability of handling code-switching. In NLP and linguistics, code-switching refers to switching among languages in a text. For example, the following is a mixed language text containing both Chinese and English:

周末 我们 打算 去 做 hiking , 你 想 一 起 来 吗 ?  
(We plan to go hiking this weekend, would you like to join us?)

For multi-lingual pre-trained models, we do not need to identify whether a token is Chinese or English. Instead, every token is just an entry of the shared vocabulary. This can be imagined as creating a “new” language that encompasses all the languages we want to process.

The result of multi-lingual pre-training is influenced by several factors. Given that the model architecture is fixed, one needs to specify the size of the shared vocabulary, the number (or percentage) of samples in each language, the size of the model, and so on. [Conneau et al.](#)

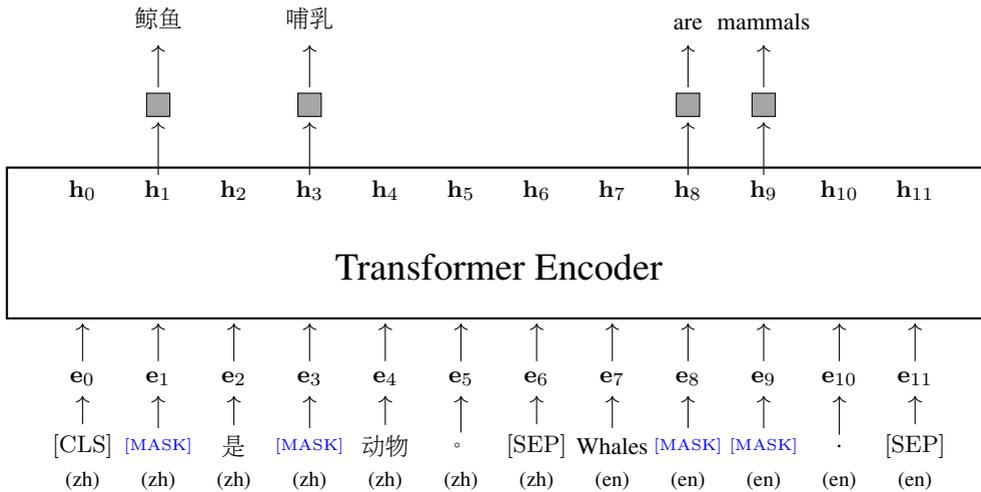


Figure 7.7: An illustration of translation language modeling. For ease of understanding, we present a simple example where all the selected tokens are masked. The model is trained to predict these masked tokens. As the sequence contains tokens in two languages, predicting a token in one language allows access to tokens in the other language, thereby enabling cross-lingual modeling. In [Lample and Conneau \[2019\]](#)’s work, an input embedding (i.e.,  $e_i$ ) is the sum of the token embedding, positional embedding, and language embedding. This requires that each token is assigned with a language label. Thus we can distinguish tokens in different languages. In multi-lingual pre-training, particularly in work using shared vocabularies, specifying the language to which a token belongs is not necessary. The use of language embeddings in turn makes it difficult to handle code-switching. Therefore, we assume here that all token representations are language-independent.

[\[2020\]](#) point out several interesting issues regarding large-scale multi-lingual pre-training for XLM-like models. First, as the number of supported languages increases, a larger model is needed to handle these languages. Second, a larger shared vocabulary is helpful for modeling the increased diversity in languages. Third, low-resource languages more easily benefit from cross-lingual transfer from high-resource languages, particularly when similar high-resource languages are involved in pre-training. However, **interference** may occur if the model is trained for an extended period, meaning the overall performance of the pre-trained model starts decreasing at a certain point during pre-training. Thus, in practical systems, one may need to stop the pre-training early to prevent interference.

## 7.4 Applying BERT Models

Once a BERT model is pre-trained, it can then be used to solve NLP problems. But BERT models are not immediately ready for performing specific downstream tasks. In general, additional fine-tuning work is required to make them adapt. As a first step, we need a predictor to align the output of the model with the problem of interest. Let  $\text{BERT}_{\hat{\theta}}(\cdot)$  be a BERT model with pre-trained parameters  $\hat{\theta}$ , and  $\text{Predict}_{\omega}(\cdot)$  be a prediction network with parameters  $\omega$ . By

integrating the prediction network with the output of the BERT model, we develop a model to tackle the downstream tasks. This model can be expressed as

$$\mathbf{y} = \text{Predict}_{\omega}(\text{BERT}_{\hat{\theta}}(\mathbf{x})) \quad (7.21)$$

where  $\mathbf{x}$  is the input and  $\mathbf{y}$  is the output that fits the problem. For example, in classification problems, the model outputs a probability distribution over labels.

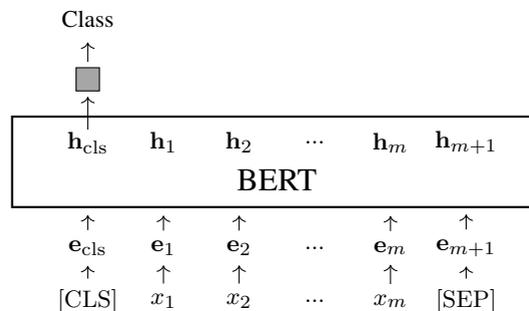
Then, we collect a set of labeled samples  $\mathcal{D}$ , and fine-tune the model by

$$(\tilde{\omega}, \tilde{\theta}) = \underset{\omega, \hat{\theta}^+}{\text{argmin}} \sum_{(\mathbf{x}, \mathbf{y}_{\text{gold}}) \in \mathcal{D}} \text{Loss}(\mathbf{y}_{\omega, \hat{\theta}^+}, \mathbf{y}_{\text{gold}}) \quad (7.22)$$

where  $(\mathbf{x}, \mathbf{y}_{\text{gold}})$  represents a tuple of an input and its corresponding output. The notation of this equation seems a bit complicated, but the training/tuning process is standard. We optimize the model by minimizing the loss over the tuning samples. The outcome is the optimized parameters  $\tilde{\omega}$  and  $\tilde{\theta}$ . The optimization starts with the pre-trained parameters  $\hat{\theta}$ . Here we use  $\hat{\theta}^+$  to indicate that the parameters are initialized with  $\hat{\theta}$ , and use  $\mathbf{y}_{\omega, \hat{\theta}^+}$  to denote the model output computed using the parameters  $\omega$  and  $\hat{\theta}^+$ .

With the fine-tuned parameters  $\tilde{\omega}$  and  $\tilde{\theta}$ , we can apply the model  $\text{Predict}_{\tilde{\omega}}(\text{BERT}_{\tilde{\theta}}(\cdot))$  to new data of the same tasks for which the model was fine-tuned. The form of the downstream tasks determines the input and output formats of the model, as well as the architecture of the prediction network. In the following we list some tasks to which BERT models are generally suited.

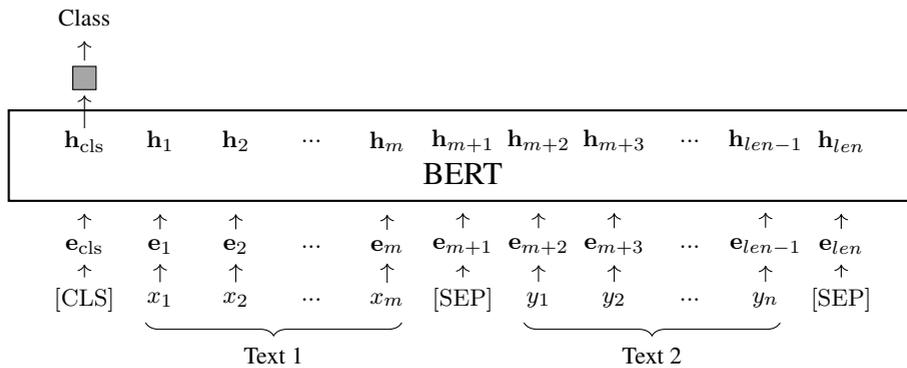
- **Classification** (Single Text). One of the most widely-used applications of BERT models is text classification. In this task, a BERT model receives a sequence of tokens and encodes it as a sequence of vectors. The first output vector  $\mathbf{h}_{\text{cls}}$  (or  $\mathbf{h}_0$ ) is typically used as the representation of the entire text. The prediction network takes  $\mathbf{h}_{\text{cls}}$  as input to produce a distribution of labels. Let  $[\text{CLS}]x_1x_2\dots x_mx_m$  be an input text. See below for an illustration of BERT-based text classification.



Here the gray box denotes the prediction network. Many NLP problems can be categorized as text classification tasks, and there have been several text classification benchmarks for evaluating pre-trained models. For example, we can classify texts by their grammatical correctness (grammaticality) or emotional tone (sentiment) [Socher

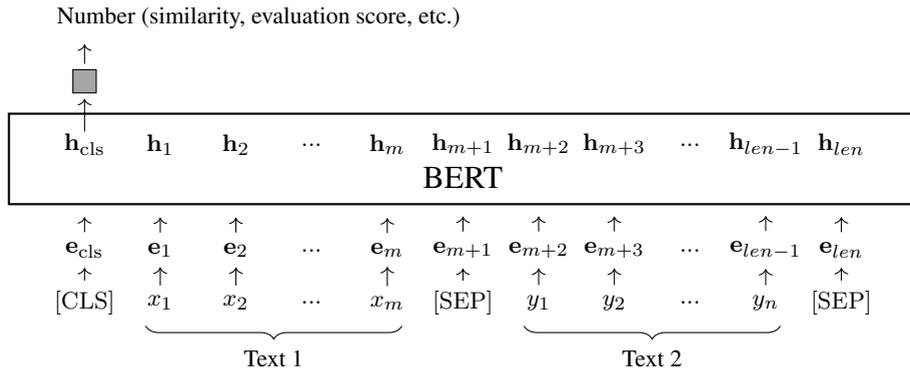
et al., 2013; Warstadt et al., 2019]. Note that the prediction network could be any classification model, such as a deep neural network or a more traditional classification model. The entire model can then be trained or fine-tuned in the manner of a standard classification model. For example, the prediction network can be simply a Softmax layer and the model parameters can be optimized by maximizing the probabilities of the correct labels.

- **Classification (Pair of Texts).** Classification can also be performed on a pair of texts. Suppose we have two texts,  $x_1 \dots x_m$  and  $y_1 \dots y_n$ . We can concatenate these texts to form a single sequence with a length  $len$ . Then, we predict a label for this combined text sequence based on the  $\mathbf{h}_{cls}$  vector, as follows



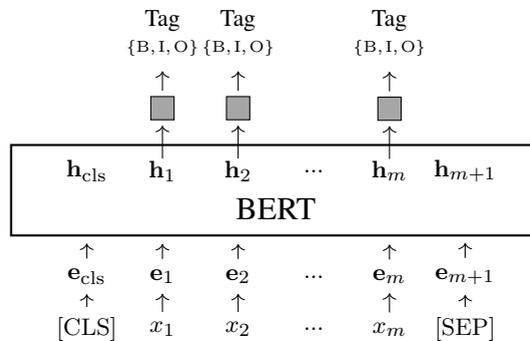
where  $len = n + m + 2$ . Text pair classification covers several problems, including semantic equivalence judgement (determine whether two texts are semantically equivalent) [Dolan and Brockett, 2005], text entailment judgement (determine whether a hypothesis can be logically inferred or entailed from a premise) [Bentivogli and Giampiccolo, 2011; Williams et al., 2018], grounded commonsense inference (determine whether an event is likely to happen given its context) [Zellers et al., 2018], and question-answering inference (determine whether an answer corresponds to a given question).

- **Regression.** Instead of generating a label distribution, we can have the prediction network output a real-valued score. For example, by adding a Sigmoid layer to the prediction network, the system can be employed to compute the similarity between two given sentences. The architecture is the same as that of BERT-based classification systems, with only the change of the output layer.



For training or fine-tuning, we can minimize the regression loss of the model output as usual.

- **Sequence Labeling.** Sequence labeling is a machine learning approach applicable to a wide range of NLP problems. This approach assigns a label to each token in an input sequence, and some linguistic annotations can then be derived from this sequence of labels. An example of sequence labeling in NLP is part-of-speech (POS) tagging. We label each word in a sentence with its corresponding POS tag. Another example is named entity recognition (NER) in which we label each word with an NER tag, and named entities are identified using these tags. See below for an illustration of the model architecture for NER.



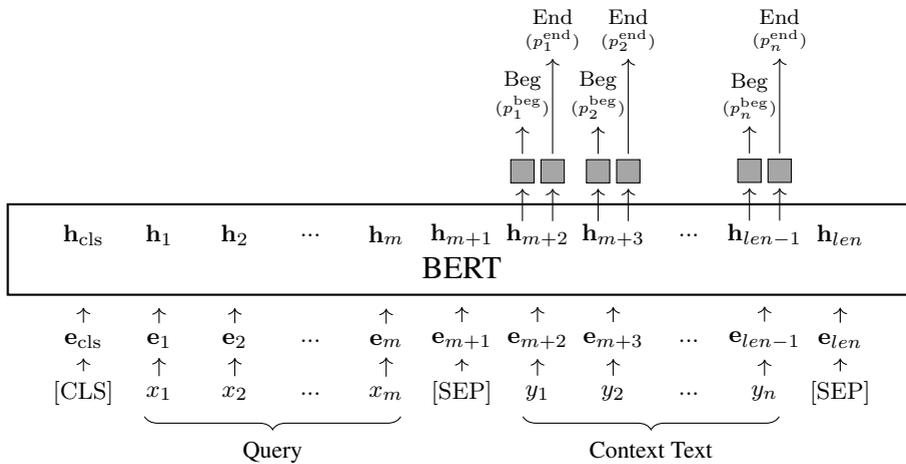
Here  $\{B, I, O\}$  is the tag set of NER. For example, B-ORG means the beginning of an organization, I-ORG means the word is inside an organization, and O means the word does not belong to any named entity. This NER model can output a distribution over the tag set at each position, denoted as  $\mathbf{p}_i$ . The training or fine-tuning of the model can be performed over these distributions  $\{\mathbf{p}_1, \dots, \mathbf{p}_m\}$ . For example, suppose  $p_i(\text{tag}_i)$  is the probability of the correct tag at position  $i$ . The training loss can be defined to be the negative likelihood

$$\text{Loss} = -\frac{1}{m} \sum_{i=1}^m \log p_i(\text{tag}_i) \quad (7.23)$$

Finding the best label sequence given a trained NER model is a well-studied issue in

NLP. This is often achieved via dynamic programming, which, in the context of path finding over a lattice, has linear complexity [Huang, 2009].

- **Span Prediction.** Some NLP tasks require predicting a span in a text. A common example is reading comprehension. In this task, we are given a query  $x_1 \dots x_m$  and a context text  $y_1 \dots y_n$ . The goal is to identify a continuous span in  $y_1 \dots y_n$  that best answers the query. This problem can be framed as a sequence labeling-like task in which we predict a label for each  $y_j$  to indicate the beginning or ending of the span. Following Seo et al. [2017], we add two networks on top of the BERT output for  $y_j$ : one for generating the probability of  $y_j$  being the beginning of the span (denoted by  $p_j^{\text{beg}}$ ), and one for generating the probability of  $y_j$  being the ending of the span (denoted by  $p_j^{\text{end}}$ ). The resulting model architecture is shown as follows



We pack the query and context text together to obtain the input sequence. The prediction networks are only applied to outputs for the context text, generating the probabilities  $p_j^{\text{beg}}$  and  $p_j^{\text{end}}$  at each position. The loss can be computed by summing the negative log likelihoods of the two models across the entire context text.

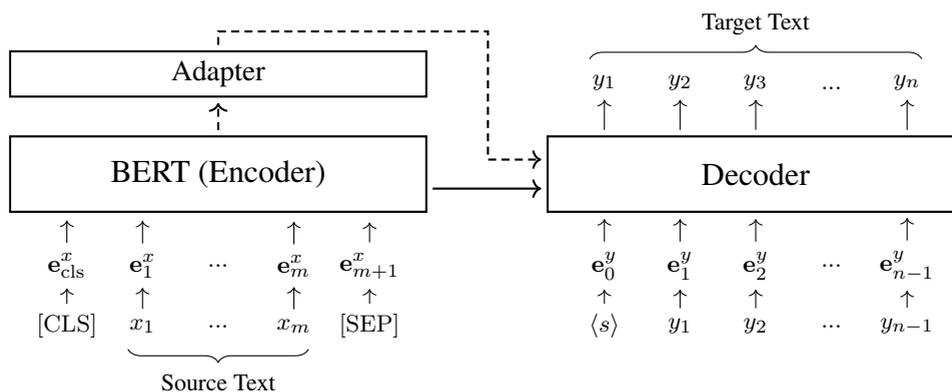
$$\text{Loss} = -\frac{1}{n} \sum_{j=1}^n (\log p_j^{\text{beg}} + \log p_j^{\text{end}}) \quad (7.24)$$

At test time, we search for the best span by

$$(\hat{j}_1, \hat{j}_2) = \arg \max_{1 \leq j_1 \leq j_2 \leq n} (\log p_{j_1}^{\text{beg}} + \log p_{j_2}^{\text{end}}) \quad (7.25)$$

- **Encoding for Encoder-Decoder Models.** While our focus in this section has been primarily on language understanding problems, it is worth noting that BERT models can be applied to a broader range of NLP tasks. In fact, BERT models can be used in all the scenarios where we need to encode a piece of text. One application that we have not mentioned is text generation which includes a range of tasks such as machine translation, summarization, question answering, and dialogue generation. These tasks

can be formulated as sequence-to-sequence problems: we use an encoder to represent the source text, and a decoder to generate the corresponding target text. A straightforward method to apply BERT models is to consider them as encoders. Before fine-tuning, we can initialize the parameters of the encoder with those from a pre-trained BERT model. Then, the encoder-decoder model can be fine-tuned on pairs of texts as usual. The following shows the architecture of a neural machine translation system where a BERT model is applied on the source side.



Here  $x_1 \dots x_m$  denotes the source sequence,  $y_1 \dots y_n$  denotes the target sequence,  $e_1^x \dots e_m^x$  denotes the embedding sequence of  $x_1 \dots x_m$ , and  $e_1^y \dots e_n^y$  denotes the embedding sequence of  $y_1 \dots y_n$ . The adapter, which is optional, maps the output of the BERT model to the form that is better suited to the decoder.

Fine-tuning BERT models is a complicated engineering problem, influenced by many factors, such as the amount of fine-tuning data, the model size, and the optimizer used in fine-tuning. In general, we wish to fine-tune these models sufficiently so that they can perform well in the downstream tasks. However, fine-tuning BERT models for specific tasks may lead to overfitting, which in turn reduces their ability to generalize to other tasks. For example, suppose we have a BERT model that performs well on a particular task. If we then fine-tune it for new tasks, this may decrease its performance on the original task. This problem is related to the **catastrophic forgetting** problem in continual training, where a neural network forgets previously learned information when updated on new samples. In practical applications, a common way to alleviate catastrophic forgetting is to add some old data into fine-tuning and train the model with more diverse data. Also, one may use methods specialized to catastrophic forgetting, such as experience replay [Rolnick et al., 2019] and elastic weight consolidation [Kirkpatrick et al., 2017]. The interested reader can refer to some surveys for more detailed discussions of this issue in continual learning [Parisi et al., 2019; Wang et al., 2023a;f].

## 7.5 Summary

In this chapter we have discussed the general idea of pre-training in NLP. In particular, we have discussed self-supervised pre-training and its application to encoder-only, decoder-only, and

encoder-decoder architectures. Moreover, we have presented and compared a variety of pre-training tasks for these architectures. As an example, BERT is used to illustrate how sequence models are pre-trained via masked language modeling and applied to different downstream tasks.

Recent years have shown remarkable progress in NLP, led by the large-scale use of self-supervised pre-training. And sweeping advances are being made across many tasks, not only in NLP but also in computer vision and other areas of AI. One idea behind these advances is that a significant amount of knowledge about the world can be learned by simply training these AI systems on huge amounts of unlabeled data. For example, a language model can learn some general knowledge of a language by repeatedly predicting masked words in large-scale text. As a result, this pre-trained language model can serve as a foundation model, which can be easily adapted to address specific downstream NLP tasks. This paradigm shift in NLP has enabled the development of incredibly powerful systems for language understanding, generation, and reasoning [Manning, 2022]. However, it is important to recognize that we are still in the early stages of creating truly intelligent systems, and there is a long way to go. Nevertheless, large-scale pre-training has opened a door to intelligent systems that researchers have long aspired to develop, though several key research areas remain open for exploration, such as learning intelligence efficiently using reasonably small-sized data and acquiring complex reasoning and planning abilities.

Note that this chapter is mostly introductory and cannot cover all aspects of pre-training. For example, there are many methods to fine-tune a pre-trained model, offering different ways to better adapt the model to diverse situations. Moreover, large language models, which are considered one of the most significant achievements in AI in recent years, are skipped in this section. We leave the discussion of these topics to the following chapters.

# Chapter 8

## Generative Models

One of the most significant advances in NLP in recent years might be the development of large language models (LLMs). This has helped create systems that can understand and generate natural languages like humans. These systems have even been found to be able to reason, which is considered a very challenging AI problem. With these achievements, NLP made big strides and entered a new era of research in which difficult problems are being solved, such as building conversational systems that can communicate with humans smoothly.

The concept of language modeling or probabilistic language modeling dates back to early experiments conducted by [Shannon \[1951\]](#). In his work, a language model was designed to estimate the predictability of English — *how well can the next letter of a text be predicted when the preceding  $N$  letters are known*. Although Shannon’s experiments were preliminary, the fundamental goals and methods of language modeling have remained largely unchanged over the decades since then. For quite a long period, particularly before 2010, the dominant approach to language modeling was the  $n$ -gram approach [[Jurafsky and Martin, 2008](#)]. In  $n$ -gram language modeling, we estimate the probability of a word given its preceding  $n - 1$  words, and thus the probability of a sequence can be approximated by the product of a series of  $n$ -gram probabilities. These probabilities are typically estimated by collecting smoothed relative counts of  $n$ -grams in text. While such an approach is straightforward and simple, it has been extensively used in NLP. For example, the success of modern statistical speech recognition and machine translation systems has largely depended on the utilization of  $n$ -gram language models [[Jelinek, 1998](#); [Koehn, 2010](#)].

Applying neural networks to language modeling has long been attractive, but a real breakthrough appeared as deep learning techniques advanced. A widely cited study is [Bengio et al. \[2003a\]](#)’s work where  $n$ -gram probabilities are modeled via a feed-forward network and learned by training the network in an end-to-end fashion. A by-product of this neural language model is the distributed representations of words, known as word embeddings. Rather than representing words as discrete variables, word embeddings map words into low-dimensional real-valued vectors, making it possible to compute the meanings of words and word  $n$ -grams in a continuous representation space. As a result, language models are no longer burdened with the curse of dimensionality, but can represent exponentially many  $n$ -grams via a compact

and dense neural model.

The idea of learning word representations through neural language models inspired subsequent research in representation learning in NLP. However, this approach did not attract significant interest in developing NLP systems in the first few years after its proposal. Starting in about 2012, though, advances were made in learning word embeddings from large-scale text via simple word prediction tasks. Several methods, such as Word2Vec, were proposed to effectively learn such embeddings, which were then successfully applied in a variety of NLP systems [Mikolov et al., 2013a;c]. As a result of these advances, researchers began to think of learning representations of sequences using more powerful language models, such as LSTM-based models [Sutskever et al., 2014; Peters et al., 2018]. And further progress and interest in sequence representation exploded after Transformer was proposed. Alongside the rise of Transformer, the concept of language modeling was generalized to encompass models that learn to predict words in various ways. Many powerful Transformer-based models were pre-trained using these word prediction tasks, and successfully applied to a variety of downstream tasks [Devlin et al., 2019].

Indeed, training language models on large-scale data has led NLP research to exciting times. While language modeling has long been seen as a foundational technique with no direct link to the goals of artificial intelligence that researchers had hoped for, it helps us see the emergence of intelligent systems that can learn a certain degree of general knowledge from repeatedly predicting words in text. Recent research demonstrates that a single, well-trained LLM can handle a large number of tasks and generalize to perform new tasks with a small adaptation effort [Bubeck et al., 2023]. This suggests a step towards more advanced forms of artificial intelligence, and inspires further exploration into developing more powerful language models as foundation models.

In this chapter, we consider the basic concepts of generative LLMs. For simplicity, we use the terms *large language models* or *LLMs* to refer to generative models like GPT, though this term can broadly cover other types of models like BERT. We begin by giving a general introduction to LLMs, including the key steps of building such models. We then discuss two scaling issues of LLMs: how LLMs are trained at scale, and how LLMs can be improved to handle very long texts. Finally, we give a summary of these discussions.

## 8.1 A Brief Introduction to LLMs

In this section we give an introduction to the basic ideas of LLMs as required for the rest of this chapter and the following chapters. We will use terms *word* and *token* interchangeably. Both of them refer to the basic units used in language modeling, though their original meanings are different.

Before presenting details, let us first consider how language models work. The goal of language modeling is to predict the probability of a sequence of tokens occurring. Let  $\{x_0, x_1, \dots, x_m\}$  be a sequence of tokens, where  $x_0$  is the start symbol  $\langle s \rangle$  (or  $\langle \text{SOS} \rangle$ )<sup>1</sup>. The

---

<sup>1</sup>The start symbol can also be  $\langle \text{CLS} \rangle$  following BERT models.

probability of this sequence can be defined using the chain rule

$$\begin{aligned}\Pr(x_0, \dots, x_m) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdots \Pr(x_m|x_0, \dots, x_{m-1}) \\ &= \prod_{i=0}^m \Pr(x_i|x_0, \dots, x_{i-1})\end{aligned}\quad (8.1)$$

or alternatively in a logarithmic form

$$\log \Pr(x_0, \dots, x_m) = \sum_{i=0}^m \log \Pr(x_i|x_0, \dots, x_{i-1})\quad (8.2)$$

Here  $\Pr(x_i|x_0, \dots, x_{i-1})$  is the probability of the token  $x_i$  given all its previous tokens  $\{x_0, \dots, x_{i-1}\}$ <sup>2</sup>. In the era of deep learning, a typical approach to language modeling is to estimate this probability using a deep neural network. Neural networks trained to accomplish this task receive a sequence of tokens  $x_0, \dots, x_{i-1}$  and produce a distribution over the vocabulary  $\mathcal{V}$  (denoted by  $\Pr(\cdot|x_0, \dots, x_{i-1})$ ). The probability  $\Pr(x_i|x_0, \dots, x_{i-1})$  is the value of the  $i$ -th entry of  $\Pr(\cdot|x_0, \dots, x_{i-1})$ .

When applying a trained language model, a common task is to find the most likely token given its previous context tokens. This token prediction task can be described as

$$\hat{x}_i = \arg \max_{x_i \in \mathcal{V}} \Pr(x_i|x_0, \dots, x_{i-1})\quad (8.3)$$

We can perform word prediction multiple times to generate a continuous text: each time we predict the best token  $\hat{x}_i$ , and then add this predicted token to the context for predicting the next token  $\hat{x}_{i+1}$ . This results in a left-to-right generation process implementing Eqs. (8.1) and (8.2). To illustrate, consider the generation of the following three words given the prefix ‘ $\langle s \rangle$  a’, as shown in Table 8.1. Now we discuss how LLMs are constructed, trained, and applied.

### 8.1.1 Decoder-only Transformers

As is standard practice, the input of a language model is a sequence of tokens (denoted by  $\{x_0, \dots, x_{m-1}\}$ ). For each step, an output token is generated, shifting the sequence one position forward for the next prediction. To do this, the language model outputs a distribution  $\Pr(\cdot|x_0, \dots, x_{i-1})$  at each position  $i$ , and the token  $x_i$  is selected according to this distribution. This model is trained by maximizing the log likelihood  $\sum_{i=1}^m \log \Pr(x_i|x_0, \dots, x_{i-1})$ <sup>3</sup>.

Here, we focus on the decoder-only Transformer architecture, as it is one of the most popular model architectures used in LLMs. The input sequence of tokens is represented by a sequence of  $d_e$ -dimensional vectors  $\{\mathbf{e}_0, \dots, \mathbf{e}_{m-1}\}$ .  $\mathbf{e}_i$  is the sum of the token embedding of  $x_i$  and the positional embedding of  $i$ . The major body of the model is a stack of Transformer

<sup>2</sup>We assume that when  $i = 0$ ,  $\Pr(x_i|x_0, \dots, x_{i-1}) = \Pr(x_0) = 1$ . Hence  $\Pr(x_0, \dots, x_m) = \Pr(x_0) \Pr(x_1, \dots, x_m|x_0) = \Pr(x_1, \dots, x_m|x_0)$ .

<sup>3</sup>Note that  $\sum_{i=1}^m \log \Pr(x_i|x_0, \dots, x_{i-1}) = \sum_{i=0}^m \log \Pr(x_i|x_0, \dots, x_{i-1})$  since  $\log \Pr(x_0) = 0$ .

Context	Predict	Decision Rule	Sequence Probability
$\langle s \rangle a$	$b$	$\arg \max_{x_2 \in V} \Pr(x_2   \langle s \rangle a)$	$\Pr(\langle s \rangle) \cdot \Pr(a   \langle s \rangle) \cdot \Pr(b   \langle s \rangle a)$
$\langle s \rangle a b$	$c$	$\arg \max_{x_3 \in V} \Pr(x_3   \langle s \rangle a b)$	$\Pr(\langle s \rangle) \cdot \Pr(a   \langle s \rangle) \cdot \Pr(b   \langle s \rangle a) \cdot \Pr(c   \langle s \rangle a b)$
$\langle s \rangle a b c$	$d$	$\arg \max_{x_4 \in V} \Pr(x_4   \langle s \rangle a b c)$	$\Pr(\langle s \rangle) \cdot \Pr(a   \langle s \rangle) \cdot \Pr(b   \langle s \rangle a) \cdot \Pr(c   \langle s \rangle a b) \cdot \Pr(d   \langle s \rangle a b c)$

Table 8.1: Illustration of generating the three tokens  $b c d$  given the prefix  $\langle s \rangle a$  via a language model. In each step, the model picks a token  $x_i$  from  $V$  so that  $\Pr(x_i | x_0, \dots, x_{i-1})$  is maximized. This token is then appended to the end of the context sequence. In the next step, we repeat the same process, but based on the new context.

blocks (or layers). Each Transformer block has two stacked sub-layers, one for self-attention modeling and one for FFN modeling. These sub-layers can be defined using the post-norm architecture

$$\text{output} = \text{LNorm}(F(\text{input}) + \text{input}) \quad (8.4)$$

or the pre-norm architecture

$$\text{output} = \text{LNorm}(F(\text{input})) + \text{input} \quad (8.5)$$

where input and output denote the input and output, both being an  $m \times d$  matrix. The  $i$ -th rows of input and output can be seen as contextual representations of the  $i$ -th token in the sequence.

$F(\cdot)$  is the core function of a sub-layer. For FFN sub-layers,  $F(\cdot)$  is a multi-layer FFN. For self-attention sub-layers,  $F(\cdot)$  is a multi-head self-attention function. In general, self-attention is expressed in a form of QKV attention

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \mathbf{Mask}\right)\mathbf{V} \quad (8.6)$$

where  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V} \in \mathbb{R}^{m \times d}$  are the queries, keys, and values, respectively. It is important to note that only previous tokens are considered when predicting a token. So a masking variable  $\mathbf{Mask} \in \mathbb{R}^{m \times m}$  is incorporated into self-attention to achieve this. The entry  $(i, k)$  of  $\mathbf{Mask}$  has a value of 0 if  $i \leq k$ , and a value of  $-\text{inf}$  otherwise.

Given a representation  $\mathbf{H} \in \mathbb{R}^{m \times d}$ , the multi-head self-attention function can be defined as

$$F(\mathbf{H}) = \text{Merge}(\text{head}_1, \dots, \text{head}_\tau) \mathbf{W}^{\text{head}} \quad (8.7)$$

where  $\text{Merge}(\cdot)$  represents a concatenation of its inputs, and  $\mathbf{W}^{\text{head}} \in \mathbb{R}^{d \times d}$  represents a

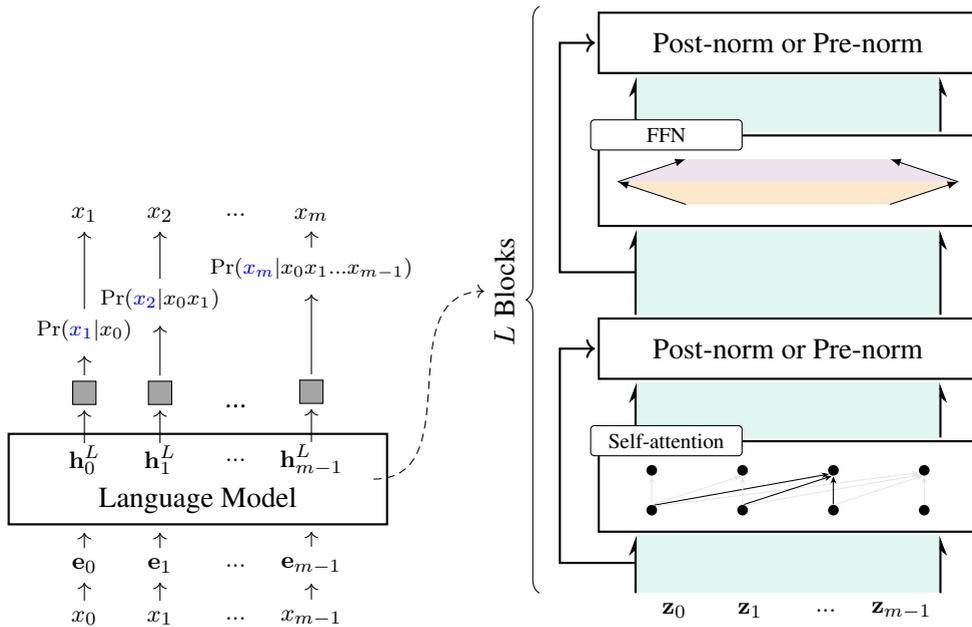


Figure 8.1: The Transformer-decoder architecture for language modeling. The central components are  $L$  stacked Transformer blocks, each comprising a self-attention sub-layer and an FFN sub-layer. To prevent the model from accessing the right-context, a masking variable is incorporated into self-attention. The output layer uses a Softmax function to generate a probability distribution for the next token, given the sequence of previous tokens. During inference, the model takes the previously predicted token to predict the next one, repeating this process until the end of the sequence is reached.  $\{z_0, \dots, z_{m-1}\}$  denote the inputs of a Transformer block, and  $\{h_0^L, \dots, h_{m-1}^L\}$  denote the outputs of the last Transformer block.

parameter matrix.  $\text{head}_j$  is the output of QKV attention on a sub-space of representation

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{Q}^{[j]}, \mathbf{K}^{[j]}, \mathbf{V}^{[j]}) \quad (8.8)$$

$\mathbf{Q}^{[j]}$ ,  $\mathbf{K}^{[j]}$ , and  $\mathbf{V}^{[j]}$  are the queries, keys, and values projected onto the  $j$ -th sub-space via linear transformations

$$\mathbf{Q}^{[j]} = \mathbf{H}\mathbf{W}_j^q \quad (8.9)$$

$$\mathbf{K}^{[j]} = \mathbf{H}\mathbf{W}_j^k \quad (8.10)$$

$$\mathbf{V}^{[j]} = \mathbf{H}\mathbf{W}_j^v \quad (8.11)$$

where  $\mathbf{W}_j^q$ ,  $\mathbf{W}_j^k$ , and  $\mathbf{W}_j^v \in \mathbb{R}^{d \times \frac{d}{\tau}}$  are the parameter matrices of the transformations.

Suppose we have  $L$  Transformer blocks. A Softmax layer is built on top of the output of the last block. The Softmax layer outputs a sequence of  $m$  distributions over the vocabulary,

like this

$$\begin{bmatrix} \Pr(\cdot|x_0, \dots, x_{m-1}) \\ \vdots \\ \Pr(\cdot|x_0, x_1) \\ \Pr(\cdot|x_0) \end{bmatrix} = \text{Softmax}(\mathbf{H}^L \mathbf{W}^o) \quad (8.12)$$

where  $\mathbf{H}^L$  is the output of the last Transformer block, and  $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$  is the parameter matrix.

Figure 8.1 shows the Transformer architecture for language modeling. Applying this language model follows an autoregressive process. Each time the language model takes a token  $x_{i-1}$  as input and predicts a token  $x_i$  that maximizes the probability  $\Pr(x_i|x_0, \dots, x_{i-1})$ . It is important to note that, despite different implementation details, many LLMs share the same architecture described above. These models are called large because both their depth and width are significant. Table 8.2 shows the model sizes for a few LLMs, as well as their model setups.

### 8.1.2 Training LLMs

Now suppose that we are given a training set  $\mathcal{D}$  comprising  $K$  sequences. The log-likelihood of each sequence  $\mathbf{x} = x_0 \dots x_m$  in  $\mathcal{D}$  can be calculated using a language model

$$\mathcal{L}_\theta(\mathbf{x}) = \sum_{i=1}^m \log \Pr_\theta(x_i|x_0, \dots, x_{i-1}) \quad (8.13)$$

Here the subscript  $\theta$  affixed to  $\mathcal{L}(\cdot)$  and  $\Pr(\cdot)$  denotes the parameters of the language model. Then, the objective of maximum likelihood training is defined as

$$\hat{\theta} = \arg \max_{\theta} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}_\theta(\mathbf{x}) \quad (8.14)$$

Training Transformer-based language models with the above objective is commonly viewed as a standard optimization process for neural networks. This can be achieved using gradient descent algorithms, which are widely supported by off-the-shelf deep learning toolkits. Somewhat surprisingly, better results were continuously yielded as language models were evolved into more computationally intensive models and trained on larger datasets [Kaplan et al., 2020]. These successes have led NLP researchers to continue increasing both the training data and model size in order to build more powerful language models.

However, as language models become larger, we confront new training challenges, which significantly change the problem compared to training relatively small models. One of these challenges arises from the need for large-scale distributed systems to manage the data, model parameters, training routines, and so on. Developing and maintaining such systems requires a significant amount of work in both software and hardware engineering, as well as expertise in deep learning. A related issue is that when the training is scaled up, we need more computing resources to ensure the training process can be completed in an acceptable time. For example,

LLM	# of Parameters	Depth $L$	Width $d$	# of Heads (Q/KV)
GPT-1 [Radford et al., 2018]	0.117B	12	768	12/12
GPT-2 [Radford et al., 2019]	1.5B	48	1,600	25/25
GPT-3 [Brown et al., 2020]	175B	96	12,288	96/96
LLaMA2 [Touvron et al., 2023b]	7B	32	4,096	32/32
	13B	40	5,120	40/40
	70B	80	8,192	64/64
LLaMA3/3.1 [Dubey et al., 2024]	8B	32	4,096	32/8
	70B	80	8,192	64/8
	405B	126	16,384	128/8
Gemma2 [Team et al., 2024]	2B	26	2,304	8/4
	9B	42	3,584	16/8
	37B	46	4,608	32/16
Qwen2.5 [Yang et al., 2024]	0.5B	24	896	14/2
	7B	28	3,584	28/4
	72B	80	8,192	64/8
DeepSeek-V3 [Liu et al., 2024a]	671B	61	7,168	128/128
Falcon [Penedo et al., 2023]	7B	32	4,544	71/71
	40B	60	8,192	128/128
	180B	80	14,848	232/232
Mistral [Jiang et al., 2023a]	7B	32	4,096	32/32

Table 8.2: Comparison of some LLMs in terms of model size, model depth, model width, and number of heads ( $a/b$  means  $a$  heads for queries and  $b$  heads for both keys and values).

it generally requires hundreds or thousands of GPUs to train an LLM with tens of billions of parameters from scratch. This requirement drastically increases the cost of training such models, especially considering that many training runs are needed as these models are developed. Also, from the perspective of deep learning, the training process can become unstable if the neural networks are very deep and/or the model size is very large. In response, we typically need to modify the model architecture to adapt LLMs to large-scale training. In Section 8.2 we will present more discussions on these issues.

### 8.1.3 Fine-tuning LLMs

Once we have pre-trained an LLM, we can then apply it to perform various NLP tasks. Traditionally language models are used as components of other systems, for example, they are widely applied to score translations in statistical machine translation systems. By contrast, in generative AI, LLMs are considered complete systems and are employed to address NLP problems by making use of their generation nature. A common approach is to describe the task

we want to address in text and then prompt LLMs to generate text based on this description. This is a standard text generation task where we continue or complete the text starting from a given context.

More formally, let  $\mathbf{x} = x_0 \dots x_m$  denote a token sequence of context given by users, and  $\mathbf{y} = y_1 \dots y_n$  denote a token sequence following the context. Then, the inference of LLMs can be defined as a problem of finding the most likely sequence  $\mathbf{y}$  based on  $\mathbf{x}$ :

$$\begin{aligned} \hat{\mathbf{y}} &= \underset{\mathbf{y}}{\operatorname{arg\,max}} \log \Pr(\mathbf{y}|\mathbf{x}) \\ &= \underset{\mathbf{y}}{\operatorname{arg\,max}} \sum_{i=1}^n \log \Pr(y_i|x_0, \dots, x_m, y_1, \dots, y_{i-1}) \end{aligned} \quad (8.15)$$

Here  $\sum_{i=1}^n \log \Pr(y_i|x_0, \dots, x_m, y_1, \dots, y_{i-1})$  essentially expresses the same thing as the right-hand side of Eq. (8.2). It models the log probability of predicting tokens from position  $m + 1$ , rather than position 0. Throughout this chapter and subsequent ones, we will employ separate variables  $\mathbf{x}$  and  $\mathbf{y}$  to distinguish the input and output of an LLM, though they can be seen as sub-sequences from the same sequence. By adopting such notation, we see that the form of the above equation closely resembles those used in other text generation models in NLP, such as neural machine translation models.

To illustrate how LLMs are applied, consider the problem of determining the grammaticality for a given sentence. We can define a template like this

```
{*sentence*}
Question: Is this sentence grammatically correct?
Answer: ____
```

Here `__` represents the text we intend to generate. `{*sentence*}` is a placeholder variable that will be replaced by the actual sentence provided by the users. For example, suppose we have a sentence “*John seems happy today.*”. We can replace the `{*sentence*}` in the template with this sentence to have an input to the language model

```
John seems happy today.
Question: Is this sentence grammatically correct?
Answer: ____
```

To perform the task, the language model is given the context  $\mathbf{x}$  = “John seems happy today.  
.\n Question : Is this sentence grammatically correct?.\n Answer :”<sup>4</sup>. It then generates the following text as the answer, based on the context. For example, the language model may output “Yes” (i.e.,  $\mathbf{y}$  = “Yes”) if this text is the one with the maximum probability of prediction given this context.

<sup>4</sup>\n is a special character used for line breaks.

Likewise, we can define more templates to address other tasks. For example, we can translate an English sentence into Chinese using the following template

```
{*sentence*}
Question: What is the Chinese translation of this English sentence?
Answer: _____
```

or using an instruction-like template

```
{*sentence*}
Translate this sentence from English into Chinese.
_____
```

or using a code-like template.

```
[src-lang] = English [tgt-lang] = Chinese [input] = {*sentence*}
[output] = _____
```

The above templates provide a simple but effective method to “prompt” a single LLM to perform various tasks without adapting the structure of the model. However, this approach requires that the LLM can recognize and follow the instructions or questions. One way to do this is to incorporate training samples with instructions and their corresponding responses into the pre-training dataset. While this method is straightforward, building and training LLMs from scratch is computationally expensive. Moreover, making instruction-following data effective for pre-training requires a significant amount of such data, but collecting large-scale labeled data for all tasks of interest is very difficult.

A second method, which has been a de facto standard in recent research, is to adapt LLMs via fine-tuning. As such, the token prediction ability learned in the pre-training phase can be generalized to accomplish new tasks. The idea behind fine-tuning is that some general knowledge of language has been acquired in pre-training, but we need a mechanism to activate this knowledge for applying it to new tasks. To achieve this, we can slightly fine-tune the model parameters using instruction-following data. This approach is called **instruction fine-tuning**.

An instruction fine-tuning sample, which is represented by a sequence of tokens, can be seen as a tuple consisting of an input and the desired output. Here, the input includes instructions, system information (or system prefix), and any other user-provided information<sup>5</sup>. To illustrate, consider the following examples (blue text = input and underlined text = output).

<sup>5</sup>System information refers to a sequence of tokens added at the beginning of an input in order to guide the behavior of an LLM, such as, *you are a helpful assistant and should not output toxic content*.

Read the following sentence, and answer whether it is grammatically correct.  
LLMs are powerful models but are expensive to build.

Yes

LLMs are powerful models but are expensive to build.

Does this sentence make sense grammatically?

Answer Option

- Yes

- No

Yes

Yes or No? Yes = the following sentence is grammatically correct. No = it contains grammatical errors.

LLMs are powerful models but are expensive to build.

Yes

All these samples describe the same binary classification task, but with different instructions. To increase the diversity in the fine-tuning data and make LLMs generalize, we can define more tasks using instructions.

Write a short story about three characters.

Character 1: A boy whose name is Andy.

Character 2: Andy's dog.

Character 3: Andy's friend, Jane.

In the heart of a bustling city, Andy, his loyal dog Rocket, and his friend Jane discovered a hidden garden beneath the glow of twilight. Surrounded by ancient trees and whispering winds, they shared dreams and secrets, forging an unbreakable bond. As the night deepened, they made a pact under the stars to always find magic in the mundane. Together, they realized that true adventure lay in the moments spent with friends, and every day was a treasure waiting to be discovered.

Show me a recipe for making ice cream.

Ingredients:

2 cups heavy cream; 1 cup milk; 3/4 cup sugar; 2 tsp vanilla extract;  
Pinch of salt.

Steps:

1. Whisk all ingredients until sugar dissolves.
2. Chill mixture for 2 hours in the fridge.
3. Freeze in a shallow dish, stirring every 30 minutes, 3-4 times.
4. Freeze solid for 2 hours.
5. Serve your homemade ice cream.

If you buy 5 apples and each apple costs \$1.20, how much do you spend in total?

\$6.00

Write a Python program to calculate the sum of squares of the following numbers.

1, 2, 10, -9, 78

```
numbers = [1,2,10,-9,78]  
sum_of_squares = sum(x**2 for x in numbers)  
print(sum_of_squares)
```

To acquire instruction-following abilities, a certain amount of fine-tuning data is required. This data may include diverse instructions and possible responses. It has been found that scaling the number of fine-tuning tasks is beneficial for improving the performance of LLMs [Chung et al., 2022]. Note that although more fine-tuning data is favorable, the amount of this data is generally orders of magnitude smaller than that of the pre-training data. For example, LLMs can be fine-tuned with tens or hundreds of thousands of samples, or even fewer if these samples are of high quality [Zhou et al., 2023a; Chen et al., 2023b], whereas pre-training such models may require billions or trillions of tokens, resulting in significantly larger computational demands and longer training times [Touvron et al., 2023a].

It is also worth noting that we should not expect the fine-tuning data to cover all the downstream tasks to which we intend to apply LLMs. A common understanding of how the pre-training + fine-tuning approach works is that LLMs have gained knowledge for understanding instructions and generating responses in the pre-training phase. However, these abilities are not fully activated until we introduce some form of supervision. The general instruction-following behavior emerges as we fine-tune the models with a relatively small amount of labeled data.

As a result, we can achieve some level of **zero-shot learning**: the fine-tuned models can handle new tasks that they have not been explicitly trained or fine-tuned for [Sanh et al., 2022; Wei et al., 2022a]. This zero-shot learning ability distinguishes generative LLMs from earlier pre-trained models like BERT, which are primarily fine-tuned for specific tasks.

Once we have prepared a collection of instruction-described data, the fine-tuning process is relatively simple. This process can be viewed as a standard training process as pre-training, but on a much smaller training dataset. Let  $\mathcal{D}_{\text{tune}}$  be the fine-tuning dataset and  $\hat{\theta}$  be the model parameters optimized via pre-training. We can modify Eq. (8.14) to obtain the objective of fine-tuning

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \sum_{\text{sample} \in \mathcal{D}_{\text{tune}}} \mathcal{L}_{\hat{\theta}^+}(\text{sample}) \quad (8.16)$$

Here  $\tilde{\theta}$  denotes the optimal parameters. The use of notation  $\hat{\theta}^+$  means that the fine-tuning starts with the pre-trained parameters  $\hat{\theta}$ .

For each  $\text{sample} \in \mathcal{D}_{\text{tune}}$ , we divide it into an input segment  $\mathbf{x}_{\text{sample}}$  and an output segment  $\mathbf{y}_{\text{sample}}$ , that is,

$$\text{sample} = [\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}] \quad (8.17)$$

We then define the loss function to be

$$\mathcal{L}_{\hat{\theta}^+}(\text{sample}) = -\log \Pr_{\hat{\theta}^+}(\mathbf{y}_{\text{sample}} | \mathbf{x}_{\text{sample}}) \quad (8.18)$$

In other words, we compute the loss over the sub-sequence  $\mathbf{y}_{\text{sample}}$ , rather than the entire sequence. In a practical implementation of back-propagation for this equation, the sequence  $[\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}]$  is constructed in the forward pass as usual. However, in the backward pass, error gradients are propagated back only through the parts of the network that correspond to  $\mathbf{y}_{\text{sample}}$ , leaving the rest of the network unchanged. As an example, consider a sequence

$$\underbrace{\langle s \rangle \text{ Square this number . 2 .}}_{\text{Context (Input)}} \quad \underbrace{\text{The result is 4 .}}_{\text{Prediction (Output)}}$$

The loss is calculated and back propagated only for The result is 4 .

Instruction fine-tuning also requires substantial engineering work. In order to achieve satisfactory results, one may experiment with different settings of the learning rate, batch size, number of fine-tuning steps, and so on. This typically requires many fine-tuning runs and evaluations. The cost and experimental effort of fine-tuning remain critical and should not be overlooked, though they are much lower than those of the pre-training phase.

While we focus on instruction fine-tuning for an illustrative example here, fine-tuning techniques play an important role in developing various LLMs and are more widely used. Examples include fine-tuning LLMs as chatbots using dialog data, and adapting these models to handle very long sequences. The wide application of fine-tuning has led researchers to improve these techniques, such as designing more efficient fine-tuning algorithms. While the

research on fine-tuning is fruitful, in this section we just give a flavour of the key steps involved. We will see more detailed discussions on this topic in the following chapters.

#### 8.1.4 Aligning LLMs with the World

Instruction fine-tuning provides a simple way to adapt LLMs to tasks that can be well defined. This problem can broadly be categorized as an **alignment** problem. Here, alignment is referred to as a process of guiding LLMs to behave in ways that align with human intentions. The guidance can come from labeled data, human feedback, or any other form of human preferences. For example, we want LLMs not only to be accurate in following instructions, but also to be unbiased, truthful, and harmless. So we need to supervise the models towards human values and expectations. A common example is that when we ask an LLM how to build a weapon, it may provide a list of key steps to do so if it is not carefully aligned. However, a responsible model should recognize and avoid responding to requests for harmful or illegal information. Alignment in this case is crucial for ensuring that LLMs act responsibly and in accordance with ethical guidelines.

A related concept to alignment is AI safety. One ultimate goal of AI is to build intelligent systems that are safe and socially beneficial. To achieve this goal we should keep these systems robust, secure, and subjective, in any conditions of real-world use, even in conditions of misuse or adverse use. For LLMs, the safety can be increased by aligning them with appropriate human guidance, such as human labeled data and interactions with users during application.

Alignment is difficult as human values and expectations are diverse and shifting. Sometimes, it is hard to describe precisely what humans want, unless we see the response of LLMs to user requests. This makes alignment no longer a problem of tuning LLMs on predefined tasks, but a bigger problem of training them with the interactions with the real world.

As a result of the concerns with controlling AI systems, there has been a surge in research on the alignment issue for LLMs. Typically, two alignment steps are adopted after LLMs are pre-trained on large-scale unlabeled data.

- **Supervised Fine-tuning (SFT).** This involves continuing the training of pre-trained LLMs on new, task-oriented, labelled data. A commonly used SFT technique is instruction fine-tuning. As described in the previous subsection, by learning from instruction-response annotated data, LLMs can align with the intended behaviors for following instructions, thereby becoming capable of performing various instruction-described tasks. Supervised fine-tuning can be seen as following the pre-training + fine-tuning paradigm, and offers a relatively straightforward method to adapt LLMs.
- **Learning from Human Feedback.** After an LLM finishes pre-training and supervised fine-tuning, it can be used to respond to user requests if appropriately prompted. But this model may generate content that is unfactual, biased, or harmful. To make the LLM more aligned with the users, one simple approach is to directly learn from human feedback. For example, given some instructions and inputs provided by the users, experts are asked to evaluate how well the model responds in accordance with their preferences and interests. This feedback is then used to further train the LLM for better alignment.

A typical method for learning from human feedback is to consider it as a reinforcement learning (RL) problem, known as **reinforcement learning from human feedback (RLHF)** [Ouyang et al., 2022]. The RLHF method was initially proposed to address general sequential decision-making problems [Christiano et al., 2017], and was later successfully employed in the development of the GPT series models [Stiennon et al., 2020]. As a reinforcement learning approach, the goal of RLHF is to learn a policy by maximizing some reward from the environment. Specifically, two components are built in RLHF:

- **Agent.** An agent, also called an LM agent, is the LLM that we want to train. This agent operates by interacting with its environment: it receives a text from the environment and outputs another text that is sent back to the environment. The policy of the agent is the function defined by the LLM, that is,  $\Pr(\mathbf{y}|\mathbf{x})$ .
- **Reward Model.** A reward model is a proxy of the environment. Each time the agent produces an output sequence, the reward model assigns this output sequence a numerical score (i.e., the reward). This score tells the agent how good the output sequence is.

In RLHF, we need to perform two learning tasks: 1) reward model learning, which involves training a reward model using human feedback on the output of the agent, and 2) policy learning, which involves optimizing a policy guided by the reward model using reinforcement learning algorithms. Here is a brief outline of the key steps involved in RLHF.

- Build an initial policy using pre-training and instruction fine-tuning.
- Use the policy to generate multiple outputs for each input, and then collect human feedback on these outputs (e.g., comparisons of the outputs).
- Learn a reward model from the human feedback.
- Fine-tune the policy with the supervision from the reward model.

Figure 8.2 shows an overview of RLHF. Given that this section serves only as a brief introduction to concepts of LLMs, a detailed discussion of RLHF techniques will not be included. We instead illustrate the basic ideas behind RLHF using a simple example.

Suppose we have trained an LLM via pre-training and instruction fine-tuning. This LLM is deployed to respond to requests from users. For example, a user may input

How can I live a more environmentally friendly life?

We use the LLM to generate 4 different outputs (denoted by  $\{\mathbf{y}_1, \dots, \mathbf{y}_4\}$ ) by sampling the output space

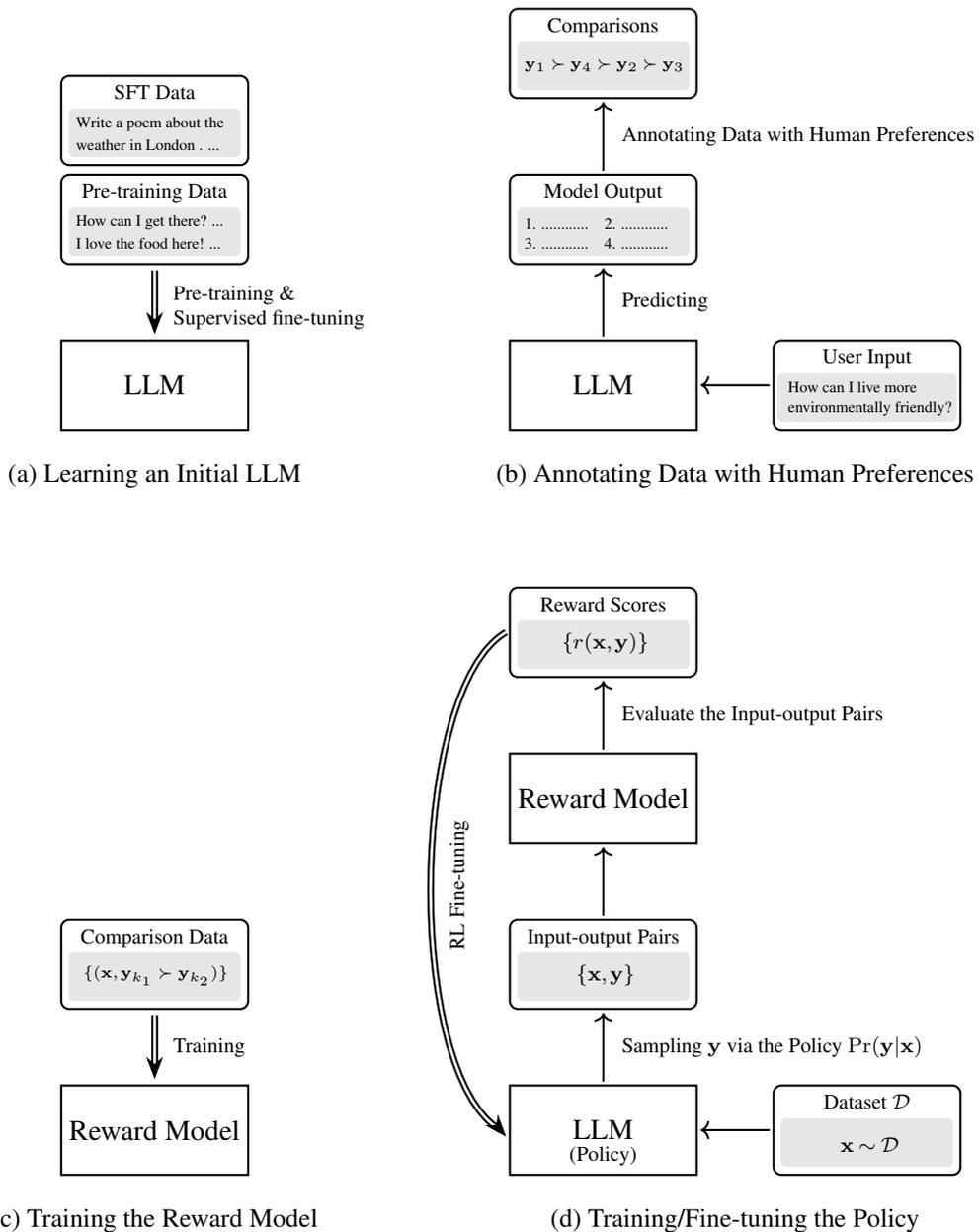


Figure 8.2: An overview of RLHF. There are 4 key steps involved: a) training an initial LLM (i.e., policy) using pre-training and supervised fine-tuning; b) collecting human preference data by ranking the outputs of the LLM; c) training a reward model using the ranking results; d) RL fine-tuning of the policy based on the reward model. Double line arrows mean training or fine-tuning.

Output 1 ( $y_1$ ): Consider switching to an electric vehicle or bicycle instead of traditional cars to reduce carbon emissions and protect our planet.

Output 2 ( $y_2$ ): Adopt a minimalist lifestyle. Own fewer possessions to reduce consumption and the environmental impact of manufacturing and disposal.

Output 3 ( $y_3$ ): Go off-grid. Generate your own renewable energy and collect

We then ask annotators to evaluate these outputs. One straightforward way is to assign a rating score to each output. In this case, the reward model learning problem can be framed as a task of training a regression model. But giving numerical scores to LLM outputs is not an easy task for annotators. It is usually difficult to design an annotation standard that all annotators can agree on and easily follow. An alternative method, which is more popular in the development of LLMs, is to rank these outputs. For example, a possible ranking of the above outputs is

$$\mathbf{y}_1 \succ \mathbf{y}_4 \succ \mathbf{y}_2 \succ \mathbf{y}_3$$

A reward model is then trained using this ranking result. In general, a reward model in RLHF is a language model that shares the same architecture as the target LLM, but with a smaller model size. Given the input  $\mathbf{x}$  and output  $\mathbf{y}_k$ , we concatenate them to form a sequence  $\text{seq}_k = [\mathbf{x}, \mathbf{y}_k]$ . This sequence is processed from left to right using forced decoding. Since each position can only access its left context in language modeling, the output of the top-most Transformer layer at the first position cannot be used as the representation of the sequence. Instead, a special symbol (e.g.,  $\langle s \rangle$ ) is added to the end of the sequence, and the corresponding output of the Transformer layer stack is considered as the representation of the entire sequence. An output layer, such as a linear transformation layer, is built on top of this representation to generate the reward, denoted by  $R(\text{seq}_k)$  or  $R(\mathbf{x}, \mathbf{y}_k)$ .

We train this reward model using ranking loss. For example, a pair-wise ranking loss function can be written in the form

$$\text{Loss}_\omega(\mathcal{D}_r) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2}) \sim \mathcal{D}_r} \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \quad (8.19)$$

where  $\omega$  represents the parameters of the reward model, and  $\mathcal{D}_r$  represents a set of tuples of an input and a pair of outputs.  $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2}) \sim \mathcal{D}_r$  is a sampling operation which draws a sample  $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2})$  from  $\mathcal{D}_r$  with some probability. As an example, suppose we first draw a model input  $\mathbf{x}$  with a uniform distribution and then draw a pair of model outputs with a probability of  $\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}$  given  $\mathbf{x}$  (denoted by  $\Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x})$ ). The corresponding loss function is given by

$$\begin{aligned} & \text{Loss}_\omega(\mathcal{D}_r) \\ &= -\sum \Pr(\mathbf{x}) \cdot \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \\ &= -\frac{1}{K} \sum \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \quad (8.20) \end{aligned}$$

where  $K$  represents the number of model inputs involved in sampling. While the form of these functions may seem complex, their idea is simple: we penalize the model if the predicted ranking of two outputs differs from the human-labeled ranking. By contrast, the model receives a bonus, if the predicted ranking matches the human-labeled ranking.

We can train the reward model by minimizing the above ranking loss

$$\hat{\omega} = \underset{\omega}{\text{arg min}} \text{Loss}_\omega(\mathcal{D}_r) \quad (8.21)$$

The resulting model  $R_{\hat{\omega}}(\cdot)$  can be employed to evaluate any given pair of input and output. Note that although the reward model is trained using a ranking-based objective, it is used for scoring. This allows it to provide continuous supervision signals, which is very beneficial for training other models.

We now turn to the policy learning problem. A commonly adopted objective is to maximize the reward on a set of input-output pairs. Following an analogous form of Eq. (8.16), we obtain a simple training objective for RL fine-tuning

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \mathbb{E}_{(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+}) \sim \mathcal{D}_{\text{rlft}}} R_{\hat{\omega}}(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+}) \quad (8.22)$$

where the optimal parameters  $\tilde{\theta}$  are obtained by fine-tuning the pre-trained parameters  $\hat{\theta}$ .  $\mathcal{D}_{\text{rlft}}$  is the RL fine-tuning dataset. For each sample  $(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+})$ ,  $\mathbf{x}$  is sampled from a prepared dataset of input sequences, and  $\mathbf{y}_{\hat{\theta}^+}$  is sampled from the distribution  $\text{Pr}_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x})$  given by the policy.

In practice, more advanced reinforcement learning algorithms, such as **proximal policy optimization (PPO)**, are often used for achieving more stable training, as well as better performance. We leave the detailed discussion of reinforcement learning algorithms to the following parts of this book where RLHF is extensively used for alignment.

An interesting question arises here: why not consider learning from human preferences as a standard supervised learning problem? This question is closely related to our aforementioned discussion on the difficulty of data annotation. Often, describing human values and goals is challenging, and it is even more difficult for humans to provide outputs that are well aligned. As an alternative, annotating the preferences of a given list of model outputs offers a simpler task. By doing so, we can create a model that understands human preferences, which can then be used as a reward model for training policies. From the perspective of machine learning, RLHF is particularly useful for scenarios where the desired behavior of an agent is difficult to demonstrate but can be easily recognized by humans. Another advantage of RLHF is its ability to explore the sample space. By employing sampling techniques, models trained with reinforcement learning can venture beyond the annotated data set to explore additional samples. This exploratory ability allows RLHF to discover potentially beneficial policies that are not immediately apparent from the labeled data alone.

### 8.1.5 Prompting LLMs

We have so far shown that LLMs can be used to perform various tasks by giving them appropriate prompts. There are no restrictions on these prompts, which can include any information we wish to ask or communicate with LLMs, such as natural language instructions and the context of conversations. Since this approach requires no additional training or tuning, adapting LLMs becomes highly efficient once they are developed. This somewhat influences the paradigms in NLP: we no longer need to develop specific systems for individual tasks but can instead use a single, well-trained LLM to perform different tasks by prompting it. An appealing aspect of LLM prompting arises as a result: users can easily have “customized” systems by designing their own prompts for LLMs. Given the important role played by prompting in LLMs, **prompting engineering** has become a very active area of research in

NLP.

The term *prompt* is used in many different ways in the literature. In this chapter, this term refers to the entire input to LLMs, and so we use the terms *prompt* and *model input* interchangeably. Before discussing prompting further, let us first see a few examples where the prompts are more complex than those presented in the previous subsections. Note that this subsection is not aimed at writing high-quality prompts but rather at highlighting some interesting issues in prompting LLMs.

One of the popular ways to use LLMs is to assign them a “role” played in generating responses. For example, LLMs can act as a psychologist when answering questions.

Please explain what delayed gratification is?

Note, you are a researcher with a deep background in psychology. Your own field of study is developmental psychology in children, with a focus on aspects of control and intelligence. You are very familiar with the concept of delayed gratification and have applied it to your own research. You have published papers on delayed gratification and conducted related experiments.

Delayed gratification is the process of resisting an immediate reward in anticipation of receiving a more valuable reward in the future. It is a significant concept in developmental psychology, especially in relation to how children learn self-control and develop cognitive abilities. Research shows that the ability to delay gratification is associated with positive outcomes in academic performance, social competence, and emotional well-being. Studies and experiments in this area highlight the importance of cultivating this skill early in life to support long-term success and resilience.

Another example is the use of LLMs in detecting and correcting errors such as syntactic or semantic mistakes in text. For an LLM which is trained on both code and natural language data, we may use it for code debugging<sup>6</sup>.

---

<sup>6</sup>In this example, the code is not tokenized for easier reading.

Fix the bugs in this C language program.

```
#include <stdio.h>
int main() {
    printg("Hello, World!")
    return 0;
}
```

1. The function name 'printg' is incorrect . It should be 'printf'.
2. There is a missing semicolon at the end of the 'printf' function call.

Here is the corrected version of the program:

```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

Also, LLMs can be fine-tuned to act as a conversation partner <sup>7</sup>.

Hi! I'm an assistant. How can I help you?

Who won the FIFA World Cup 2022?

Argentina won the FIFA World Cup 2022.

Where was it held?

The 2022 FIFA World Cup was held in Qatar.

How many times has Argentina won the World Cup?

Argentina has won the FIFA World Cup three times.

...

These examples and previous ones have shown that appropriate responses can be generated via prompts involving clear instructions and questions. However, when problem solving requires knowledge that is not explicitly specified, LLMs may make mistakes, even though the instructions are sufficiently clear and precise. A family of challenging tasks for LLMs involves arithmetic reasoning and commonsense reasoning. For example, we can ask an LLM to solve primary school math problems presented in natural language.

---

<sup>7</sup>To fine-tune an LLM for multi-turn dialogue, one needs to consider conversation history in the context for predicting the response in the current round of conversation. This makes the actual prompt used in response generation relatively longer than that used in single-turn dialogue.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

The answer is 10.

The correct answer should be 7, so the model output is incorrect.

One approach to addressing such issues is to incorporate learning into prompts, called **in-context learning** or (**ICL**). The idea of ICL is to demonstrate the ways to solve problems in prompts, and condition predictions on these demonstrations. Here is an example where a similar problem and the corresponding answer are presented in the prompt (green = demonstrations).

Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?

The answer is 17.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

The answer is 12.

But the LLM still made mistakes this time. A reason for this might be that solving math problems does not only involve problem-answer mappings but also, to a larger extent, the underlying logical inference in multiple steps. A method to improve the inference abilities of LLMs is **chain-of-thought prompting (COT prompting)** [Wei et al., 2022c]. In COT prompting, we decompose complex reasoning problems into multiple problem-solving intermediate steps. These steps are demonstrated in prompts so that LLMs can be prompted to learn to mimic the underlying thought processes in solving the problems. COT prompting has been found to be very useful for achieving good results on challenging tasks, such as the GSM8K mathematical reasoning benchmark.

Consider the above demonstration example *Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now ?*. The corresponding logical reasoning steps can be structured:

1. Tom starts with 12 marbles.
2. He wins 7 more marbles in a game, so we add those to his initial count: 12 marbles + 7 marbles = 19 marbles.
3. The next day, Tom loses 5 marbles, so we subtract those from his current total: 19 marbles - 5 marbles = 14 marbles.

4. His brother gives him 3 more marbles, so we add those to his current total:  $14 \text{ marbles} + 3 \text{ marbles} = 17 \text{ marbles}$ .
5. After all these events, Tom has 17 marbles.

We can add these reasoning steps into the prompt, and get the correct answer.

Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?

Tom initially has 12 marbles. After winning a game, he gains 7 more, increasing his total to 19 marbles. He loses 5 marbles the next day, bringing his total down to 14 marbles. His brother gifts him 3 more marbles, increasing his total to 17 marbles. Therefore, Tom now has 17 marbles. So the answer is 17.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

Jack starts with 7 apples. He eats 2 apples for dinner, so we subtract 2 from 7, leaving him with 5 apples. His mom gives him 5 more apples, adding 5 to his current total. Now he has  $5 + 5 = 10$  apples. The next day, Jack gives 3 apples to his friend John. Subtracting 3 from 10, Jack is left with 7 apples. Therefore, Jack has 7 apples left. So the answer is 7.

This approach is also called **one-shot COT prompting**. Likewise, we can add more demonstrations to have a **few-shot COT prompting** approach.

Furthermore, a special kind of COT, **zero-shot COT**, introduces instructions that provoke LLMs to think, without presenting any intermediate reasoning steps in the prompts [Kojima et al., 2022]. An interesting example is that by appending a simple sentence “Let’s think step by step.” to the end of a prompt, LLMs can generate reasoning steps to reach the final answer.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

Let's think step by step.

1. Initial Quantity: Jack starts with 7 apples.
  2. After Dinner: He eats 2 apples, so  $7 - 2 = 5$  apples remain.
  3. His Mom Gives More: His mom gives him 5 more apples, so  $5 + 5 = 10$  apples are now with him.
  4. Giving to John: The next day, Jack gives 3 apples to his friend John, so  $10 - 3 = 7$  apples are left.
- In the end, Jack has 7 apples left.

Zero-shot, one-shot, and few-shot learning are common concepts in the area of in-context learning for LLMs and are not restricted to COT prompting. Broadly speaking, any prompting that involves only simple instructions without any demonstrations can be considered a form of zero-shot learning. This zero-shot learning ability emerges as LLMs are pre-trained and/or fine-tuned. Also, one-shot and few-shot learning methods are more often considered when LLMs do not acquire the corresponding zero-shot learning ability. These methods are therefore important for in-context learning when addressing new tasks. Examples include those for performing various NLP tasks by demonstrating task-formatted samples. See the following examples for sentiment sentence classification and phrase translation via few-shot learning.

Given the following text snippets, classify their sentiment as Positive, Negative, or Neutral.

Example 1: "I had an amazing day at the park!"

Sentiment: Positive

Example 2: "The service at the restaurant was terrible."

Sentiment: Negative

Example 3: "I think it's going to rain today."

Sentiment: Neutral

Text: "This movie was a fantastic journey through imagination."

Sentiment: Positive

Translate the following Chinese phrases into English.

Example 1: “你好”

Translation: “Hello”

Example 2: “谢谢你”

Translation: “Thank you”

Phrase to translate: “早上好”

Translation: “Good Morning”

Above, we have presented examples to illustrate the fundamental in-context learning capabilities of prompting LLMs. This section, however, does not include more advanced prompting techniques in order to keep the content concise and compact. More discussions on prompting can be found in Chapter 9.

## 8.2 Training at Scale

As a first step in developing LLMs, we need to train these models on large amounts of data. The training task is itself standard: the objective is to maximize the likelihood, which can be achieved via gradient descent. However, as we scale up both the model size and the amount of data, the problem becomes very challenging, for example, large models generally make the training unstable. In this section, we discuss several issues of large-scale training for LLMs, including data preparation, model modification, and distributed training. We also discuss the scaling laws for LLMs, which help us understand their training efficiency and effectiveness.

### 8.2.1 Data Preparation

The importance of data cannot be overstated in NLP. As larger neural networks are developed, the demand for data continues to increase. For example, developing LLMs may require trillions of tokens in pre-training (see Table 8.3), orders of magnitude larger than those used in training conventional NLP models. In general, we may want to gather as much training data as possible. However, larger training datasets do not mean better training results, and the development of LLMs raises new issues in creating or collecting these datasets.

A first issue is the quality of data. High-quality data has long been seen as crucial for training data-driven NLP systems. Directly using raw text from various sources is in general undesirable. For example, a significant portion of the data used to train recent LLMs comes from web scraping, which may contain errors and inappropriate content, such as toxic information and fabricated facts. Also, the internet is flooded with machine-generated content due to the widespread use of AI, presenting further challenges for processing and using web-scraped data. Researchers have found that training LLMs on unfiltered data is harmful [Raffel et al., 2020]. Improving data quality typically involves incorporating filtering and cleaning steps in the data processing workflow. For example, Penedo et al. [2023] show that by adopting a number of data processing techniques, 90% of their web-scraped data can be removed for

LLM	# of Tokens	Data
GPT3-175B [Brown et al., 2020]	0.5T	Webpages, Books, Wikipedia
Falcon-180B [Almazrouei et al., 2023]	3.5T	Webpages, Books, Conversations, Code, Technical Articles
LLaMA2-65B [Touvron et al., 2023a]	1.0T ~ 1.4T	Webpages, Code, Wikipedia, Books, Papers, Q&As
PaLM-450B [Chowdhery et al., 2022]	0.78T	Webpages, Books, Conversations, Code, Wikipedia, News
Gemma-7B [Gemma Team, 2024]	6T	Webpages, Mathematics, Code

Table 8.3: Amounts of training data used in some LLMs in terms of the number of tokens.

LLM training. In addition to large-scale web-scraped data, LLM training data often includes books, papers, user-generated data on social media, and so on. Most of the latest LLMs are trained on such combined datasets, which are found to be important for the strong performance of the resulting models.

A second issue is the diversity of data. We want the training data to cover as many types of data as possible, so that the trained models can adapt to different downstream tasks easily. It has been widely recognized that the quality and diversity of training data both play very important roles in LLMs. An interesting example is that incorporating programming code into training data has been found to be beneficial for LLMs. The benefits are demonstrated not only in enhancing the programming abilities of LLMs, but also in improving reasoning for complex problems, especially those requiring COT prompting. The concept “diversity” can be extended to include language diversity as well. For example, many LLMs are trained on multi-lingual data, and therefore we can handle multiple languages using a single model. While this approach shows strong abilities in multi-lingual and cross-lingual tasks, its performance on specific languages largely depends on the volume and quality of the data for those languages. It has been shown in some cases to provide poor results for low-resource languages.

A third issue is the bias in training data. This is not a problem that is specific to LLMs but exists in many NLP systems. A common example is gender bias, where LLMs show a preference for one gender over another. This can partly be attributed to class imbalance in the training data, for example, the term *nurses* is more often associated with women. In order to debias the data, it is common practice to balance the categories of different language phenomena, such as gender, ethnicity, and dialects. The bias in data is also related to the diversity issue mentioned above. For example, since many LLMs are trained and aligned with English-centric data, they are biased towards the cultural values and perspectives prevalent among English-speaking populations. Increasing language diversity in training data can somewhat mitigate the bias.

Another issue with collecting large-scale data is the privacy concern. If LLMs are trained on data from extensive sources, this potentially leads to risks regarding the exposure of sensitive information, such as intellectual property and personal data. This is particularly

concerning given the capacity of LLMs to represent patterns from the data they are trained on, which might inadvertently involve memorizing and reproducing specific details. A simple approach to privacy protection is to remove or anonymize sensitive information. For example, anonymization techniques can be applied to remove personally identifiable information from training data to prevent LLMs from learning from such data. However, in practice, erasing or redacting all sensitive data is difficult. Therefore, many LLMs, particularly those launched for public service, typically work with systems that can detect the potential exposure of sensitive data, or are fine-tuned to reject certain requests that could lead to information leakage.

## 8.2.2 Model Modifications

Training LLMs is difficult. A commonly encountered problem is that the training process becomes more unstable as LLMs get bigger. For example, one needs to choose a small learning rate to achieve stable training with gradient descent, but this in turn results in much longer training times. Sometimes, even when the training configuration is carefully designed, training may diverge at certain points during optimization. The training of LLMs is generally influenced by many factors, such as parameter initialization, batching, and regularization. Here, we focus on common modifications and improvements to the standard Transformer architecture, which are considered important in developing trainable LLMs.

### 1. Layer Normalization with Residual Connections

Layer normalization is used to stabilize training for deep neural networks. It is a process of subtracting the mean and dividing by the standard deviation. By normalizing layer output in this way, we can effectively reduce the covariate shift problem and improve the training stability. In Transformers, layer normalization is typically used together with residual connections. As described in Section 8.1.1, a sub-layer can be based on either the post-norm architecture, in which layer normalization is performed right after a residual block, or the pre-norm architecture, in which layer normalization is performed inside a residual block. While both of these architectures are widely used in Transformer-based systems [Wang et al., 2019a], the pre-norm architecture has proven to be especially useful in training deep Transformers. Given this, most LLMs are based on the pre-norm architecture, expressed as  $\text{output} = \text{LNorm}(F(\text{input})) + \text{input}$ .

A widely-used form of the layer normalization function is given by

$$\text{LNorm}(\mathbf{h}) = \alpha \cdot \frac{\mathbf{h} - \mu}{\sigma + \epsilon} + \beta \quad (8.23)$$

where  $\mathbf{h}$  is a  $d$ -dimensional real-valued vector,  $\mu$  is the mean of all the entries of  $\mathbf{h}$ , and  $\sigma$  is the corresponding standard deviation.  $\epsilon$  is introduced for the sake of numerical stability.  $\alpha \in \mathbb{R}^d$  and  $\beta \in \mathbb{R}^d$  are the gain and bias terms.

A variant of layer normalization, called root mean square (RMS) layer normalization, only re-scales the input vector but does not re-center it [Zhang and Sennrich, 2019]. The RMS layer

normalization function is given by

$$\text{LNorm}(\mathbf{h}) = \alpha \cdot \frac{\mathbf{h}}{\sigma_{\text{rms}} + \epsilon} + \beta \quad (8.24)$$

where  $\sigma_{\text{rms}}$  is the root mean square of  $\mathbf{h}$ , that is,  $\sigma_{\text{rms}} = (\frac{1}{d} \sum_{k=1}^d h_k^2)^{\frac{1}{2}}$ . This layer normalization function is used in LLMs like the LLaMA series.

## 2. Activation Functions in FFNs

In Transformers, FFN sub-layers are designed to introduce non-linearities into representation learning, and are found to be useful for preventing the representations learned by self-attention from degeneration<sup>8</sup> [Dong et al., 2021]. A standard form of the FFNs used in these sub-layers can be expressed as

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_f + \mathbf{b}_f \quad (8.25)$$

where  $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ ,  $\mathbf{b}_h \in \mathbb{R}^{d_h}$ ,  $\mathbf{W}_f \in \mathbb{R}^{d_h \times d}$ , and  $\mathbf{b}_f \in \mathbb{R}^d$  are the parameters, and  $d_h$  is the hidden size.  $\sigma(\cdot)$  is the activation function of the hidden layer. A common choice for  $\sigma(\cdot)$  is the **rectified linear unit (ReLU)**, given by

$$\sigma_{\text{relu}}(\mathbf{h}) = \max(0, \mathbf{h}) \quad (8.26)$$

In practical implementations, increasing  $d_h$  is helpful and thus it is often set to a larger number in LLMs. But a very large hidden size poses challenges for both training and deployment. In this case, the design of the activation function plays a relatively more important role in wide FFNs. There are several alternatives to the ReLU in LLMs. One of these is the **gaussian error linear unit (GeLU)** which can be seen as a smoothed version of the ReLU. Rather than controlling the output by the sign of the input, the GeLU function weights its input by the percentile  $\Pr(h \leq \mathbf{h})$ . Here  $h$  is a  $d$ -dimensional vector whose entries are drawn from the standard normal distribution  $\text{Gaussian}(0, 1)$ <sup>9</sup>. Specifically, the GeLU function is defined to be

$$\begin{aligned} \sigma_{\text{gelu}}(\mathbf{h}) &= \mathbf{h} \Pr(h \leq \mathbf{h}) \\ &= \mathbf{h} \Phi(\mathbf{h}) \end{aligned} \quad (8.27)$$

where  $\Phi(\mathbf{h})$  is the cumulative distribution function of  $\text{Gaussian}(0, 1)$ , which can be implemented in convenient ways [Hendrycks and Gimpel, 2016]. The GeLU function has been adopted in several LLMs, such as BERT, GPT-3, and BLOOM.

Another family of activation functions which is popular in LLMs is **gated linear unit**

<sup>8</sup>Here degeneration refers to the phenomenon in which the rank of a matrix is reduced after some processing.

<sup>9</sup> $\Pr(h \leq \mathbf{h})$  is an informal notation. It refers to a vector, with each entry representing the percentile for the corresponding entry of  $\mathbf{h}$ .

(GLU)-based functions. The basic form of GLUs is given by

$$\sigma_{\text{glu}}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (8.28)$$

where  $\mathbf{W}_1 \in \mathbb{R}^{d \times d}$ ,  $\mathbf{b}_1 \in \mathbb{R}^d$ ,  $\mathbf{W}_2 \in \mathbb{R}^{d \times d}$ , and  $\mathbf{b}_2 \in \mathbb{R}^d$  are model parameters. Different choices of  $\sigma(\cdot)$  result in different versions of GLU functions. For example, if  $\sigma(\cdot)$  is defined to be the GeLU function, we will have the GeGLU function

$$\sigma_{\text{geglu}}(\mathbf{h}) = \sigma_{\text{gelu}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (8.29)$$

This activation function has been successfully applied in LLMs like Gemma.

As another example, consider  $\sigma(\cdot)$  to be the Swish function  $\sigma_{\text{swish}}(\mathbf{h}) = \mathbf{h} \odot \text{Sigmoid}(c\mathbf{h})$  [Ramachandran et al., 2017]. Then, the SwiGLU function is given by

$$\sigma_{\text{swiglu}}(\mathbf{h}) = \sigma_{\text{swish}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (8.30)$$

Both the PaLM and LLaMA series are based on the SwiGLU function. For more discussions of GLUs, the reader can refer to Shazeer [2020]’s work.

### 3. Removing Bias Terms

Another popular model design is to remove the bias terms in affine transformations used in LLMs. This treatment can be applied to layer normalization, transformations of the inputs to QKV attention, and FFNs. For example, we can modify Eq. (8.25) to obtain an FFN with no bias terms

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h)\mathbf{W}_f \quad (8.31)$$

Chowdhery et al. [2022] report that removing bias terms helps improve the training stability of LLMs. This method has been used in several recent LLMs, such as LLaMA and Gemma.

### 4. Other Issues

Many LLMs also involve modifications to their positional embedding models. For example, one can replace sinusoidal positional encodings with rotary position embeddings so that the learned LLMs can handle long sequences better. These models will be discussed in Section 8.3.

Note that while model modifications are common in training LLMs, the stability of training can be improved in many different ways. For example, increasing the batch size as the training proceeds has been found to be useful for some LLMs. In general, achieving stable and efficient large-scale LLM training requires carefully designed setups, including learning schedules, optimizer choices, training parallelism, mixed precision training, and so on. Some of these issues are highly engineered, and therefore, we typically need a number of training runs to obtain satisfactory LLMs.

### 8.2.3 Distributed Training

Training LLMs requires significant amounts of computational resources. A common approach to improving training efficiency is to use large-scale distributed systems. Fortunately, alongside the rise of neural networks in AI, deep learning-oriented software and hardware have been developed, making it easier to implement LLMs and perform computations. For example, one can now easily fine-tune an LLM using deep learning software frameworks and a machine with multiple GPUs. However, scaling up the training of LLMs is still challenging, and requires significant efforts in developing hardware and software systems for stable and efficient distributed training.

An important consideration of distributed training is parallelism. There are several forms of parallelism: data parallelism, model parallelism, tensor parallelism, and pipeline parallelism. Despite different ways to distribute computations across devices, these parallelism methods are based on a similar idea: the training problem can be divided into smaller tasks that can be executed simultaneously. The issue of parallelism in training LLMs has been extensively studied [Narayanan et al., 2021; Fedus et al., 2022b]. Here we sketch the basic concepts.

- **Data Parallelism.** This method is one of the most widely used parallelism methods for training neural networks. To illustrate, consider the simplest case where the standard delta rule is used in gradient descent

$$\theta_{t+1} = \theta_t - lr \cdot \frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \quad (8.32)$$

where the new parameters  $\theta_{t+1}$  is obtained by updating the latest parameters  $\theta_t$  with a small step  $lr$  in the direction of the negative loss gradient.  $\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t}$  is the gradient of the loss with respect to the parameters  $\theta_t$ , and is computed on a minibatch of training sample  $\mathcal{D}_{\text{mini}}$ . In data parallelism, we divide  $\mathcal{D}_{\text{mini}}$  into  $N$  smaller batches, denoted by  $\{\mathcal{D}^1, \dots, \mathcal{D}^N\}$ . Then, we distribute these batches to  $N$  workers, each with a corresponding batch. Once the data is distributed, these workers can work at the same time. The gradient of the entire minibatch is obtained by aggregating the gradients computed by the workers, like this

$$\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} = \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^1)}{\partial \theta_t}}_{\text{worker 1}} + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^2)}{\partial \theta_t}}_{\text{worker 2}} + \dots + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^N)}{\partial \theta_t}}_{\text{worker } N} \quad (8.33)$$

In ideal cases where the workers coordinate well and the communication overhead is small, data parallelism can achieve nearly an  $N$ -fold speed-up for training.

- **Model Parallelism.** Although data parallelism is simple and effective, it requires each worker to run the entire LLM and perform the complete forward and backward process. As LLMs grow larger, it sometimes becomes unfeasible to load and execute an LLM on a single device. In this case, we can decouple the LLM into smaller components and run these components on different devices. One simple way to do this is to group consecutive layers in the layer stack and assign each group to a worker. The workers

operate in the order of the layers in the stack, that is, in the forward pass we process the input from lower-level to upper-level layers, and in the backward pass we propagate the error gradients from upper-level to lower-level layers. Consider, for example, a Transformer decoder with  $L$  stacked blocks. To distribute the computation load, each block is assigned to a worker. See the following illustration for a single run of the forward and backward passes of this model.

Worker $L$		$B_L$ ( $\uparrow$ )	$B_L$ ( $\downarrow$ )
...		...	...
Worker 2		$B_2$ ( $\uparrow$ )	$B_2$ ( $\downarrow$ )
Worker 1	$B_1$ ( $\uparrow$ )		$B_1$ ( $\downarrow$ )

Here  $B_l$  denotes the computation of block  $l$ , and the symbols  $\uparrow$  and  $\downarrow$  denote the forward and backward passes, respectively. Note that this parallelism method forces the workers to run in sequence, so a worker has to wait for the previous worker to finish their job. This results in the devices being idle for most of the time. In practical systems, model parallelism is generally used together with other parallelism mechanisms to maximize the use of devices.

- **Tensor Parallelism.** Parallelism can also be performed in a single computation step. A common example is splitting a large parameter matrix into chunks, multiplying an input tensor with each of these chunks separately, and then concatenating the results of these multiplications to form the output. For example, consider the multiplication of the representation  $\mathbf{h} \in \mathbb{R}^d$  with the parameter matrix  $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$  in an FFN sub-layer (see Eq. (8.25)). We can slice the matrix  $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$  vertically to a sequence of  $M$  sub-matrices

$$\mathbf{W}_h = \left[ \mathbf{W}_h^1 \quad \mathbf{W}_h^2 \quad \dots \quad \mathbf{W}_h^M \right] \quad (8.34)$$

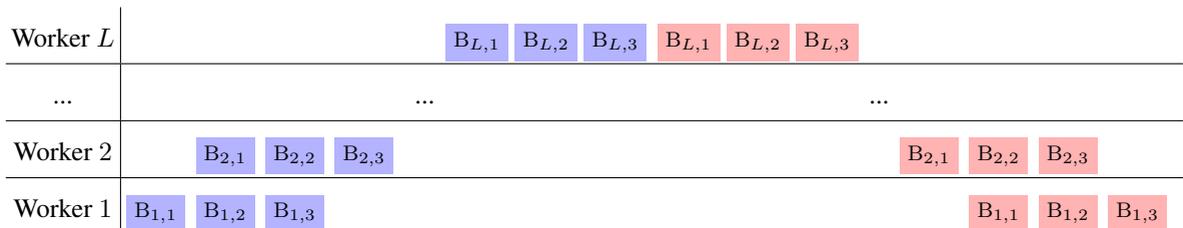
where each sub-matrix  $\mathbf{W}_h^k$  has a shape of  $d \times \frac{d_h}{M}$ . The multiplication of  $\mathbf{h}$  with  $\mathbf{W}_h$  can be expressed as

$$\begin{aligned} \mathbf{h}\mathbf{W}_h &= \mathbf{h} \left[ \mathbf{W}_h^1 \quad \mathbf{W}_h^2 \quad \dots \quad \mathbf{W}_h^M \right] \\ &= \left[ \mathbf{h}\mathbf{W}_h^1 \quad \mathbf{h}\mathbf{W}_h^2 \quad \dots \quad \mathbf{h}\mathbf{W}_h^M \right] \end{aligned} \quad (8.35)$$

We can perform matrix multiplications  $\{\mathbf{h}\mathbf{W}_h^1, \mathbf{h}\mathbf{W}_h^2, \dots, \mathbf{h}\mathbf{W}_h^M\}$  on  $M$  devices separately. As a result, we distribute a large matrix multiplication across multiple devices, each of which may have relatively small memory. From the perspective of the design of modern GPUs, tensor parallelism over GPUs provides a two-level, tile-based approach to parallel computing. First, at a higher level, we decompose a matrix multiplication into sub-matrix multiplications that can directly fit into the memory of GPUs. Then, at

a lower level, we execute these sub-matrix multiplications on GPUs using tile-based parallel algorithms that are specifically optimized for GPUs.

- **Pipeline Parallelism.** Above, in model parallelism, we have described a simple approach to spreading groups of model components across multiple devices. But this method is inefficient because only one device is activated at a time during processing. Pipeline parallelism addresses this issue by introducing overlaps between computations on different devices [Harlap et al., 2018; Huang et al., 2019]. To do this, a batch of samples is divided into a number of micro-batches, and then these micro-batches are processed by each worker as usual. Once a micro-batch is processed by a worker and passed to the next one, the following micro-batch immediately occupies the same worker. In other words, we create a pipeline in which different computation steps can overlap if multiple jobs are given to the pipeline. The following shows an illustration of pipeline parallelism for processing 3 micro-batches.



Here  $B_{l,k}$  represents the processing of the  $k$ -th micro-batch by the  $l$ -th worker. Ideally we would like to maximize the number of micro-batches, and thus minimize the idle time of the workers. However, in practice, using small micro-batches often reduces GPU utilization and increases task-switching costs. This may, in turn, decrease the overall system throughput.

The ultimate goal of parallel processing is to achieve linear growth in efficiency, that is, the number of samples that can be processed per unit of time increases linearly with the number of devices. However, distributed training is complicated, and influenced by many factors in addition to the parallelism method we choose. One problem, which is often associated with distributed systems, is the cost of communication. We can think of a distributed system as a group of networked nodes. Each of these nodes can perform local computation or pass data to other nodes. If there are a large number of such nodes, it will be expensive to distribute and collect data across them. Sometimes, the time savings brought about by parallelism are offset by the communication overhead of a large network. Another problem with large-scale distributed systems is that the synchronization of nodes introduces additional costs. As is often the case, some nodes may take longer to work, causing others to wait for the slowest ones. While we can use asynchronous training to handle heterogeneity in computational resources, this may lead to stale gradients and non-guaranteed convergence. Moreover, as more nodes are added to the network, there is more chance to have crashed nodes during training. In this case, we need to ensure that the whole system is fault tolerant. In many practical settings, to

increase scalability, one needs to take into account additional issues, including architecture design, data transfer and computation overlap, load balancing, memory bandwidth and so on.

Training LLMs is so computationally expensive that, even though distributed training is already in use, researchers and engineers often still employ various model compression and speed-up methods to improve training efficiency [Weng, 2021]. One example is mixed precision training, in which low precision data (such as FP16 and FP8 data) is used for gradient computation on each individual node, and single or double precision data (such as FP32/FP64 data) is used for updating the model [Micikevicius et al., 2018]. A key operation in this approach is gradient accumulation where gradients need to be accumulated and synchronized across nodes. However, due to the non-associativity of floating-point addition, this can lead to slight numerical differences in accumulated gradients on different nodes, which may affect model convergence and final performance. This problem is more obvious if there are a large number of nodes involved in distributed training, especially given that low-precision numerical computations may encounter overflow and underflow issues, as well as inconsistencies across different hardware devices. Therefore, the design of distributed systems needs to consider these numerical computation issues to ensure satisfactory results and convergence.

### 8.2.4 Scaling Laws

The success of LLMs reveals that training larger language models using more resources can lead to improved model performance. Researchers have explained this as **scaling laws** of LLMs. More specifically, scaling laws describe the relationships between the performance of LLMs and the attributes of LLM training, such as the model size, the amount of computation used for training, and the amount of training data. For example, Hestness et al. [2017] show that the performance of deep neural networks is a power-law-like function of the training data size. In the beginning, when the amount of training data is not large, the performance of the model improves slowly. Afterward, when more training data is used, the model enters a phase of rapid performance improvement, and the performance curve resembles a power-law curve. Ultimately, the improvement in performance becomes slow again, and more data does not lead to significant gains. Figure 8.3 shows an example of such curves.

In NLP, a traditional view holds that the performance gains will disappear at a certain point as the training is scaled up. However, recent results show that, if we consider the problem on a larger scale, scaling up training is still a very effective method for obtaining stronger LLMs. For example, both closed-source and open-source LLMs can benefit from more data, even though trillions of tokens have already been used for training.

With the increase in the scale of model training, LLMs exhibit new capabilities, known as the **emergent abilities** of LLMs. For example, Wei et al. [2022b] studied the scaling properties of LLMs across different model sizes and amounts of computational resources. Their work shows that some abilities emerge when we scale the model size to certain level. The appearance of emergent abilities has demonstrated the role of scaled training in enhancing the performance of LLMs, and it has also, to some extent, motivated researchers to continuously attempt to train larger models. As larger and stronger LMs continue to appear, our understanding of the scaling laws continues to mature. This helps researchers predict the performance of LLMs

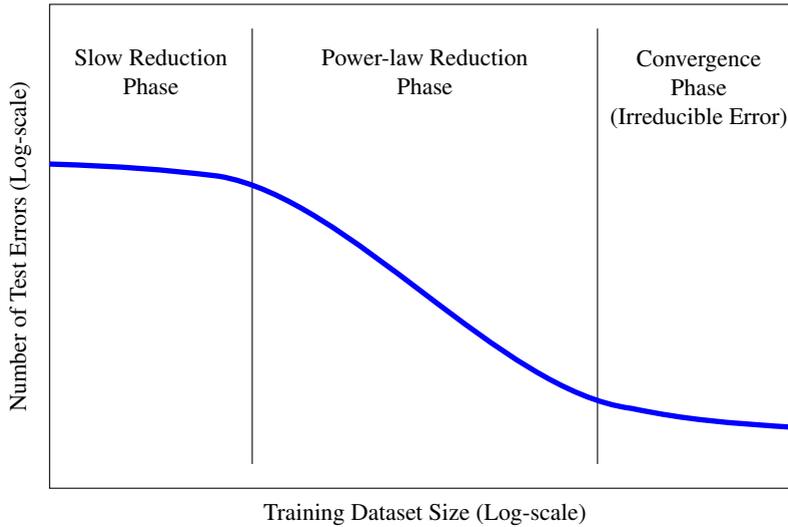


Figure 8.3: A scaling law of test error against a variable of interest (e.g., training dataset size) [Hestness et al., 2017]. The curve of the scaling law can be divided into three phases. At the beginning, the number of test errors decreases slowly when more training data is used, but this only lasts for a short period. In the second phase, the number of test errors decreases drastically, and the curve becomes a power law curve. After that, the error reduction slows down again in the third phase. Note that there are irreducible errors that cannot be eliminated, regardless of the amount of training data.

during training and estimate the minimal computational resources required to achieve a given level of performance.

To understand how model performance scales with various factors considered during training, it is common to express the model performance as a function of these factors. For example, in the simplest case, we can express the loss or error of an LLM as a function of a single variable of interest. However, there are no universal scaling laws that can describe this relationship. Instead, different functions are proposed to fit the learning curves of LLMs.

Let  $x$  be the variable of interest (such as the number of model parameters) and  $\mathcal{L}(x)$  be the loss of the model given  $x$  (such as the cross-entropy loss on test data). The simplest form of  $\mathcal{L}(x)$  is a power law

$$\mathcal{L}(x) = ax^b \quad (8.36)$$

where  $a$  and  $b$  are parameters that are estimated empirically. Despite its simplicity, this function has successfully interpreted the scaling ability of language models and machine translation systems in terms of model size (denoted by  $N$ ) and training dataset size (denoted by  $D$ ) [Gordon et al., 2021; Hestness et al., 2017]. For example, Kaplan et al. [2020] found that the performance of their language model improves as a power law of either  $N$  or  $D$  after an initial transient period, and expressed these relationships using  $\mathcal{L}(N) = \left(\frac{N}{8.8 \times 10^{13}}\right)^{-0.076}$  and  $\mathcal{L}(D) = \left(\frac{D}{5.4 \times 10^{13}}\right)^{-0.095}$  (see Figure 8.4).

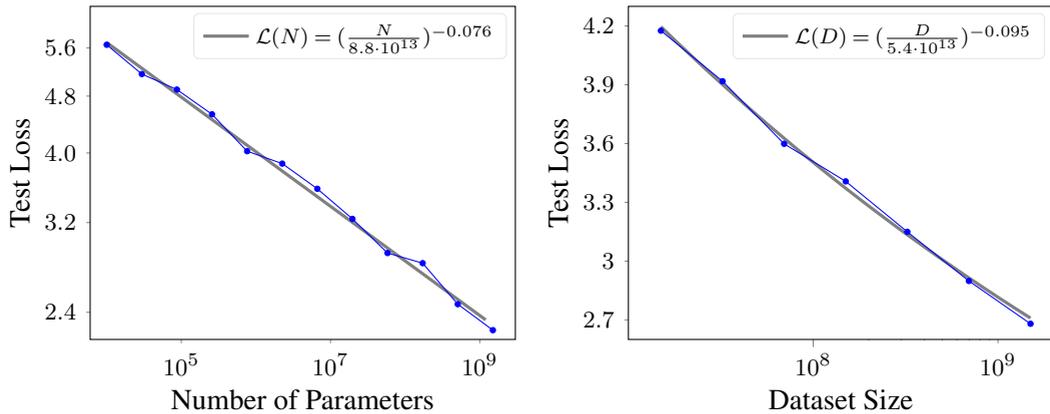


Figure 8.4: Test loss against model size ( $N$ ) and training dataset size ( $D$ ) (data points are plotted for illustrative purposes). We plot test loss as a function of  $N$ , which is defined as  $\mathcal{L}(N) = \left(\frac{N}{8.8 \times 10^{13}}\right)^{-0.076}$ , and a function of  $D$ , which is defined as  $\mathcal{L}(D) = \left(\frac{D}{5.4 \times 10^{13}}\right)^{-0.095}$  [Kaplan et al., 2020].

An improvement to this scaling law is to add an **irreducible error** term to the power law. The form of  $\mathcal{L}(x)$  is then given by

$$\mathcal{L}(x) = ax^b + \epsilon_\infty \quad (8.37)$$

where  $\epsilon_\infty$  is the irreducible error that accounts for the error due to unknown variables, which is present even as  $x \rightarrow \infty$ . Eq. (8.37) is one of the most widely used forms for designing scaling laws of LLMs. For example, Rosenfeld et al. [2020] developed a scaling law that involves both model scaling and dataset scaling, like this

$$\mathcal{L}(N, D) = aN^b + cD^d + \epsilon_\infty \quad (8.38)$$

An example of such formulation is the Chinchilla scaling law. It states that the test loss per token is the sum of the inverse proportion functions of  $N$  and  $D$ , with an additional irreducible error term. Hoffmann et al. [2022] express this scaling law as

$$\mathcal{L}(N, D) = \underbrace{\frac{406.4}{N^{0.34}}}_{\text{model scaling}} + \underbrace{\frac{410.7}{D^{0.28}}}_{\text{dataset scaling}} + \underbrace{1.69}_{\text{irreducible error}} \quad (8.39)$$

All the scaling laws mentioned above are based on monotonic functions. So they cannot cover functions with inflection points, such as double descent curves. In response, researchers have explored more sophisticated functions to fit the learning curves. Examples of such functions can be found in Alabdulmohsin et al. [2022] and Caballero et al. [2023]’s work.

The significance of scaling laws lies in providing directional guidance for LLM research: if we are still in the region of the power law curve, using more resources to train larger models

is a very promising direction. While this result “forces” big research groups and companies to invest more in computational resources to train larger models, which is very expensive, scaling laws continuously push the boundaries of AI further away. On the other hand, understanding scaling laws helps researchers make decisions in training LLMs. For example, given the computational resources at hand, the performance of LLMs may be predicted.

One last note on scaling laws in this section. For LLMs, a lower test loss does not always imply better performance on all downstream tasks. To adapt LLMs, there are several steps such as fine-tuning and prompting that may influence the final result. Therefore, the scaling laws for different downstream tasks might be different in practice.

## 8.3 Long Sequence Modeling

We have already seen that, in large-scale training, larger language models can be developed by using more data and computational resources. However, scaling up can also occur in other directions. For instance, in many applications, LLMs are adapted to process significantly long sequences. An interesting example is that we pre-train an LLM on extensive texts of normal length and then apply it to deal with very long token sequences, far beyond the length encountered in pre-training. Here we use  $\Pr(y|x)$  to denote the text generation probability where  $x$  is the context and  $y$  is the generated text. There are broadly three types of long sequence modeling problems.

- **Text generation based on long context** (i.e.,  $x$  is a long sequence). For example, we generate a short summary for a very long text.
- **Long text generation** (i.e.,  $y$  is a long sequence). For example, we generate a long story based on a few keywords.
- **Long text generation based on long context** (i.e., both  $x$  and  $y$  are long sequences). For example, we translate a long document from Chinese to English.

Recently, NLP researchers have been more interested in applying and evaluating LLMs on tasks where extremely long input texts are involved. Imagine an LLM, which reads a C++ source file containing tens of thousands of lines, and outlines the functionality of the program corresponding to the source file. Such models, capable of handling extensive textual contexts, are sometimes called **long-context LLMs**. In this section we will restrict ourselves to long-context LLMs, but the methods discussed here can be applicable to other problems.

For Transformers, dealing with long sequences is computationally expensive, as the computational cost of self-attention grows quadratically with the sequence length. This makes it infeasible to train and deploy such models for very long inputs. Two strands of research have tried to adapt Transformers to long-context language modeling.

- The first explores efficient training methods and model architectures to learn self-attention models from long-sequence data.
- The other adapts pre-trained LLMs to handle long sequences with modest or no fine-tuning efforts.

Here, we will discuss the former briefly since Chapter 6 extensively covers many methods in this strand. We will focus on the latter, highlighting popular methods in recent LLMs. We will also discuss the strengths and limitations of these long-sequence models.

### 8.3.1 Optimization from HPC Perspectives

We begin our discussion by considering improvements to standard Transformer models from the perspectives of high-performance computing. Most of these improvements, though not specifically designed for LLMs, have been widely applied across various deep learning models [Kim et al., 2023]. A commonly used approach is to adopt a low-precision implementation of Transformers. For example, we can use 8-bit or 16-bit fixed-point data types for arithmetic operations, instead of 32-bit or 64-bit floating-point data types. Using these low-precision data types can increase the efficiency and memory throughput, so that longer sequences can be processed more easily. An alternative approach is to improve Transformers by using hardware-aware techniques. For example, on modern GPUs, the efficiency of Transformers can be improved by using IO-aware implementations of the self-attention function [Dao et al., 2022; Kwon et al., 2023].

Another way to handle long sequences is through sequence parallelism [Li et al., 2023b; Korthikanti et al., 2023]. Specifically, consider the general problem of attending the query  $\mathbf{q}_i$  at the position  $i$  to the keys  $\mathbf{K}$  and values  $\mathbf{V}$ . We can divide  $\mathbf{K}$  by rows and obtain a set of sub-matrices  $\{\mathbf{K}^{[1]}, \dots, \mathbf{K}^{[n_u]}\}$ , each corresponding to a segment of the sequence. Similarly, we can obtain the sub-matrices of  $\mathbf{V}$ , denoted by  $\{\mathbf{V}^{[1]}, \dots, \mathbf{V}^{[n_u]}\}$ . Then, we assign each pair of  $\mathbf{K}^{[u]}$  and  $\mathbf{V}^{[u]}$  to a computing node (e.g., a GPU of a GPU cluster). The assigned nodes can run in parallel, thereby parallelizing the attention operation.

Recall that the output of the self-attention model can be written as

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \sum_{j=0}^{m-1} \alpha_{i,j} \mathbf{v}_j \quad (8.40)$$

where  $\alpha_{i,j}$  is the attention weight between positions  $i$  and  $j$ . In Transformers,  $\alpha_{i,j}$  is obtained by normalizing the rescaled version of the dot product between  $\mathbf{q}_i$  and  $\mathbf{k}_j$ . Let  $\beta_{i,j}$  denote the attention score between  $\mathbf{q}_i$  and  $\mathbf{k}_j$ . We have

$$\beta_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} + \text{Mask}(i,j) \quad (8.41)$$

where  $\text{Mask}(i,j)$  is the masking variable for  $(i,j)$ . Then, we define the attention weight  $\alpha_{i,j}$  to be

$$\begin{aligned} \alpha_{i,j} &= \text{Softmax}(\beta_{i,j}) \\ &= \frac{\exp(\beta_{i,j})}{\sum_{j'} \exp(\beta_{i,j'})} \end{aligned} \quad (8.42)$$

On each computing node, we need to implement these equations. Given the keys and values assigned to this node, computing the numerator of the right-hand side of Eq. (8.42) (i.e.,  $\exp(\beta_{i,j})$ ) is straightforward, as all the required information is stored on the node. However, computing the denominator of the right-hand side of Eq. (8.42) involves a sum of  $\exp(\beta_{i,j'})$  over all  $j'$ 's, which requires transferring data to and from other nodes. To illustrate, suppose that  $\mathbf{v}_j$  and  $\mathbf{k}_j$  are placed on node  $u$ . We can rewrite Eq. (8.42) as

$$\alpha_{i,j} = \frac{\overbrace{\exp(\beta_{i,j})}^{\text{node } u}}{\underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[1]}} \exp(\beta_{i,j'})}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[n_u]}} \exp(\beta_{i,j'})}_{\text{node } n_u}} \quad (8.43)$$

where the notation  $\mathbf{k}_{j'} \in \mathbf{K}^{[u]}$  represents that  $\mathbf{k}_{j'}$  is a row vector of  $\mathbf{K}^{[u]}$ . In a straightforward implementation, we first perform the summations  $\{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})\}$  separately on the corresponding nodes. Then, we collect these summation results from different nodes to combine them into a final result. This corresponds to a collective operation in the context of parallel processing. There are many efficient implementations of such operations, such as the all-reduce algorithms. Hence the sum of all  $\exp(\beta_{i,j})$  values can be computed using optimized routines in collective communication toolkits.

Given the attention weights  $\{\alpha_{i,j}\}$ , we then compute the attention results using Eq. (8.40). The problem can be re-expressed as

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[1]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[n_u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } n_u} \quad (8.44)$$

Like Eq. (8.43), Eq. (8.44) can be implemented as a summation program in parallel processing. First, perform the weighted summations of values on different nodes simultaneously. Then, we collect the results from these nodes via collective operations.

Note that, although this section primarily focuses on long sequence modeling, much of the motivation for sequence parallelism comes from the distributed training methods of deep networks, as discussed in Section 8.2.3. As a result, the implementation of these methods can be based on the same parallel processing library.

### 8.3.2 Efficient Architectures

One difficulty of applying Transformers to long sequences is that self-attention has a quadratic time complexity with respect to the sequence length. Moreover, a **key-value cache** (or **KV cache** for short) is maintained during inference, and its size increases as more tokens are processed. Although the KV cache grows linearly with the sequence length, for extremely

long input sequences, the memory footprint becomes significant and it is even infeasible to deploy LLMs for such tasks. As a result, the model architecture of long-context LLMs generally moves away from the standard Transformer, turning instead to the development of more efficient variants and alternatives.

One approach is to use sparse attention instead of standard self-attention. This family of models is based on the idea that only a small number of tokens are considered important when attending to a given token, and so most of the attention weights between tokens are close to zero. As a consequence, we can prune most of the attention weights and represent the attention model in a compressed form. To illustrate, consider the self-attention model

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \alpha(\mathbf{Q}, \mathbf{K})\mathbf{V} \quad (8.45)$$

where the attention weight matrix  $\alpha(\mathbf{Q}, \mathbf{K}) \in \mathbb{R}^{m \times m}$  is obtained by

$$\begin{aligned} \alpha(\mathbf{Q}, \mathbf{K}) &= \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \mathbf{Mask}\right) \\ &= \begin{bmatrix} \alpha_{0,0} & 0 & 0 & \dots & 0 \\ \alpha_{1,0} & \alpha_{1,1} & 0 & \dots & 0 \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & \dots & \alpha_{m-1,m-1} \end{bmatrix} \end{aligned} \quad (8.46)$$

Each row vector  $[\alpha_{i,0} \dots \alpha_{i,i} \ 0 \dots 0]$  corresponds to a distribution of attending the  $i$ -th token to every token of the sequence. Since language models predict next tokens only based on their left-context, we normally write the output of the attention model at position  $i$  as

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &= \begin{bmatrix} \alpha_{i,0} & \dots & \alpha_{i,i} \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i \end{bmatrix} \\ &= \sum_{j=0}^i \alpha_{i,j} \mathbf{v}_j \end{aligned} \quad (8.47)$$

where  $\mathbf{K}_{\leq i} = \begin{bmatrix} \mathbf{k}_0 \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$  and  $\mathbf{V}_{\leq i} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$  are the keys and values up to position  $i$ .

In the original version of self-attention  $[\alpha_{i,0} \dots \alpha_{i,i}]$  is assumed to be dense, that is, most of the values are non-zero. In sparse attention, some of the entries of  $[\alpha_{i,0} \dots \alpha_{i,i}]$  are considered non-zero, and the remaining entries are simply ignored in computation. Suppose  $G \subseteq \{0, \dots, i\}$  is the set of indices of the non-zero entries. For language models, the output of

the sparse attention model at position  $i$  is given by

$$\text{Att}_{\text{sparse}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j \in G} \alpha'_{i,j} \mathbf{v}_j \quad (8.48)$$

Here  $\{\alpha'_{i,j}\}$  are normalized over  $G$ . Hence their values are different from the original attention weights (in fact we have  $\alpha'_{i,j} > \alpha_{i,j}$ ). The sparsity of the model is determined by how large  $G$  is. Sparse attention models differ in the way we define  $G$ . One simple approach is to define  $G$  based on heuristically designed patterns. For example, a widely-used pattern involves having  $G$  cover a window of tokens located near position  $i$  [Parmar et al., 2018].

While sparse attention reduces the computation through the use of sparse operations, such models still have significant limitations as we must keep the entire KV cache (i.e.,  $\mathbf{K}_{\leq i}$  and  $\mathbf{V}_{\leq i}$ ) during inference. If the sequence is very long, storing this cache will become highly memory-intensive. To address this, we can consider a different form of attention models where the KV cache is not explicitly retained. Linear attention is one such approach [Katharopoulos et al., 2020]. It uses a kernel function  $\phi(\cdot)$  to project each query and key onto points  $\mathbf{q}'_i = \phi(\mathbf{q}_i)$  and  $\mathbf{k}'_i = \phi(\mathbf{k}_i)$ , respectively. By removing the Softmax function under such transformations<sup>10</sup>, the form of the resulting attention model is given by

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &\approx \text{Att}_{\text{linear}}(\mathbf{q}'_i, \mathbf{K}'_{\leq i}, \mathbf{V}_{\leq i}) \\ &= \frac{\mathbf{q}'_i \mu_i}{\mathbf{q}'_i \nu_i} \end{aligned} \quad (8.49)$$

where  $\mu_i$  and  $\nu_i$  are variables that are computed in the recurrent forms

$$\mu_i = \mu_{i-1} + \mathbf{k}'_i{}^T \mathbf{v}_i \quad (8.50)$$

$$\nu_i = \nu_{i-1} + \mathbf{k}'_i{}^T \mathbf{k}'_i \quad (8.51)$$

$\mu_i$  and  $\nu_i$  can be seen as representations of the history up to position  $i$ . A benefit of this model is that we need not keep all past queries and values. Instead only the latest representations  $\mu_i$  and  $\nu_i$  are used. So the computational cost of each step is a constant, and the model can be easily extended to deal with long sequences.

In fact, this sequential approach to long sequence modeling arises naturally when we adopt a viewpoint of recurrent models. Such models read one token (or a small number of tokens) at a time, update the recurrent state using these inputs, and then discard them before the next token arrives. The output at each step is generated based only on the recurrent state, rather than on all the previous states. The memory footprint is determined by the recurrent state which has a fixed size. Recurrent models can be used in real-time learning scenarios where data arrives in a stream and predictions can be made at any time step. In NLP, applying recurrent

<sup>10</sup>In the new space after this transformation, the Softmax normalization can be transformed into the simple scaling normalization.

models to language modeling is one of the earliest successful attempts to learn representations of sequences. Although Transformer has been used as the foundational architecture in LLMs, recurrent models are still powerful models, especially for developing efficient LLMs. More recently, recurrent models have started their resurgence in language modeling and have been reconsidered as a promising alternative to Transformers [Gu and Dao, 2023].

Figure 8.5 shows a comparison of the models discussed in this subsection. Since these models, along with others not mentioned here, have been intensively discussed in Chapter 6 and in related surveys [Tay et al., 2020b], a detailed discussion of them is precluded here.

### 8.3.3 Cache and Memory

LLMs based on the standard Transformer architecture are global models. The inference for these models involves storing the entire left-context in order to make predictions for future tokens. This requires a KV cache where the representations (i.e., keys and values) of all previously-generated tokens are kept, and the cost of caching grows as the inference proceeds. Above, we have discussed methods for optimizing this cache via efficient attention approaches, such as sparse attention and linear attention. Another idea, which may have overlap with the previous discussion, is to explicitly encode the context via an additional memory model.

#### 1. Fixed-size KV Cache

A straightforward approach is to represent the keys and values using a fixed-size memory model. Suppose we have a memory Mem which retains the contextual information. We can write the attention operation at position  $i$  in a general form

$$\text{Att}(\mathbf{q}_i, \text{Mem}) = \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \quad (8.52)$$

In this model, Mem is simply the KV cache, i.e.,  $\text{Mem} = (\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$ . Thus the size of Mem is determined by  $i$ . If we define Mem as a fixed-size variable, then the cost of performing  $\text{Att}(\mathbf{q}_i, \text{Mem})$  will be fixed. There are several alternative ways to design Mem.

- One of the simplest methods is to consider a fixed-size window of previous keys and values. Mem is therefore given by

$$\text{Mem} = (\mathbf{K}_{[i-n_c+1, i]}, \mathbf{V}_{[i-n_c+1, i]}) \quad (8.53)$$

where  $n_c$  denotes the size of the window. The notation  $\mathbf{K}_{[i-n_c+1, i]}$  and  $\mathbf{V}_{[i-n_c+1, i]}$  denote the keys and values over positions from  $i - n_c + 1$  to  $i$ .<sup>11</sup> This model can be seen as a type of local attention model.

- It is also possible to define Mem as a pair of summary vectors, which leads to a more compressed representation of the history. A simple way to summarize the previous keys

---

<sup>11</sup>More formally, we write  $\mathbf{K}_{[i-n_c+1, i]} = \begin{bmatrix} \mathbf{k}_{i-n_c+1} \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$  and  $\mathbf{V}_{[i-n_c+1, i]} = \begin{bmatrix} \mathbf{v}_{i-n_c+1} \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$ . Sometimes we denote  $\mathbf{K}_{[i-n_c+1, i]}$  by  $\{\mathbf{k}_{i-n_c+1}, \dots, \mathbf{k}_i\}$  and  $\mathbf{V}_{[i-n_c+1, i]}$  by  $\{\mathbf{v}_{i-n_c+1}, \dots, \mathbf{v}_i\}$  for notation simplicity.

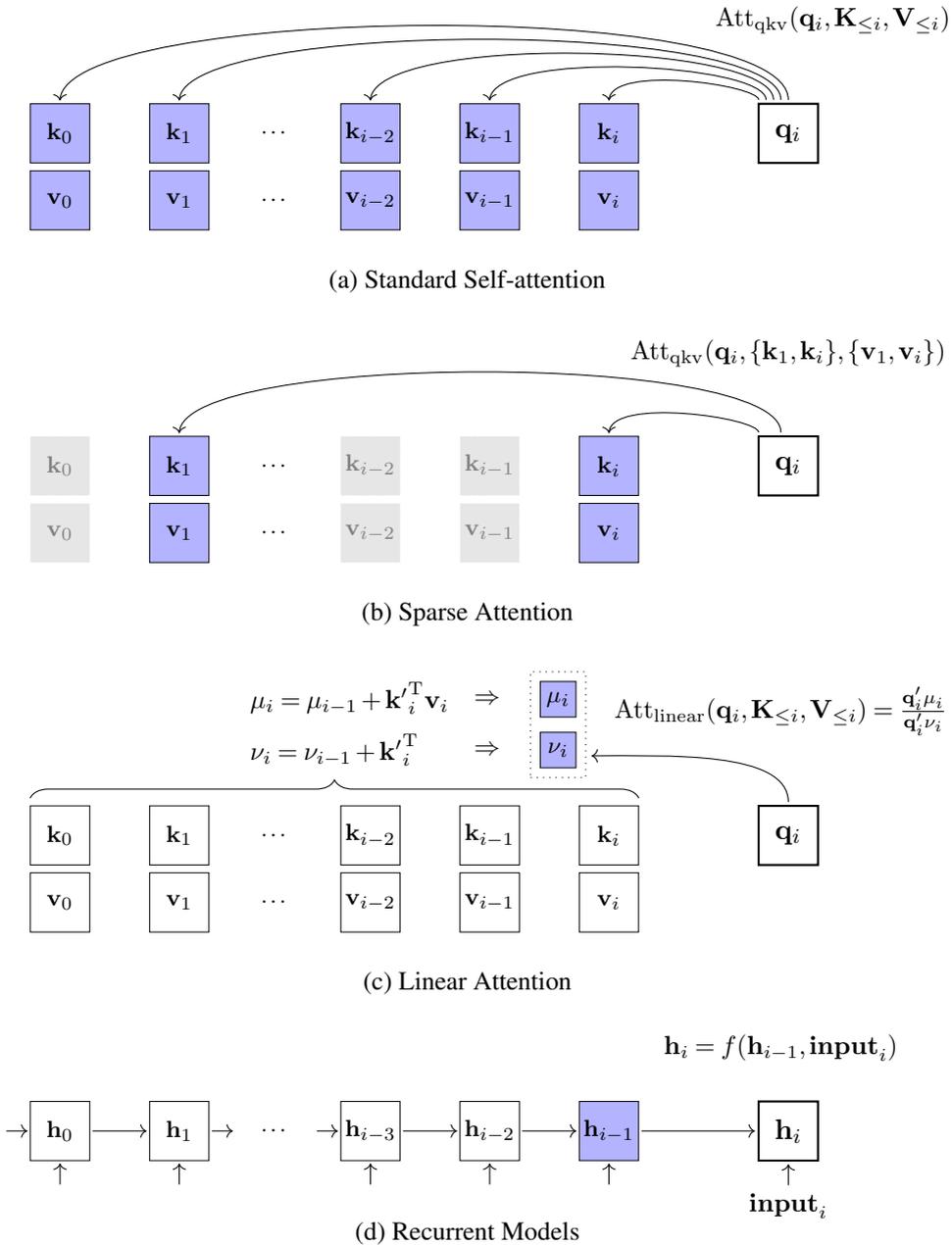


Figure 8.5: Illustrations of self-attention, sparse attention, linear attention and recurrent models. Blue boxes = cached states for producing the output at position  $i$ .  $f(\cdot)$  = a recurrent cell.

and values is to use the moving average of them. For example, Mem can be defined as the unweighted moving average of the previous  $n_c$  keys and values

$$\text{Mem} = \left( \frac{\sum_{j=i-n_c+1}^i \mathbf{k}_j}{n_c}, \frac{\sum_{j=i-n_c+1}^i \mathbf{v}_j}{n_c} \right) \tag{8.54}$$

Alternatively, we can use a weighted version of moving average

$$\text{Mem} = \left( \frac{\sum_{j=i-n_c+1}^i \beta_{j-i+n_c} \mathbf{k}_j}{\sum_{j=1}^{n_c} \beta_j}, \frac{\sum_{j=i-n_c+1}^i \beta_{j-i+n_c} \mathbf{v}_j}{\sum_{j=1}^{n_c} \beta_j} \right) \quad (8.55)$$

Here  $\{\beta_1, \dots, \beta_{n_c}\}$  are the coefficients, which can be either learned as model parameters or determined via heuristics. For example, they can be set to increasing coefficients (i.e.,  $\beta_1 < \beta_2 < \dots < \beta_{n_c-1} < \beta_{n_c}$ ) in order to give larger weight to positions that are closer to  $i$ . We can extend the moving average to include all the positions up to  $i$ . This leads to the cumulative average of the keys and values, given in the form

$$\text{Mem} = \left( \frac{\sum_{j=0}^i \mathbf{k}_j}{i+1}, \frac{\sum_{j=0}^i \mathbf{v}_j}{i+1} \right) \quad (8.56)$$

In general, the cumulative average can be written using a recursive formula

$$\text{Mem}_i = \frac{(\mathbf{k}_i, \mathbf{v}_i) + i \cdot \text{Mem}_{i-1}}{i+1} \quad (8.57)$$

where  $\text{Mem}_i$  and  $\text{Mem}_{i-1}$  denote the cumulative averages of the current and previous positions, respectively. An advantage of this model is that we only need to store a single key-value pair during inference, rather than storing all the key-value pairs. Note that the above memory models are related to recurrent models, and more advanced techniques have been used to develop alternatives to self-attention mechanisms in Transformers [Ma et al., 2023].

- The memory Mem can also be a neural network. At each step, it takes both the previous output of the memory and the current states of the model as input, and produces the new output of the memory. This neural network can be formulated as the function

$$\text{Mem} = \text{Update}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \quad (8.58)$$

Here Mem and  $\text{Mem}_{\text{pre}}$  represent the outputs of the memory at the current step and the previous step, respectively.  $S_{\text{kv}}$  is a set of key-value pairs, representing the recent states of the model. This formulation is general and allows us to develop various memory models by selecting different  $\text{Update}(\cdot)$  and  $S_{\text{kv}}$  configurations. For example, if  $S_{\text{kv}}$  only contains the latest key-value pair  $(\mathbf{k}_i, \mathbf{v}_i)$  and  $\text{Update}(\cdot)$  is defined as a recurrent cell, then Eq. (8.58) can be expressed as an RNN-like model

$$\text{Mem} = f((\mathbf{k}_i, \mathbf{v}_i), \text{Mem}_{\text{pre}}) \quad (8.59)$$

where  $f(\cdot)$  is a recurrent cell. Recurrence can also be applied to segment-level modeling for efficiency consideration. A simple approach is that we can divide the sequence into segments, and treat  $S_{\text{kv}}$  as a segment. Applying recurrent models to  $\text{Update}(\cdot)$  will result in memory models that operate on segments. A special example is that we define

Update( $\cdot$ ) as an FIFO function that adds  $S_{kv}$  into the memory and removes the oldest key-value segment from the memory, given by

$$\text{Mem} = \text{FIFO}(S_{kv}, \text{Mem}_{\text{pre}}) \quad (8.60)$$

Consider a memory which includes two segments, one for current segment, and one for the previous segment. In the attention operation, each position can access the history key-value pairs in two closest consecutive segments. This essentially defines a local memory, but it and its variants have been widely used segment-level recurrent models [Dai et al., 2019; Hutchins et al., 2022; Bulatov et al., 2022].

- The above memory models can be extended to involve multiple memories. An example of this approach is compressive Transformer [Rae et al., 2019b]. It employs two distinct fixed-size memories: one for modeling local context (denoted by Mem), and the other for modeling and compressing long-term history (denoted by CMem). The KV cache in this model is the combination of Mem and CMem. The attention function can be written as

$$\text{Att}_{\text{com}}(\mathbf{q}_i, \text{Mem}, \text{CMem}) = \text{Att}_{\text{qkv}}(\mathbf{q}_i, [\text{Mem}, \text{CMem}]) \quad (8.61)$$

where  $[\text{Mem}, \text{CMem}]$  is a combined memory of Mem and CMem. As with other segment-level models, the compressive Transformer model operates on segments of the sequence. Each segment is a sequence of  $n_s$  consecutive tokens, and we denote  $S_{kv}^k$  as the key-value pairs corresponding to the tokens of the  $k$ -th segment. When a new segment arrives, Mem is updated in an FIFO fashion: we append the  $n_c$  key-value pairs in  $S_{kv}^k$  to Mem, and then pop the  $n_s$  oldest key-value pairs from Mem, which is given by

$$\text{Mem} = \text{FIFO}(S_{kv}^k, \text{Mem}_{\text{pre}}) \quad (8.62)$$

The popped key-value pairs are then used to update the compressive memory CMem. These  $n_s$  key-value pairs are compressed into  $\frac{n_s}{c}$  key-value pairs via a compression network. CMem is an FIFO which appends the compressed  $\frac{n_s}{c}$  key-value pairs to the tail of the queue, and drops the first  $\frac{n_s}{c}$  key-value pairs of the queue. It is given by

$$\text{CMem} = \text{FIFO}(C_{kv}^k, \text{CMem}_{\text{pre}}) \quad (8.63)$$

where  $C_{kv}^k$  represents the set of compressed key-value pairs. Implicit in the compressive Transformer model is that local context should be represented explicitly with minimal information loss, while long-range context can be more compressed.

- We have already seen that both global and local contexts are useful and can be modeled using attention models. This view motivates the extension to attention models for combining both local and long-term memories [Ainslie et al., 2020; Zaheer et al., 2020; Gupta and Berant, 2020]. A simple but widely-used approach is to involve the first few

tokens of the sequence in attention, serving as global tokens. This approach is usually applied along with other sparse attention models. An advantage of incorporating global tokens of the sequence is that it helps smooth the output distribution of the Softmax function used in attention weight computation, and thus stabilizes model performance when the context size is very large [Xiao et al., 2024]. One drawback, however, is that using a fixed-size global memory may result in information loss. When dealing with long sequences, we need to enlarge the KV cache for sufficient representations of the context, but this in turn increases the computational cost.

Figure 8.6 shows illustrations of the above approaches. Note that, while we focus on optimization of the KV cache here, this issue is closely related to those discussed in the previous section. All of the methods we have mentioned so far can broadly be categorized as efficient attention approaches, which are widely used in various Transformer variants.

## 2. Memory-based Models

The modeling of memories discussed above was based on updates to the KV cache, and the resulting models are typically referred to as **internal memories**. We now consider another family of models, called **external memories**, which operate as independent models to access large-scale contexts for LLMs. Many such models are based on **memory-based methods** which have been extensively discussed in machine learning [Bishop, 2006]. A common example is nearest neighbor algorithms: we store context representations in a datastore, and try to find the most similar stored representations to match a given query. The retrieved context representations are then used to improve attention for this query.

Here, we consider the  **$k$ -nearest neighbors** ( $k$ -NN) method which is one of the most popular memory-based methods. Since our focus is language modeling in this section, we define a sample in the datastore as a key-value pair corresponding to some context state. Note that “context” is a broad concept here, not just a sequence prefix in text generation. One might, for example, view the entire dataset as the context for predicting tokens. This allows us to retrieve the closest context situation in a set of sequences, rather than a given sequence prefix. Although we will restrict ourselves to context modeling for a single sequence, in this subsection, we discuss a relatively more general case.

Suppose we have a set of keys  $\{\mathbf{k}_j\}$  with corresponding values  $\{\mathbf{v}_j\}$ , and suppose we store these key-value pairs in a vector database<sup>12</sup>. For each query  $\mathbf{q}_i$ , we find its  $k$  nearest neighbours by growing the radius of the sphere centered as  $\mathbf{q}_i$  until it contains  $k$  data points in  $\{\mathbf{k}_j\}$ . This results in a set of  $k$  keys along with their corresponding values, denoted by  $\text{Mem}_{k\text{nn}}$ . As before, we denote  $\text{Mem}$  as the local memory for the query, such as the KV cache of neighboring tokens. Our goal is to attend query  $\mathbf{q}_i$  to both the local memory  $\text{Mem}$  and the long-term memory  $\text{Mem}_{k\text{nn}}$ . There are, of course, several ways to incorporate  $\text{Mem}$  and  $\text{Mem}_{k\text{nn}}$  into the attention model. For example, we might simply combine them to form a single KV cache  $[\text{Mem}, \text{Mem}_{k\text{nn}}]$ , and attend  $\mathbf{q}_i$  to  $[\text{Mem}, \text{Mem}_{k\text{nn}}]$  via standard QKV attention. Or we might

<sup>12</sup>A vector database, or vector store, is a database that provides highly optimized retrieval interfaces for finding stored vectors that closely match a query vector.

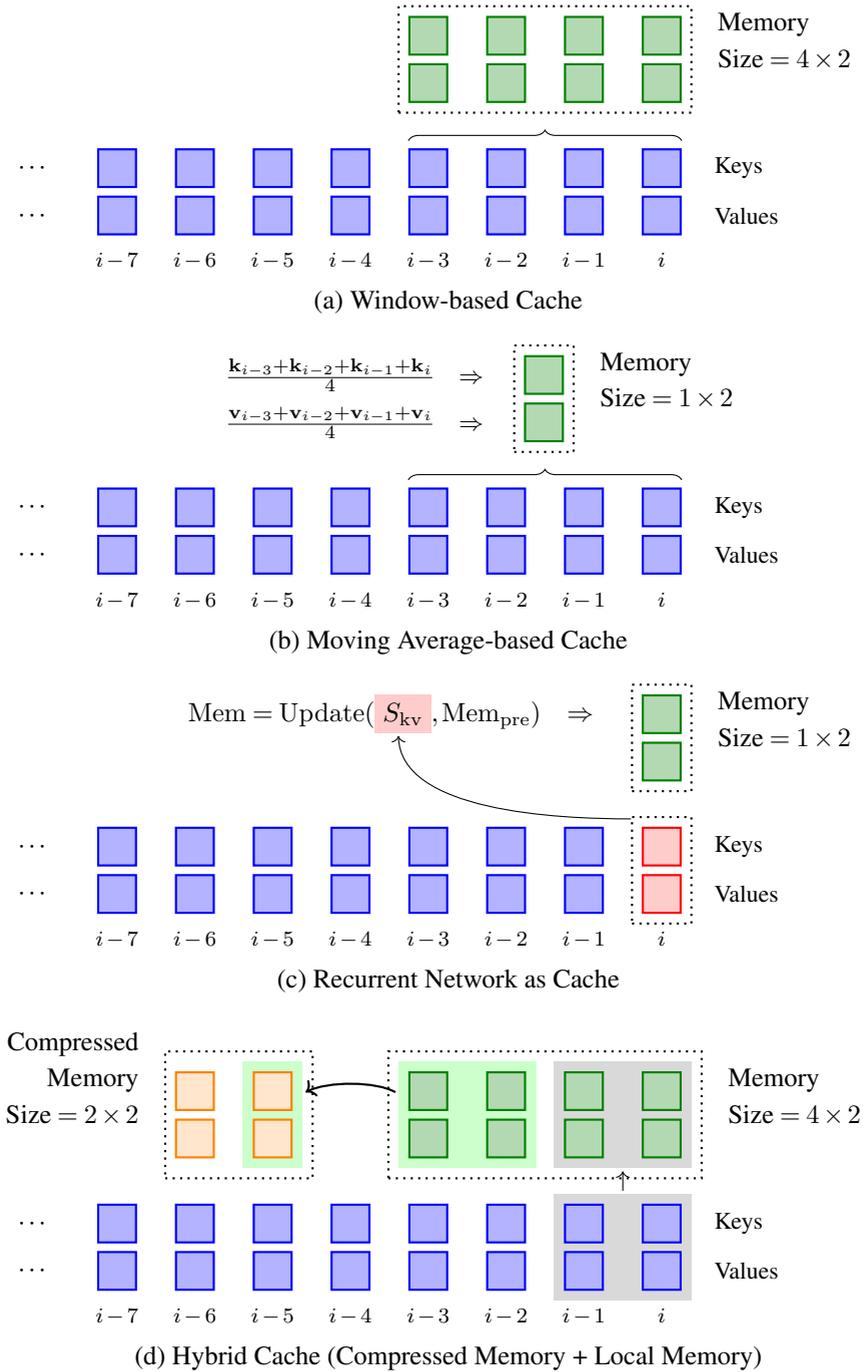


Figure 8.6: Illustrations of fixed-size KV caches in LLMs. Blue boxes represent the keys and values generated during LLM inference, green boxes represent the keys and values stored or encoded in the primary memory, and orange boxes represent the keys and values stored or encoded in the compressed memory.

use  $\text{Mem}$  and  $\text{Mem}_{k\text{nn}}$  in separate attention steps. An example of such approaches is the model developed by [Wu et al. \[2021\]](#). It linearly combines the two types of attention, given by

$$\text{Att}(\mathbf{q}_i, \text{Mem}, \text{Mem}_{k\text{nn}}) = \mathbf{g} \odot \text{Att}_{\text{local}} + (1 - \mathbf{g}) \odot \text{Att}_{k\text{nn}} \quad (8.64)$$

$$\text{Att}_{\text{local}} = \text{Att}(\mathbf{q}_i, \text{Mem}) \quad (8.65)$$

$$\text{Att}_{k\text{nn}} = \text{Att}(\mathbf{q}_i, \text{Mem}_{k\text{nn}}) \quad (8.66)$$

Here  $\mathbf{g} \in \mathbb{R}^d$  is the coefficient vector, which can be the output of a learned gate.

Given the  $k$ -NN-based memory model described above, the remaining task is to determine which key-value pairs are retained in the datastore. For standard language modeling tasks, we consider the previously seen tokens in a sequence as the context, so we can add the keys and values of all these tokens into the datastore. In this case, the resulting  $k$ -NN-based attention model is essentially equivalent to a sparse attention model [[Gupta et al., 2021](#)].

Alternatively, we can extend the context from one sequence to a collection of sequences. For example, we might collect all key-value pairs across the sequences in a training dataset and add them to the datastore to model a larger context. Thus, LLMs can predict tokens based on a generalized context. A problem with this approach is that the computational cost would be large if many sequences are involved. Since these sequences are part of our training data, we can build and optimize an index for the vectors in the datastore before running the LLMs. As a result, the retrieval of similar vectors can be very efficient, as in most vector databases.

In fact, all the above-mentioned methods can be viewed as instances of a retrieval-based approach. Instead of using retrieval results to improve attention, we can apply this approach in other ways as well. One application of  $k$ -NN-based search is  **$k$ -NN language modeling** (or  **$k$ -NN LM**) [[Khandelwal et al., 2020](#)]. The idea is that, although it is attempting to extend the context used in self-attention by incorporating nearest neighbors in representation learning, in practice, similar hidden states in Transformers are often highly predictive of similar tokens in subsequent positions. In  $k$ -NN LM, each item in the datastore is a key-value tuple  $(\mathbf{z}, w)$ , where  $\mathbf{z}$  represents a hidden state of the LLM at a position, and  $w$  represents the corresponding prediction. A typical way to create the datastore is to collect the output vector of the Transformer layer stack and the corresponding next token for each position of each sequence in a training dataset. During inference, we have a representation  $\mathbf{h}_i$  given a prefix. Given this representation, we first search the datastore for  $k$  closest matching data items  $\{(\mathbf{z}_1, w_1), \dots, (\mathbf{z}_k, w_k)\}$ . Here  $\{w_1, \dots, w_k\}$  are thought of as reference tokens for prediction, and thus can be used to guide the token prediction based on  $\mathbf{h}_i$ . One common way to make use of reference tokens is to define a distribution over the vocabulary  $V$ ,

$$\text{Pr}_{k\text{nn}}(\cdot | \mathbf{h}_i) = \text{Softmax}\left(\begin{bmatrix} -d_0 & \dots & -d_{|V|} \end{bmatrix}\right) \quad (8.67)$$

where  $d_v$  equals the distance between  $\mathbf{h}_i$  and  $\mathbf{z}_j$  if  $w_j$  equals the  $v$ -th entry of  $V$ , and equals 0 otherwise. We use a linear function with a coefficient  $\lambda$  that interpolates between the

retrieval-based distribution  $\Pr_{k_{\text{nn}}}(\cdot|\mathbf{h}_i)$  and the LLM output distribution  $\Pr_{\text{lm}}(\cdot|\mathbf{h}_i)$

$$\Pr(\cdot|\mathbf{h}_i) = \lambda \cdot \Pr_{k_{\text{nn}}}(\cdot|\mathbf{h}_i) + (1 - \lambda) \cdot \Pr_{\text{lm}}(\cdot|\mathbf{h}_i) \quad (8.68)$$

Then, as usual, we can choose the next token  $y$  by maximizing the probability  $\Pr(y|\mathbf{h}_i)$ .

As with information retrieval (IR) systems, the datastore can also manage texts and provide access to relevant texts for a query. For example, we can store a collection of text documents in a search engine with full-text indexing, and then search it for documents that match a given text-based query. Applying IR techniques to LLMs leads to a general framework called **retrieval-augmented generation (RAG)**. The RAG framework works as follows. We use the context  $\mathbf{x}$  as the query and find the  $k$  most relevant document pieces  $\{\mathbf{c}_1, \dots, \mathbf{c}_k\}$  from the datastore via efficient IR techniques<sup>13</sup>. These search results are combined with the original context via a prompting template  $g(\cdot)$ <sup>14</sup>, resulting in an augmented input for the LLM

$$\mathbf{x}' = g(\mathbf{c}_1, \dots, \mathbf{c}_k, \mathbf{x}) \quad (8.69)$$

Then, we use  $\mathbf{x}'$  as the context and predict the following text using the model  $\Pr(y|\mathbf{x}')$ . One advantage of RAG is that we need not modify the architecture of LLMs, but instead augment the input to LLMs via an additional IR system. Figure 8.7 shows a comparison of the use of different external memories in LLMs.

### 3. Memory Capacity

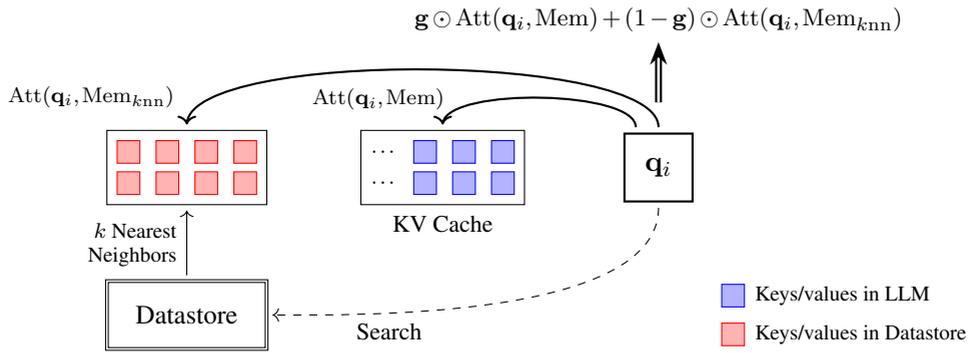
A memory model in LLMs, in the form of a simple key-value cache or a datastore, can broadly be seen as an encoder of contextual information. Ideally, before we say that a memory model is representative of the entire context in token prediction, we need to make sure that the model can accurately represent any part of the context. The standard KV cache is one such model that completely stores all past history. In this case, the model is said to have adequate capacity for memorizing the context. In many practical applications, however, complete memorization is not required. Instead, the goal is to enable LLMs to access important contextual information. As a result, efficient and compressed memory models are developed, as described in this section. Note that, the longer the sequence, the more difficult it becomes for a low-capacity memory model to capture important contextual information. It is therefore common practice to simply increase the model capacity when processing long contexts.

While high-capacity models are generally favorable, they are difficult to train and deploy. A challenging scenario is that the tokens arrive in a stream and the context continuously grows.

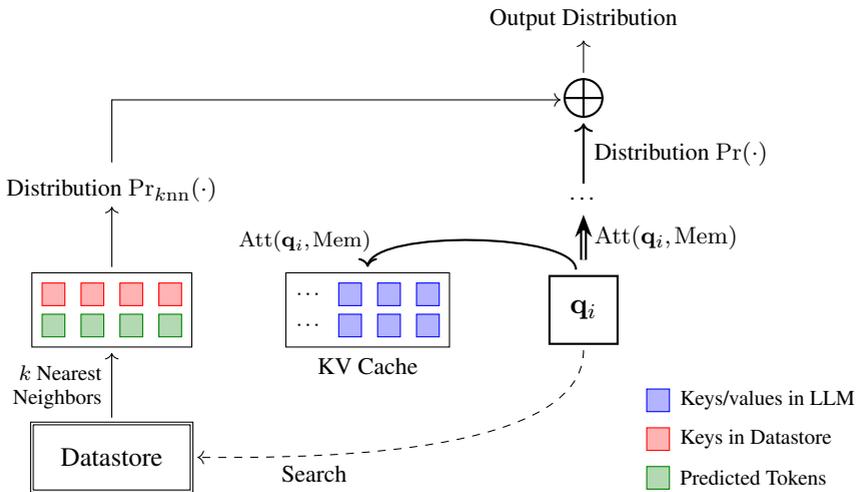
<sup>13</sup>In piratical applications, queries are typically generated using a query generation system, which may expand it with variations of tokens and query intent.

<sup>14</sup>For example, the template could be:

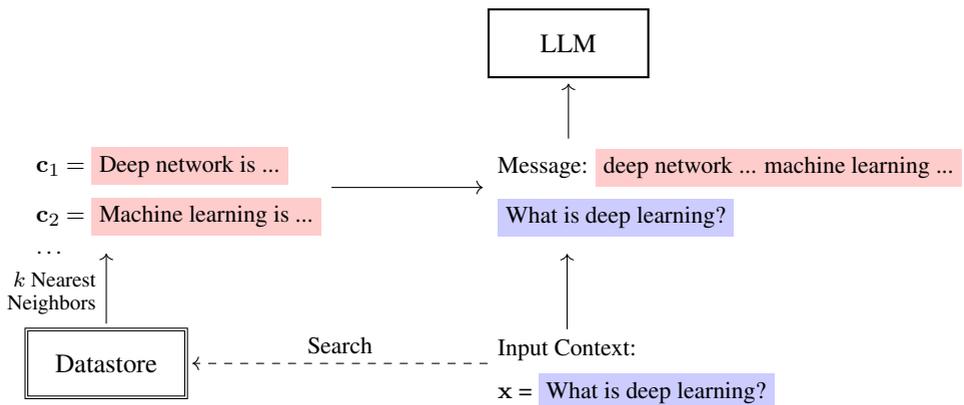
```
message = {*\mathbf{c}_1*} ... {*\mathbf{c}_k*}
input: {*\mathbf{x}*}
output: _____
```



(a)  $k$ -NN Search Augmented Attention



(b)  $k$ -NN Language Modeling



(c) Retrieval-augmented Generation

Figure 8.7: Illustrations of external memories (or datastores) for language modeling.

Developing LLMs for such tasks is difficult as we need to train Transformers on extremely long sequences. A possible way to address this difficulty is to use non-parametric methods, such as retrieval-based methods. For example, as discussed above, we can use a vector database to store previously generated key-value pairs, and thus represent the context by this external memory model. Although this approach side-steps the challenge of representing long context in Transformers, building and updating external memory models are computationally expensive. These models are more often used in problems where the context is given in advance and fixed during inference, and hence unsuitable for streaming context modeling.

In cases where the size of the context continuously grows, applying fixed-size memory models is a commonly used approach. For example, in recurrent models, a sequence of arbitrary length can be summarized into a set of hidden states by which we have a fixed computational cost per step. While recurrent models were initially found to be not very good at handling long-distance dependencies in sequence modeling in early applications of deep learning to NLP, recent advancements have shown that their variants are now effective in modeling extremely long sequences. [Bulatov et al., 2022; Hutchins et al., 2022; Munkhdalai et al., 2024; Ma et al., 2024].

There is no general definition of memory capacity in LLMs. A simple approach might consider how much storage is used to retain contextual information. For example, memory capacity could be defined by the size of the KV cache in Transformers or the vector database used in retrieval-based methods. A related concept is model complexity. In machine learning, there are several ways to define the model complexity of a model. One of the simplest methods is by counting the number of parameters. However, it should be emphasized that the memory models discussed here primarily serve to store information, rather than add trainable parameters. Therefore, a model with a large memory capacity is not necessarily more complex. Nevertheless, in practice determining the capacity of a memory model is not straightforward. In general, we need to control the trade-off between maximizing the performance and controlling the memory footprint.

### 8.3.4 Sharing across Heads and Layers

In Transformers, the KV cache is a data structure that can be dynamically adjusted along multiple dimensions, such as heads, layers, and sequence length. For example, consider an LLM with  $L$  layers. Each layer has  $\tau$  attention heads, and each head produces a  $d_h$ -dimensional output. During inference, we store the keys and values for up to  $m$  tokens. The space complexity of this caching mechanism is  $O(L \cdot \tau \cdot d_h \cdot m)$ . As we have seen previously, this complexity can be reduced by caching the keys and values for fewer tokens. For example, in sliding window attention, a fixed-size window is used to cache the keys and values in local context. And this model has a space complexity of  $O(L \cdot \tau \cdot d_h \cdot m_w)$ , with  $m_w$  being the size of the window.

In addition to reducing  $m$ , we can also decrease the size of the KV cache along other dimensions. A widely-used approach is to enable sharing across heads in multi-head self-attention. Recall from Section 8.1.1 that multi-head self-attention uses multiple sets of queries, keys, and values (each set is called a head), each performing the QKV attention mechanism as

usual. This can be expressed as

$$\text{Output} = \text{Merge}(\text{head}_1, \dots, \text{head}_\tau) \mathbf{W}^{\text{head}} \quad (8.70)$$

where  $\text{head}_j \in \mathbb{R}^{d_h}$  is computed using the standard QKV attention function

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[j]}, \mathbf{V}_{\leq i}^{[j]}) \quad (8.71)$$

Here,  $\mathbf{q}_i^{[j]}$ ,  $\mathbf{K}_{\leq i}^{[j]}$ , and  $\mathbf{V}_{\leq i}^{[j]}$  are the query, keys, and values that are projected onto the  $j$ -th feature sub-space. So this model can be interpreted as performing attention on a group of feature sub-spaces in parallel (see Figure 8.8 (b)). The KV cache needs to retain the keys and values for all these heads, that is,  $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), \dots, (\mathbf{K}_{\leq i}^{[\tau]}, \mathbf{V}_{\leq i}^{[\tau]})\}$ .

One refinement to the multi-head attention model, called **multi-query attention (MQA)**, is to share keys and values across heads, while allowing queries to be unique for each head [Shazeer, 2019]. In MQA, there is a single set of keys and values  $(\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$ . In addition, there are  $\tau$  queries  $\{\mathbf{q}_i^{[1]}, \dots, \mathbf{q}_i^{[\tau]}\}$ , each corresponding to a different head. For each head, we have

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \quad (8.72)$$

Figure 8.8 (c) illustrates this model. By sharing keys and values, the size of the KV cache would be  $O(L \cdot d_h \cdot m)$ .

**Grouped query attention (GQA)** is a natural extension to multi-head attention and MQA [Ainslie et al., 2023]. In GQA, heads are divided into  $n_g$  groups, each corresponding to a shared set of keys and values. Hence we have  $n_g$  sets of keys and values  $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), \dots, (\mathbf{K}_{\leq i}^{[n_g]}, \mathbf{V}_{\leq i}^{[n_g]})\}$ . See Figure 8.8 (d) for an illustration. Let  $g(j)$  be the group id for the  $j$ -th head. The GQA model can be expressed as

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[g(j)]}, \mathbf{V}_{\leq i}^{[g(j)]}) \quad (8.73)$$

The size of the KV cache of GQA is  $O(L \cdot n_g \cdot d_h \cdot m)$ . One benefit of GQA is that we can trade-off between computational efficiency and model expressiveness by adjusting  $n_g$ . When  $n_g = \tau$ , the model becomes the standard multi-head attention model. By contrast, when  $n_g = 1$ , it becomes the MQA model.

Sharing can also be performed across layers. Such a method falls into the family of shared weight and shared activation methods, which have been extensively used in Transformers [Dehghani et al., 2018; Lan et al., 2020]. For example, one can share KV activations or attention weights across layers to reduce both computation and memory footprints [Xiao et al., 2019; Brandon et al., 2024]. Figure 8.8 (e) shows an illustration of this method, where a query in a layer directly accesses the KV cache of a lower-level layer.

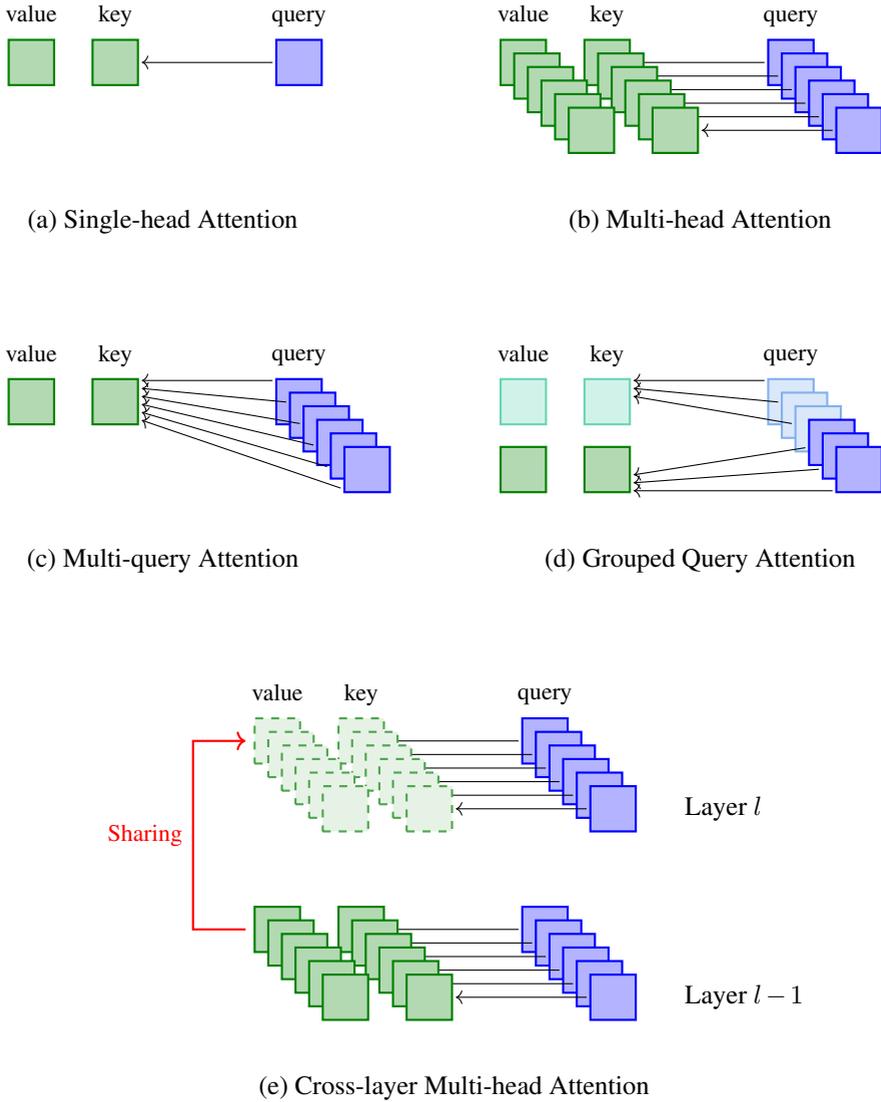


Figure 8.8: Illustration of QKV attention based on different multi-head and sharing mechanisms. (a) = single-head attention, and (b-e) = attention with multiple heads.

### 8.3.5 Position Extrapolation and Interpolation

Since Transformer layers are order-insensitive to input, we need some way to encode positional information in the input tokens. To do this, it is common to add positional embeddings to token embeddings, and then feed these combined embeddings into the Transformer layer stack as input. In this case, the embedding at position  $i$  can be expressed as

$$\mathbf{e}_i = \mathbf{x}_i + \text{PE}(i) \tag{8.74}$$

where  $\mathbf{x}_i \in \mathbb{R}^d$  denotes the token embedding, and  $\text{PE}(i) \in \mathbb{R}^d$  denotes the positional embedding.

In general, the token embedding  $\mathbf{x}_i$  is a position-independent vector, and so the positional embedding  $\text{PE}(i)$  is used to encode the positional context. A straightforward approach is to treat  $\text{PE}(i)$  as a learnable variable and train it alongside other model parameters. In this way, we can learn a unique representation for each position, and thus distinguish the tokens appearing at different positions of a sequence.

Representations of positions using learned vectors can work well in tasks where the sequences at training and test times are of similar lengths. In practice, however, we often impose length restrictions on sequences during training to prevent excessive computational costs, but wish to apply the trained models to much longer sequences during inference. In this case, using learned positional embeddings has obvious drawbacks, as there are no trained embeddings for positions that are not observed in the training phase.

An alternative approach to modeling positional information is to develop positional embeddings that can generalize: once trained, the embedding model can be used to handle longer sequences. Suppose that we train a positional embedding model on sequences with a maximum length of  $m_l$ , and we wish to apply the trained model to a sequence of length  $m$  ( $m \gg m_l$ ). If the embedding model is limited in the range of positions that we can observe from training data, then this model will simply fail to deal with new data outside that range. See Figure 8.9 (a) for an illustration where the learned embedding model cannot model data points outside the training domain if it lacks the ability to extrapolate.

There are several approaches to making positional embedding models generalize. They can be grouped into two classes.

- **Extrapolation.** The model learned on observed data points (i.e., positions) can be directly employed to assign meaningful values to data points beyond the original range. For example, suppose we have a series of numbers 1, 2, ..., 10, and we want to understand the meaning of a new number, 15. Knowing that these numbers are natural numbers used for ordering, we can easily infer that 15 is a number that follows 10, even though 15 has not been observed before. Figure 8.9 (b) shows an example of this approach, where a function is learned to fit the data points within a specific range and then applied to estimate the values of data points outside that range.
- **Interpolation.** This approach maps a larger range of data points into the original observation range. For example, suppose we have a model designed for numbers in the range [1, 10]. When given a new range of [1, 20], we can scale this down by dividing every number by 2, thereby fitting all numbers into [1, 10]. This scaling allows us to use the model trained on the range [1, 10] to describe data points in the expanded range of [1, 20]. See Figure 8.9 (c) for an illustration of this approach.

In fact, positional embeddings in many systems have achieved some level of generalization. For example, sinusoidal encoding, the most common positional embedding method, employs sine and cosine functions that can naturally extend to sequences of any length. Although this approach might seem direct and simple, it does not perform well when we significantly extend the sequences for processing. In this subsection, we will discuss several alternative methods based on either extrapolation or interpolation.

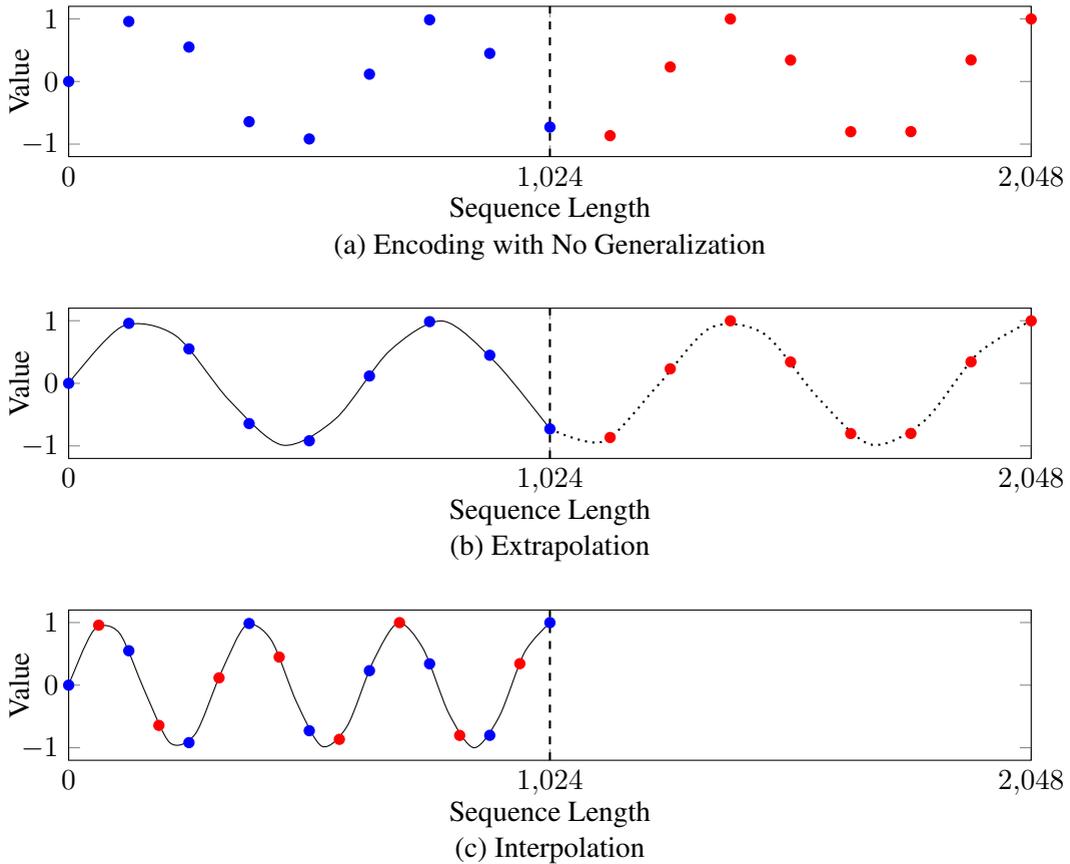


Figure 8.9: Illustrations of different positional embedding methods for a range of positions. Blue points represent the positions that have been observed during training, and red points represent the positions that are newly observed at test time. In sub-figure (a), the encoding model only memorizes the points seen during training, and cannot generalize. In sub-figures (b) and (c), the model can generalize through extrapolation and interpolation.

## 1. Attention with Learnable Biases

One problem with Eq. (8.74) is that the embedding model treats each token independently and therefore ignores the distance between different tokens. A common improvement to this model, called relative positional embedding, is to consider the pairwise relationship between tokens [Shaw et al., 2018]. The general idea behind this is to obtain the offset between any pair of positions and incorporate it into the self-attention model. One of the simplest forms of self-attention with relative positional embedding is given by

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j=0}^i \alpha(i, j) \mathbf{v}_j \quad (8.75)$$

$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + \text{PE}(i, j)}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (8.76)$$

The only difference between this model and the original self-attention model is that a bias term  $\text{PE}(i, j)$  is added to the query-key product in this new model. Intuitively,  $\text{PE}(i, j)$  can be interpreted as a distance penalty for the pair of positions  $i$  and  $j$ . As  $i$  moves away from  $j$ , the value of  $\text{PE}(i, j)$  decreases.

$\text{PE}(i, j)$  can be defined in several different ways. Here, we consider the T5 version of relative positional embedding, called the T5 bias [Raffel et al., 2020]. For each pair of query  $\mathbf{q}_i$  and key  $\mathbf{k}_j$ , the offset between them is defined to be<sup>15</sup>

$$d(i, j) = i - j \quad (8.77)$$

A simple design for the bias  $\text{PE}(i, j)$  is to share the same learnable variable for all query-key pairs with the same offset, i.e.,  $\text{PE}(i, j) = u_{i-j}$ , where  $u_{i-j}$  is the variable corresponding to the offset  $i - j$ . However, simply assigning a unique value to each offset will restrict this model to observed offsets. When  $i - j$  is larger than the maximum trained offset, the model cannot generalize.

The T5 bias instead adopts a generalization of this model. Rather than assigning each query-key offset a unique bias term, it groups difference offsets into “buckets”, each corresponding to one learnable parameter. More specifically, the bias terms for  $n_b + 1$  buckets are given as follows.

- For buckets 0 to  $\frac{n_b+1}{2} - 1$ , each bucket corresponds to one offset, that is, bucket 0  $\leftrightarrow$  offset 0, bucket 1  $\leftrightarrow$  offset 1, bucket 2  $\leftrightarrow$  offset 2, and so on. We express this as  $b(i - j) = i - j$ .
- For buckets  $\frac{n_b+1}{2}$  to  $n_b$ , the size of each bucket increases logarithmically. For example, the bucket number for a given offset  $i - j \geq \frac{n_b+1}{2}$  can be defined as

$$b(i - j) = \frac{n_b + 1}{2} + \left\lfloor \frac{\log(i - j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b + 1}{2} \right\rfloor \quad (8.78)$$

where the parameter  $\text{dist}_{\max}$  is typically set to a relatively large number to indicate the maximum offset we may encounter.

- When  $i - j > \text{dist}_{\max}$ , we place  $i - j$  in the last bucket. In other words, bucket  $n_b$  contains all the offsets that are not assigned to the previous buckets.

Together, these can be expressed as the function

$$b(i - j) = \begin{cases} i - j & 0 \leq i - j < \frac{n_b+1}{2} \\ \min(n_b, \frac{n_b+1}{2} + \left\lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \right\rfloor) & i - j \geq \frac{n_b+1}{2} \end{cases} \quad (8.79)$$

Figure 8.10 shows an illustration of these buckets. We see that in the first half of the

<sup>15</sup>For language modeling, a query is only allowed to attend to its left-context, and so we have  $i - j \geq 0$ . In the more general case of self-attention, where a token can attend to all tokens in the sequence, we may have negative offsets when  $i < j$ .

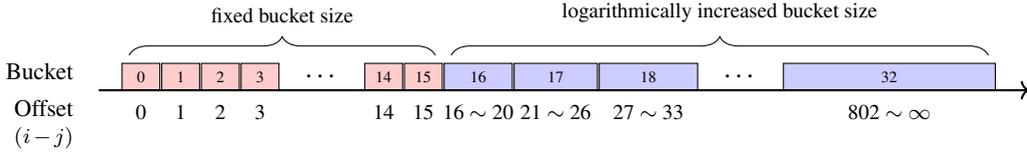


Figure 8.10: Illustration of distributing query-key offsets into buckets in the T5 model ( $n_b = 32$  and  $\text{dist}_{\max} = 1024$ ). Boxes represent buckets. In the first half of the buckets, we use a fixed bucket size. In the second half of the buckets, we increase the bucket size logarithmically. The last bucket contains all the query-key offsets that are not covered by previous buckets.

buckets, each bucket is associated with only one value of  $i - j$ , while in the second half, the bucket size increases as  $i - j$  grows. The last bucket is designed to handle sequences of arbitrarily long lengths.

All  $\text{PE}(i, j)$ s in a bucket share the same bias term  $u_{b(i-j)}$ . Substituting  $\text{PE}(i, j) = u_{b(i-j)}$  into Eq. (8.76), the attention weight for  $\mathbf{q}_i$  and  $\mathbf{k}_j$  becomes<sup>16</sup>

$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + u_{b(i-j)}}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (8.81)$$

The parameters  $\{u_0, \dots, u_{n_b}\}$  are learned as common parameters during training. It should be emphasized that this model can generalize to long sequences. This is because  $\text{PE}(i, j)$ s with similar query-key offsets share the same parameter, and this sharing strategy is particularly important for achieving good generalization, given that large query-key offsets are rare in training. In practice, we often set  $n_b$  to a moderate number, and thus it can help control the overfitting of positional embedding models.

## 2. Attention with Non-learned Biases

Relative positional embedding models are based on a set of learned biases for the query-key product in self-attention. An alternative approach is to give these biases fixed values via heuristics, rather than training them on a particular dataset. One benefit of this heuristics-based approach is that it does not rely on a training process and thus can be directly applied to any sequences once the biases are set.

One example of such an approach is Press et al. [2022]’s approach, called **attention with linear biases** or **ALiBi** for short. In the ALiBi approach, the bias term is defined as the negative

<sup>16</sup>Note that, in Raffel et al. [2020]’s T5 model, the rescaling operation for the query-key product is removed. The attention weight  $\alpha(i, j)$  is then given by

$$\alpha(i, j) = \text{Softmax}(\mathbf{q}_i \mathbf{k}_j^T + u_{b(i-j)} + \text{Mask}(i, j)) \quad (8.80)$$

Entry	Query-Key Bias (PE( $i, j$ ))
T5 [Raffel et al., 2020]	$u_{b(i-j)}$
ALiBi [Press et al., 2022]	$-\beta \cdot (i - j)$
Kerple [Chi et al., 2022]	$-\beta_1 (i - j)^{\beta_2}$ (power) $-\beta_1 \log(1 + \beta_2 (i - j))$ (logarithmic)
Sandwich [Chi et al., 2023]	$\sum_{k=1}^{\bar{d}/2} \cos((i - j)/10000^{2k/\bar{d}})$
FIRE [Li et al., 2024b]	$f(\psi(i - j)/\psi(\max(m_{\text{len}}, i)))$

Table 8.4: Query-key biases as relative positional embeddings.  $\beta$ ,  $\beta_1$ ,  $\beta_2$ ,  $\bar{d}$ , and  $m_{\text{len}}$  are hyper-parameters. In the T5 model,  $b(i - j)$  denotes the bucket assigned to  $i - j$ . In the FIRE model,  $\psi(\cdot)$  is a monotonically increasing function such as  $\psi(x) = \log(cx + 1)$ , and  $f(\cdot)$  is an FFN.

scaled query-key offset

$$\begin{aligned} \text{PE}(i, j) &= -\beta \cdot (i - j) \\ &= \beta \cdot (j - i) \end{aligned} \quad (8.82)$$

where  $\beta$  is the scaling factor. Adding this term to the query-key product, we obtain a new form of attention weights

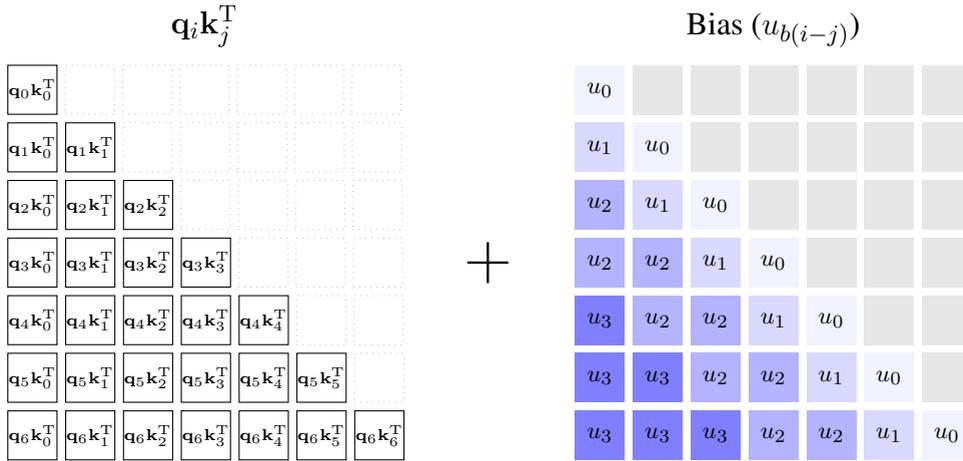
$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + \beta \cdot (j - i)}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (8.83)$$

This model can be interpreted as adding a fixed penalty to  $\mathbf{q}_i \mathbf{k}_j^T$  whenever  $j$  moves one step away from  $i$ . So we do not need to adapt it to a range of sequence lengths, and can employ it to model arbitrarily long sequences. See Figure 8.11 for a comparison of the T5 bias and the ALiBi bias.

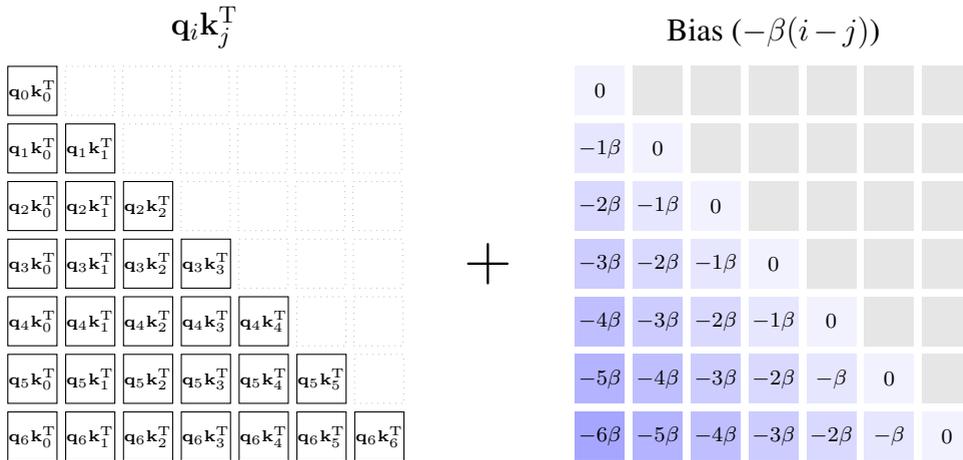
In general, the scalar  $\beta$  should be tuned on a validation dataset. However, Press et al. [2022] found that setting  $\beta$  to values decreasing geometrically by a factor of  $\frac{1}{2^a}$  for multi-head attention performs well on a variety of tasks. Specifically, for a self-attention sub-layer involving  $n_{\text{head}}$  heads, the scalar for the  $k$ -th head is given by

$$\beta_k = \frac{1}{2^{\frac{k}{n_{\text{head}}}}} \quad (8.84)$$

The ALiBi approach provides a simple form of relative positional embeddings. There are other similar methods for designing query-key biases using the offset  $i - j$ . Table 8.4 shows a comparison of such biases. As an aside it is worth noting that the form of the right-hand side of Eq. (8.82) is very similar to length features used in conventional feature-based systems. For example, in statistical machine translation systems, such features are widely used to model word reordering problems, resulting in models that can generalize well across different translation tasks [Koehn, 2010].



(a) The T5 bias ( $n_b = 3$  and  $\text{dist}_{\max} = 5$ )



(b) The ALiBi bias

Figure 8.11: Query-key products with biases (above = the T5 bias and below = the ALiBi bias). The color scale of the biases ranges from light blue denoting small absolute values to deep blue denoting large absolute values.

### 3. Rotary Positional Embedding

As with sinusoidal embeddings, rotary positional embeddings are based on hard-coded values for all dimensions of an embedding [Su et al., 2024]. Recall that in the sinusoidal embedding model, positions are represented as combinations of sine and cosine functions with different frequencies. These embeddings are then added to token embeddings to form the inputs to the Transformer layer stack. Rotary positional embeddings instead model positional context as rotations to token embeddings in a complex space. This leads to a model expressed in the form

of multiplicative embeddings

$$\mathbf{e}_i = \mathbf{x}_i R(i) \quad (8.85)$$

where  $R(i) \in \mathbb{R}^{d \times d}$  is the rotation matrix representing the rotations performed on the token embedding  $\mathbf{x}_i \in \mathbb{R}^d$ .

For simplicity, we will first consider embeddings with only two dimensions and return to a discussion of the more general formulation later. Suppose we have a 2-dimensional token embedding  $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ . We can represent it as a vector in a plane, originating at the origin  $(0,0)$  and terminating at  $(x_1, x_2)$ . A counterclockwise rotation of this vector refers to an operation of moving the vector around the origin while maintaining its magnitude, as shown in Figure 8.12 (a). The degree of rotation is usually defined by a specific angle, denoted by  $\theta$ . The rotation can be expressed mathematically in the form

$$\begin{aligned} \text{Ro}(\mathbf{x}, \theta) &= \mathbf{x} R_\theta \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta \cdot x_1 - \sin \theta \cdot x_2 & \sin \theta \cdot x_1 + \cos \theta \cdot x_2 \end{bmatrix} \end{aligned} \quad (8.86)$$

where  $R_\theta = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$  is the rotation matrix. If two or more rotations are performed on the same vector, we can rotate the vector further. This follows from the fact that the composition of successive rotations is itself a rotation. More formally, rotating a vector by an angle  $\theta$  for  $t$  times can be expressed as

$$\begin{aligned} \text{Ro}(\mathbf{x}, t\theta) &= \mathbf{x} R_{t\theta} \\ &= \begin{bmatrix} \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 & \sin t\theta \cdot x_1 + \cos t\theta \cdot x_2 \end{bmatrix} \end{aligned} \quad (8.87)$$

If we interpret  $t$  as the position of a token represented by  $\mathbf{x}$  in a sequence, then we will find that the above equation defines a simple positional embedding model. As shown in Figure 8.12 (b), we start moving the token from position 0. Each time we move one step forward, the vector is rotated by the angle  $\theta$ . Upon arriving at the position  $t$ , the representation of the token with positional context is given by  $\text{Ro}(\mathbf{x}, i\theta)$ . As the rotations do not change the magnitude of the embedding, the original “meaning” of the token is retained. The positional information is injected into the embedding, when it gets rotated.

A popular way to understand vector rotation is to define it in complex spaces. It is easy to transform each vector  $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$  in the 2D Euclidean space  $\mathbb{R}^2$  to a complex number  $\mathbf{x}' = x_1 + ix_2$  in the complex space  $\mathbb{C}$  via a bijective linear map. Then, the rotation of  $\mathbf{x}$  with the angle  $t\theta$  corresponds to the multiplication by  $e^{it\theta}$ . Given that  $e^{it\theta} = \cos t\theta + i \sin t\theta$ , the

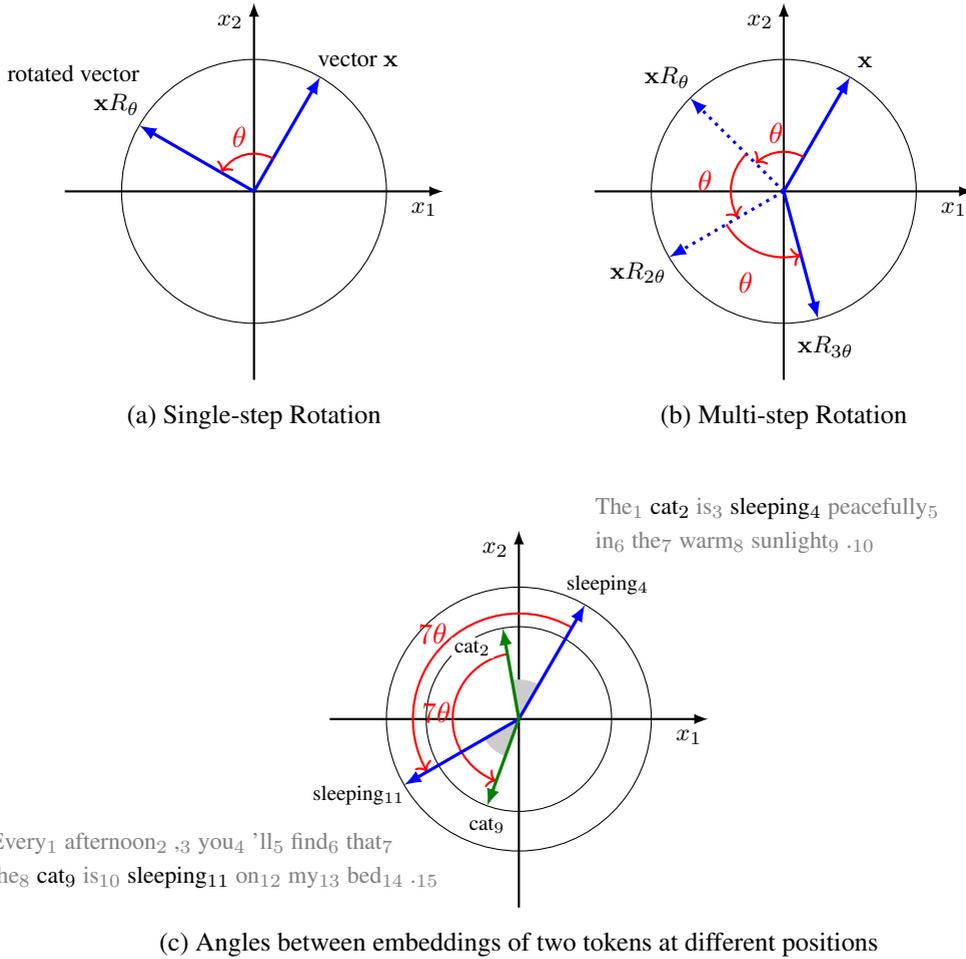


Figure 8.12: Illustrations of vector rotations in a plane. Sub-figures (a) and (b) show rotations of a vector in a single step and multiple steps, respectively. Sub-figure (c) shows the embeddings of tokens *cat* and *sleeping* in two different sentences. We show these sentences with a subscript affixed to each token to indicate its position. If we represent tokens as vectors, we can add positional information by rotating these vectors. This rotation preserves the “distances” between the vectors. For example, given that the distance between *cat* and *sleeping* is the same in both sentences, the angle between their embeddings also remains the same during rotation.

rotation operation can be re-expressed in the form

$$\begin{aligned}
 \mathbf{x}R_{t\theta} &\mapsto \mathbf{x}'e^{it\theta} \\
 &= (x_1 + ix_2)(\cos t\theta + i \sin t\theta) \\
 &= \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 + i(\sin t\theta \cdot x_1 + \cos t\theta \cdot x_2)
 \end{aligned} \tag{8.88}$$

Here we denote the token representation  $\mathbf{x}'e^{it\theta}$  by  $C(\mathbf{x}, t\theta)$ . The inner product of the represen-

tations of the tokens at positions  $t$  and  $s$  can be written as

$$\langle C(\mathbf{x}, t\theta), C(\mathbf{y}, s\theta) \rangle = (\mathbf{x}'\overline{\mathbf{y}'})e^{i(t-s)\theta} \quad (8.89)$$

where  $\overline{\mathbf{y}'}$  is the complex conjugate of  $\mathbf{y}'$ . As can be seen, the result of this inner product involves a term  $t - s$ , and so it can model the offset between the two tokens.

Now we go back to representations in the 2D Euclidean space. The dot-product of  $\text{Ro}(\mathbf{x}, t\theta)$  and  $\text{Ro}(\mathbf{y}, s\theta)$  is can be written as a function of  $(t - s)\theta$

$$\begin{aligned} \text{Ro}(\mathbf{x}, t\theta)[\text{Ro}(\mathbf{y}, s\theta)]^T &= \mathbf{x}R_{t\theta}[\mathbf{y}R_{s\theta}]^T \\ &= \mathbf{x}R_{t\theta}[R_{s\theta}]^T\mathbf{y}^T \\ &= \mathbf{x}R_{(t-s)\theta}\mathbf{y}^T \end{aligned} \quad (8.90)$$

Given this result, if we consider  $\text{Ro}(\mathbf{x}, t\theta)$  and  $\text{Ro}(\mathbf{y}, s\theta)$  as the query and the key, then the self-attention operation will implicitly involve the modeling of relative positional context.

This rotary positional embedding can be extended to multi-dimensional embeddings. For a  $d$ -dimensional token embedding  $\mathbf{x} = [x_1 \ x_2 \ \dots \ x_d]$ , we can treat it as a  $\frac{d}{2}$ -dimensional complex vector  $\mathbf{x}' = [x'_1 \ x'_2 \ \dots \ x'_{d/2}] = [x_1 + ix_2 \ x_3 + ix_4 \ \dots \ x_{d-1} + ix_d]$ , where each consecutive pair of items forms a complex number. Then, the rotary positional embedding in the complex space is given by

$$C(\mathbf{x}, t\theta) = \sum_{k=1}^{d/2} x'_k e^{it\theta_k} \vec{e}_k \quad (8.91)$$

where  $\vec{e}_k$  is the standard basis vector with a single non-zero value in the  $k$ -th coordinate and 0's elsewhere [Biderman et al., 2021].

Although this formula involves a complicated expression, its equivalent form in the  $d$ -dimensional Euclidean space is relatively easy to understand. We can write it as

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix} \begin{bmatrix} R_{t\theta_1} & & & \\ & R_{t\theta_2} & & \\ & & \ddots & \\ & & & R_{t\theta_{d/2}} \end{bmatrix} \quad (8.92)$$

where  $R_{t\theta_k} = \begin{bmatrix} \cos t\theta_k & \sin t\theta_k \\ -\sin t\theta_k & \cos t\theta_k \end{bmatrix}$ .  $\theta = [\theta_1, \dots, \theta_{d/2}]$  are the parameters for controlling the angles of rotations in different dimensions. Typically,  $\theta_k$  is set to  $10000^{-\frac{2(k-1)}{d}}$ , which is analogous to the setting in sinusoidal embeddings.

In a practical implementation, Eq. (8.92) can be rewritten into a form that relies solely on

the element-wise product and addition of vectors.

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d-1} \\ x_d \end{bmatrix}^T \odot \begin{bmatrix} \cos t\theta_1 \\ \cos t\theta_1 \\ \vdots \\ \cos t\theta_{d/2} \\ \cos t\theta_{d/2} \end{bmatrix}^T + \begin{bmatrix} -x_2 \\ x_1 \\ \vdots \\ -x_d \\ x_{d-1} \end{bmatrix}^T \odot \begin{bmatrix} \sin t\theta_1 \\ \sin t\theta_1 \\ \vdots \\ \sin t\theta_{d/2} \\ \sin t\theta_{d/2} \end{bmatrix}^T \quad (8.93)$$

Finally, we rewrite Eq. (8.85) to obtain the form of the embedding at position  $i$

$$\mathbf{e}_i = \text{Ro}(\mathbf{x}_i, i\theta) \quad (8.94)$$

#### 4. Position Interpolation

In position interpolation, our goal is to map the positions in the new sequence to match the observed range in training. Suppose the sequence length for training ranges from 0 to  $m_l$ . When  $m > m_l$  at test time, we represent the positions in  $[0, m]$  such that our representations fit  $[0, m_l]$ .

To illustrate, consider the rotary positional embedding model described above. The embedding of each token is described by a model  $\text{Ro}(\mathbf{x}_i, i\theta)$  in which  $\theta = [\theta_1, \dots, \theta_{d/2}]$  are the parameters.  $\text{Ro}(\mathbf{x}_i, i\theta)$  can be cast in the form of a linear combination of two periodic functions (see Eq. (8.93))

$$\cos i\theta = \begin{bmatrix} \cos i\theta_1 & \dots & \cos i\theta_{d/2} \end{bmatrix} \quad (8.95)$$

$$\sin i\theta = \begin{bmatrix} \sin i\theta_1 & \dots & \sin i\theta_{d/2} \end{bmatrix} \quad (8.96)$$

$\theta_k$  is an exponential function of  $k$  and takes the form

$$\theta_k = b^{-\frac{2(k-1)}{d}} \quad (8.97)$$

where  $b$  is the base. The period of  $\cos i\theta_k$  and  $\sin i\theta_k$  is

$$T_k = 2\pi \cdot b^{\frac{2(k-1)}{d}} \quad (8.98)$$

The key idea behind position interpolation is to adjust this period so that the new positions can be encoded within the range  $[0, m_l]$ . One way to achieve this is to scale up  $T_k$  by  $\frac{m}{m_l}$ , given by

$$T'_k = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(k-1)}{d}} \quad (8.99)$$

Hence all points in  $[0, m]$  are compressed into  $[0, m_l]$ . This linear scaling can be easily realized by modifying the input to the embedding model [Chen et al., 2023c]. The new model with

linear positional interpolation is given by

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}\left(\mathbf{x}_i, \frac{m_l}{m}i\theta\right) \quad (8.100)$$

Another method of positional interpolation is to scale the base<sup>17</sup>. Suppose that the base  $b$  is scaled by  $\lambda$ . We wish the period of this new model in the last dimension of  $\theta$  (i.e., dimension  $\frac{d}{2}$ ) to be equal to that of the linear positional interpolation model. This can be expressed as

$$2\pi \cdot (\lambda b)^{\frac{2(\frac{d}{2}-1)}{d}} = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(\frac{d}{2}-1)}{d}} \quad (8.101)$$

Solving this equation, we obtain

$$\begin{aligned} \lambda &= \left(\frac{m}{m_l}\right)^{\frac{d}{2(\frac{d}{2}-1)}} \\ &= \left(\frac{m}{m_l}\right)^{\frac{d}{d-2}} \end{aligned} \quad (8.102)$$

This gives an embedding model

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}(\mathbf{x}_i, i\theta') \quad (8.103)$$

where

$$\theta' = \left[ (\lambda b)^{-\frac{0}{d}}, (\lambda b)^{-\frac{2}{d}}, \dots, (\lambda b)^{-\frac{d-2}{d}} \right] \quad (8.104)$$

Note that scaling the base provides a non-uniform method for scaling the periods across different dimensions of  $\theta$ . This method has been found to be helpful for extending LLMs to longer sequences, and several improvements have been developed [Peng et al., 2024; Ding et al., 2024].

### 8.3.6 Remarks

In this section, we have presented a variety of methods for long-context language modeling. We close this section by discussing some interesting issues related to these methods.

#### 1. Need for Long Context

One of the ultimate goals of long-context LLMs is that these models can precisely encode infinite context. The so-called infinite context refers more to the fact that an LLM can continuously read words. This motivates LLMs that can handle extremely long context or stream data. As discussed in Section 8.3.3, it is common to use fixed-size memory models to process continuously expanding context. Many such systems are based on recurrent architectures or their variants, because they are inherently suited to model time series problems where

<sup>17</sup>This method was first proposed in [https://www.reddit.com/r/LocalLLaMA/comments/141z7j5/ntkaware\\_scaled\\_rope\\_allows\\_llama\\_models\\_to\\_have/](https://www.reddit.com/r/LocalLLaMA/comments/141z7j5/ntkaware_scaled_rope_allows_llama_models_to_have/)

the effects of past inputs continue indefinitely. Another way to achieve infinite memory is to develop alternatives to self-attention models, for example, one can use continuous-space attention models to encode context, which removes the dependency on context length [Martins et al., 2022].

When studying long-context LLMs, it is natural to wonder what mechanisms may explain the use of long context in language modeling. Can we compress the representation of infinite context into a relatively small-sized model? Are all context tokens useful for predicting next tokens? How do LLMs prepare for token prediction when they see the context? Can we know in advance which contextual information will be critical for prediction? General answers to all these questions are not obvious, but they inspire follow-on research of explainable models, and some interesting results have been found. For example, Deletang et al. [2024] conducted extensive experiments to show that LLMs are powerful in-context compressors. Although viewing predictive models as compression models has long been studied in machine learning, it also provides insights into our understanding of the LLM scaling laws. Pal et al. [2023] and Wu et al. [2024] investigated whether the features learned up to the current step, though not intentionally, are already sufficient for predicting tokens at the following steps. Note that the need for long-context in language modeling is highly dependent on the problem that we address. A related issue is where to apply LLMs and how to evaluate them. For example, in summarization tasks we may only need to distill and focus on a few key aspects of the text, while in retrieval-like tasks we need to “memorize” the entire context so that the relevant information can be accessed. We will discuss the evaluation issue later in this subsection.

## 2. Pre-training or Adapting LLMs?

Training LLMs requires significant computational costs. Although it is straightforward to train LLMs on long sequence data, the training becomes computationally unwieldy for large data sets. It is common practice to pre-train LLMs on general datasets, and then adapt them with modest fine-tuning effort. For example, LLMs with relative or rotary positional embeddings can be directly trained on large-scale data in the pre-training phase. While the resulting models may exhibit some abilities to extrapolate lengths in the inference phase, it may be more effective to fine-tune them on longer sequences.

Ideally, we would like to pre-train LLMs with standard Transformer architectures and adapt them to new tasks. This allows us to use many off-the-shelf LLMs and efficiently adapt them to handle long sequences. However, when new architectures are adopted, it seems inevitable that we need to train these models from scratch. This poses practical difficulties for developing long-context LLMs, as we cannot leverage well-developed, pre-trained models and must instead train them ourselves. On the other hand, fine-tuning is still an effective way to adapt LLMs with certain architectures that are different from those in pre-training. An example is models augmented with external memories. In these models, the pre-trained LLMs are fixed, and the focus is on how to make these LLMs collaborate with the memory models. In RAG, for instance, it is common to fine-tune LLMs to improve their use of retrieval-augmented inputs. Another example of fine-tuning LLMs for long-context modeling is that we train an LLM with full attention models, and then replace them with sparse attention models in the fine-tuning

phase. The pre-trained LLM provides initial values of model parameters used in a different model, and this model is then fine-tuned as usual.

### 3. Evaluating Long-context LLMs

Evaluating long-context LLMs is important, but it is a new issue in NLP. The general idea is that, if we input a long context to an LLM, then we can check from the output of the LLM whether it understands the entire context and makes use of it in predicting following tokens. In conventional research of NLP, such evaluations are often aimed at examining the ability of NLP models in handling long-range dependencies. However, the size of contexts used in recent LLMs is much larger than that used in NLP systems a few years ago. This motivates researchers to develop new evaluation benchmarks and metrics for long-context LLMs.

One approach is to use the perplexity metric. However, in spite of its apparent simplicity, this method tends to reflect more on the LLMs' ability to make use of local context rather than global context. It is therefore tempting to develop evaluation methods that are specific to long-context LLMs. Popular methods include various synthetic tasks where artificially generated or modified data is used to evaluate specific capabilities of long-context LLMs. In needle-in-a-haystack<sup>18</sup> and passkey retrieval tasks [Mohtashami and Jaggi, 2024; Chen et al., 2023c], for instance, LLMs are required to identify and extract a small, relevant piece of information from a large volume of given text. The assumption here is that an LLM with sufficient memory should remember earlier parts of the text as it processes new information. This LLM can thus pick out the relevant details, which might be sparse and hidden among much irrelevant information, from the text. Alternatively, in copy memory tasks (or copy tasks for short), LLMs are used to repeat the input text or a specific segment multiple times. These tasks were initially proposed to test the extent to which recurrent models can retain and recall previously seen tokens [Hochreiter and Schmidhuber, 1997; Arjovsky et al., 2016], and have been adopted in evaluating recent LLMs [Bulatov et al., 2022; Gu and Dao, 2023].

Another approach to evaluating long-context LLMs is to test them on NLP tasks that involve very long input sequences. Examples include long-document or multi-document summarization, long-document question answering, code completion, and so on. A benefit of this approach is that it can align evaluations with user expectations.

Although many methods have been developed, there is still no general way to evaluate long-context LLMs [Liu et al., 2024c]. One problem is that most of these methods focus on specific aspects of LLMs, rather than their fundamental ability to model very long contexts. Even though an LLM can pick out the appropriate piece of text from the input, we cannot say that it truly understands the entire context. Instead, it might just remember some important parts of the context, or even simply recall the answer via the model learned in pre-training. Moreover, the data used in many tasks is small-scale and relatively preliminary, leading to discrepancies between evaluation results and actual application performance. A more interesting issue is that the results of LLMs are influenced by many other factors and experimental setups, for example, using different prompts can lead to very different outcomes. This makes evaluation even more

---

<sup>18</sup>[https://github.com/gkamradt/LLMTest\\_NeedleInAHaystack](https://github.com/gkamradt/LLMTest_NeedleInAHaystack)

challenging because improvements may not solely result from better modeling of long contexts, and there is a risk of overclaiming our results. Nevertheless, many open questions remain in the development and evaluation of long-context LLMs. For example, these models still suffer from limitations such as restricted context length and high latency. Studying these issues is likely to prove valuable future directions.

## 8.4 Summary

In this chapter, we have discussed the concept of LLMs and related techniques. This can be considered a general, though not comprehensive, introduction to LLMs, laying the foundation for further discussions on more advanced topics in subsequent chapters. Furthermore, we have explored two ways to scale up LLMs. The first focuses on the large-scale pre-training of LLMs, which is crucial for developing state-of-the-art models. The second focuses on methods for adapting LLMs to long inputs, including optimizing attention models, designing more efficient and compressed KV caches, incorporating memory models, and exploring better positional embeddings.

The strength of LLMs lies in their ability to break the constraints of training NLP models for a limited number of specific tasks. Instead, LLMs learn from large amounts of text through the simple task of token prediction — we predict the next token in a sentence given its prior tokens. A general view is that, by repeating this token prediction task a large number of times, LLMs can acquire some knowledge of the world and language, which can then be applied to new tasks. As a result, LLMs can be prompted to perform any task by framing it as a task of predicting subsequent tokens given prompts. This emergent ability in language models comes from several dimensions, such as scaling up training, model size, and context size. It is undeniable that scaling laws are currently the fundamental principle adopted in developing large language models, although simply increasing model size has yet to prove sufficient for achieving AGI. These continuously scaled LLMs have been found to show capabilities in general-purpose language understanding, generation, and reasoning. More recently, it has been found that scaling up the compute at inference time can also lead to significant improvements in complex reasoning tasks [OpenAI, 2024].

Given their amazing power, LLMs have attracted considerable interest, both in terms of techniques and applications. As a result, the explosion of research interest in LLMs has also led to a vast number of new techniques and models. However, we do not attempt to provide a comprehensive literature review on all aspects of LLMs, given the rapid evolution of the field. Nevertheless, one can still gain knowledge about LLMs from general reviews [Zhao et al., 2023; Minaee et al., 2024] or more focused discussions on specific topics [Ruan et al., 2024].

# Chapter 9

## Prompting

In the context of LLMs, *prompting* refers to the method of providing an LLM with a specific input or cue to generate a desired output or perform a task. For example, if we want the LLM to translate a sentence from English to Chinese, we can prompt it like this

Translate the text from English to Chinese.

Text: The early bird catches the worm.

Translation: \_\_\_\_\_

Prompting is crucial for LLMs because it directly influences how effectively these models understand and respond to user queries. A well-crafted prompt can guide an LLM to generate more accurate, relevant, and contextually appropriate responses. Furthermore, this process can be iteratively refined. By analyzing the responses of the LLM, users can adjust their prompts to align more closely with their specific needs. Given the importance of prompting in applying LLMs, prompt design has become an essential skill for users and developers working with LLMs. This leads to an active research area, called **prompt engineering**, in which we design effective prompts to make better use of LLMs and enhance their practical utility in real-world applications.

An important concept related to prompting is **in-context learning**. When prompting an LLM, we can add new information to the context, such as demonstrations of problem-solving. This allows the LLM to learn from this context how to solve the problem. Here is an example of prompting LLMs with a few demonstrations of how to classify text based on sentiment polarity.

Here are some examples of text classification.

Example 1: We had a delightful dinner together. → Label: Positive

Example 2: I'm frustrated with the delays. → Label: Negative

What is the label for “That comment was quite hurtful.”?

Label: \_\_\_\_\_

In-context learning is often seen as an emergent ability of LLMs that arises after pre-training. Though LLMs can be trained or tuned to perform new tasks, in-context learning provides a very efficient way to adapt these models without any training or tuning effort. Perhaps this is one of the most notable features of LLMs: they indeed learn general knowledge about the world and language during pre-training, which we can easily apply to new challenges. Moreover, in-context learning reflects the broader trend of making AI systems more generalizable and user-friendly. Instead of requiring specialized engineers to fine-tune models for every unique task, users can interact with LLMs in a more intuitive way, simply providing examples or adjusting the context as needed.

In this chapter, we focus on prompting techniques in LLMs. We begin by considering several interesting prompt designs commonly used in prompt engineering. Then, we discuss a series of refinements to these methods. Finally, we explore approaches for automating prompt design.

## 9.1 General Prompt Design

This section presents basic concepts in prompt design, along with examples of how to prompt LLMs for various NLP tasks. Since the effectiveness of prompting is highly dependent on the LLMs being used, prompts often vary across different LLMs, making it difficult to provide a comprehensive list of prompts for all LLMs and downstream tasks. Therefore, this discussion is not focused on any specific LLM. Instead, the goal is to provide guiding principles for prompt design.

### 9.1.1 Basics

The term *prompt* is used in many different ways. In this chapter we define a prompt as the input text to an LLM, denoted by  $\mathbf{x}$ . The LLM generates a text  $\mathbf{y}$  by maximizing the probability  $\Pr(\mathbf{y}|\mathbf{x})$ . In this generation process, the prompt acts as the condition on which we make predictions, and it can contain any information that helps describe and solve the problem.

A prompt can be obtained using a prompt template (or template for short) [Liu et al., 2023b]. A template is a piece of text containing placeholders or variables, where each placeholder can be filled with specific information. Here are two templates for asking the LLM for weekend suggestions.

Please give me some suggestions for a fun weekend.

\_\_\_\_\_

If {*\*premise\**}, what are your suggestions for a fun weekend.

\_\_\_\_\_

In the first template, we simply instruct the LLM to return some suggestions. So the template is just a piece of text with no variables. In the second template, the variable {*\*premise\**} needs to be specified by the users to provide a premise for making suggestions. For example, if we input

premise = the weather is nice this weekend

then we can generate a prompt

If the weather is nice this weekend,  
what are your suggestions for a fun weekend.

\_\_\_\_\_

We can also design a template with multiple variables. Here is an example in which we compare the two sentences in terms of their semantic similarity.

Here is a sentence  
{*\*sentence1\**}  
Here is another sentence  
{*\*sentence2\**}  
Compute the semantic similarity between the two sentences

\_\_\_\_\_

A popular way to format prompts is to write each input or output in a “name:content” style. For example, we can describe a conversation between two people, named John and David, and use the LLM to continue the conversation. A template of such prompts is given by

```

John: {*utterance1*}
David: {*utterance2*}
John: {*utterance3*}
David: {*utterance4*}
John: {*utterance5*}
David: {*utterance6*}
John: {*utterance7*}
David: _____

```

The “name:content” format can be used to define the task that we want the LLM to perform. For example, given that “Q” and “A” are commonly used abbreviations for “Question” and “Answer”, respectively, we can use the following template to do question-answering.

```

Q: {*question*}
A: _____

```

This format can be used to describe more complex tasks. For example, the following is an example of providing a specification for a translation task

```

Task: Translation
Source language: English
Target language: Chinese
Style: Formal text
Template: Translate the following sentence: {*sentence*}
_____

```

In practical systems, it is common to represent and store such data in key-value pairs, such as the JSON format<sup>1</sup>.

When the problem is difficult to describe in an attribute-based manner, it is more common to instruct LLMs with a clear and detailed description. There are many ways to do this. One

---

<sup>1</sup>The JSON representation is

```

{
  "Task": "Translation"
  "Source language": "English"
  "Target language": "Chinese"
  "Style": "Formal text"
  "Template": "Translate the following sentence: {*sentence*}"
}

```

example is to assign a role to LLMs and provide sufficient context. The following is a template that instructs an LLM to act as an expert and answer questions from children.

```
You are a computer scientist with extensive knowledge in the field
of deep learning.
```

```
Please explain the following computer-related concept to a child
around 10 years old, using simple examples whenever possible.
```

```
{*concept*}
```

```
_____
```

Here the text “You are a computer scientist ... deep learning. ” is sometimes called system information, and is provided to help the LLM understand the context or constraints of the task it is being asked to perform.

### 9.1.2 In-context Learning

Learning can occur during inference. In-context learning is one such method, where prompts involve demonstrations of problem-solving, and LLMs can learn from these demonstrations how to solve new problems. Since we do not update model parameters in this process, in-context learning can be viewed as a way to efficiently activate and reorganize the knowledge learned in pre-training without additional training or fine-tuning. This enables quick adaptation of LLMs to new problems, pushing the boundaries of what pre-trained LLMs can achieve without task-specific adjustments.

In-context learning can be illustrated by comparing three methods: zero-shot learning, one-shot learning and few-shot learning. Zero-shot learning, as its name implies, does not involve a traditional “learning” process. It instead directly applies LLMs to address new problems that were not observed during training. In practice, we can repetitively adjust prompts to guide the LLMs in generating better responses, without demonstrating problem-solving steps or providing examples. Consider the following example. Suppose we want to use an LLM as an assistant that can help correct English sentences. A zero-shot learning prompt is given by

```
SYSTEM You are a helpful assistant, and are great at grammar correction.
```

```
USER You will be provided with a sentence in English. The task is
to output the correct sentence.
```

```
Input: She don't like going to the park.
```

```
Output: _____
```

Here the gray words are used to indicate different fields of the prompt.

In one-shot learning, we extend this prompt by adding a demonstration of how to correct sentences, thereby allowing the LLM to learn from this newly-added experience.

SYSTEM You are a helpful assistant, and are great at grammar correction.

DEMO You will be provided with a sentence in English. The task is to output the correct sentence.

Input: There is many reasons to celebrate.

Output: There are many reasons to celebrate.

USER You will be provided with a sentence in English. The task is to output the correct sentence.

Input: She don't like going to the park.

Output: \_\_\_\_\_

Furthermore, we can add more demonstrations to enable few-shot learning.

SYSTEM You are a helpful assistant, and are great at grammar correction.

DEMO1 You will be provided with a sentence in English. The task is to output the correct sentence.

Input: There is many reasons to celebrate.

Output: There are many reasons to celebrate.

DEMO2 You will be provided with a sentence in English. The task is to output the correct sentence.

Input: Me and my friend goes to the gym every day.

Output: My friend and I go to the gym every day.

USER You will be provided with a sentence in English. The task is to output the correct sentence.

Input: She don't like going to the park.

Output: \_\_\_\_\_

In few-shot learning, we essentially provide a pattern that maps some inputs to the corresponding outputs. The LLM attempts to follow this pattern in making predictions, provided that the prompt includes a sufficient number of demonstrations, although generally small. It is also possible to use simpler patterns to achieve this. For example, one can use the following few-shot learning prompt for translating words from Chinese to English.

DEMO	现在	→	now
	来	→	come
	去	→	go
	男孩	→	boy
USER	女孩	→	_____

If the LLM is powerful enough, few-shot learning can enable it to address complex problems, such as mathematical reasoning. For example, consider the following task of summing two numbers and then dividing the sum by their product.

DEMO	12 5	→	$(12 + 5)/(12 \times 5) = 0.283$
	3 1	→	$(3 + 1)/(3 \times 1) = 1.33$
	-9 4	→	$(-9 + 4)/(-9 \times 4) = 0.138$
	15 15	→	$(15 + 15)/(15 \times 15) = 0.133$
USER	19 73	→	_____

In many practical applications, the effectiveness of in-context learning relies heavily on the quality of prompts and the fundamental abilities of pre-trained LLMs. On one hand, we need a significant prompt engineering effort to develop appropriate prompts that help LLMs learn more effectively from demonstrations. On the other hand, stronger LLMs can make better use of in-context learning for performing new tasks. For example, suppose we wish to use an LLM to translate words from Inuktitut to English. If the LLM lacks pre-training on Inuktitut data, its understanding of Inuktitut will be weak, and it will be difficult for the model to perform well in translation regardless of how we prompt it. In this case, we need to continue training the LLM with more Inuktitut data, rather than trying to find better prompts.

It might be interesting to explore how in-context learning emerges during pre-training and why it works during inference. One simple understanding is that LLMs have gained some knowledge of problem-solving, but there are many possible predictions, which are hard to distinguish when the models confront new problems. Providing demonstrations can guide the LLMs to follow the “correct” paths. Furthermore, some researchers have tried to interpret in-context learning from several different perspectives, including Bayesian inference [Xie et al., 2022], gradient descent [Dai et al., 2023; Von Oswald et al., 2023], linear regression [Akyürek et al., 2023], meta learning [Garg et al., 2022], and so on.

### 9.1.3 Prompt Engineering Strategies

Designing prompts is highly empirical. In general, there are many ways to prompt an LLM for performing the same task, and we need to perform a number of trial-and-error runs to find a satisfactory prompt. To write good prompts more efficiently, one can follow certain strategies. Examples of common prompting principles include

- **Describing the task as clearly as possible.** When we apply an LLM to solve a problem, we need to provide a precise, specific, and clear description of the problem and instruct the LLM to perform as we expect. This is particularly important when we want the output of the LLM to meet certain expectations. For example, suppose we are curious about climate change. A simple prompt for asking the LLM to provide some information is

Tell me about climate change.

\_\_\_\_\_

Since this instruction is too general, the LLM may generate a response that addresses any aspect of climate change, which may not align with our specific interests. In this case, we can instead use prompts that are specific and detailed. One such example is

Provide a detailed explanation of the causes and effects of climate change, including the impact on global temperatures, weather patterns, and sea levels. Also, discuss possible solutions and actions being taken to mitigate these effects.

\_\_\_\_\_

Now suppose we intend to explain climate change to a 10-year-old child. We can adjust the above prompt further.

Explain the causes and effects of climate change to a 10-year-old child. Talk about how it affects the weather, sea levels, and temperatures. Also, mention some things people are doing to help. Try to explain in simple terms and do not exceed 500 words.

\_\_\_\_\_

- **Guiding LLMs to think.** LLMs have exhibited surprisingly good capabilities to “think”. A common example is that well-developed LLMs have achieved impressive performance in mathematical reasoning tasks, which are considered challenging. In prompt engineering, the “thinking” ability of LLMs needs to be activated through appropriate prompting, especially for problems that require significant reasoning efforts. In many cases, an LLM that is instructed to “think” can produce completely different results compared with the same LLM that is instructed to perform the task straightforwardly. For example, [Kojima et al. \[2022\]](#) found that simply appending “Let’s think step by step” to the end of each prompt can improve the performance of LLMs on several reasoning tasks. LLMs can be prompted to “think” in a number of ways. One method is to instruct LLMs to

generate steps for reasoning about the problem before reaching the final answer. For example, consider a task of solving mathematical problems. See below for a simple prompt for this task.

You are a mathematician. You will be provided with a math problem.  
Please solve the problem.

\_\_\_\_\_

Since solving math problems requires a detailed reasoning process, LLMs would probably make mistakes if they attempted to work out the answer directly. So we can explicitly ask LLMs to follow a given reasoning process before coming to a conclusion.

You are a mathematician. You will follow these detailed reasoning steps when solving math problems.

Step 1: Problem Interpretation.

The mathematician carefully listens to your query and understands the intricate details of the mathematical challenge you have presented.

Step 2: Strategy Formulation.

Drawing upon their extensive knowledge, the mathematician chooses the most effective strategy tailored to the type of math problem, whether it is algebra, calculus, or geometry.

Step 3: Detailed Calculation.

With precision and expertise, the mathematician performs the necessary calculations step by step, adhering to all mathematical principles.

Step 4: Solution Review.

Before providing the final answer, the mathematician meticulously checks the calculations for accuracy and offers a concise explanation or rationale for the solution.

You will be provided with a math problem. Please solve the problem.

{\*problem\*}

\_\_\_\_\_

Another method to guide LLMs to “think” is through multiple rounds of interaction with LLMs. For example, as a first step, we can instruct LLMs to solve the problem directly

You will be provided with a math problem. Please solve the problem.  
{\*problem\*}

Now we have an initial answer to the problem. As a second step, we prompt LLMs to evaluate the correctness of the answer and, if necessary, rework it to find a better solution.

You will be provided with a math problem, along with a solution.  
Evaluate the correctness of this solution, and identify any errors if present. Then, work out your own solution.  
Problem: {\*problem\*}  
Solution: {\*solution\*}

The prompts presented here are closely related to a long line of research on reasoning problems in LLMs. It is impossible to provide a complete discussion of all related issues because this topic covers a large family of methods. But we will see a relatively more detailed discussion on how to improve prompting through more reasoning in [Section 9.2](#).

- **Providing reference information.** As discussed in the previous section, we can include demonstrations in prompts and allow LLMs to in-context learn from these demonstrations how to perform the task. In fact, given the remarkable ability of language understanding of LLMs, we can add any type of text into the prompts and so these models can predict based on enriched contexts. In many applications, we have various information that is relevant to user queries. Instead of using LLMs to make unconstrained predictions, we often want LLMs to produce outputs that are confined to the relevant text. One such example is RAG, where the relevant text for the user query is provided by calling an IR system, and we prompt LLMs to generate responses based on this provided relevant text. The following prompt shows an example.

You are an expert that can generate answers to input queries. You have now been provided with a query and the corresponding context information. Please generate an answer based on this context information. Note that you need to provide the answer in your own words, not just copy from the context provided.

Context information: `{*IR-result*}`

Query: `{*query*}`

---

If the context information is highly reliable, we can even restrict LLMs to answering using only the provided text. An example prompt is shown as follows

You are an expert tasked with generating answers from input queries. You have been provided with a query and corresponding context information, organized in a table where each row represents a useful record. Please generate an answer using only this context information. Ensure that you provide the answer in your own words.

Context information: `{*table*}`

Query: `{*query*}`

---

When dealing with real-world problems, we often have prior knowledge and additional information about the problems that help produce better answers. Considering such information in prompting is generally helpful in improving the result.

- **Paying attention to prompt formats.** In general, the performance of LLMs is highly sensitive to the prompts we input. Sometimes a small modification to a prompt can lead to a big change in model output. An interesting example is that changing the order of sentences in a prompt may cause LLMs to generate different results. To make prompts easy to read and reduce ambiguity, it is common to format them in a way that ensures clarity. One example is that we define several fields for prompts and fill different information in each field. Another example is we can use code-style prompts for LLMs which can understand and generate both natural language and code. See the following for a code-style prompt that performs translation where one demonstration is presented.

```
[English] = [I have an apple.]  
[German] = [Ich habe einen Apfel.]  
[English] = [I have an orange.]  
[German] = ____
```

LLMs can receive text in various formats. This allows us to use control characters, XML tags, and specific formatting to represent complex data. And it is useful to specify how the input and output should be formatted or structured. For example, we can delimit sections of text using quotes and prompt LLMs accordingly (e.g., adding a sentence like “the input text is delimited by double quotes” to the prompt).

Above, we have discussed only a few strategies for writing good prompts. There are, of course, many such methods, and one needs to develop their own through practice. Interested readers can refer to various online documents for more information, such as OpenAI’s manual on the GPT series models<sup>2</sup>.

### 9.1.4 More Examples

In this subsection, we consider more examples of prompting LLMs to perform various NLP tasks. The motivation here is not to give standard prompts for these tasks, but rather to use simple examples to illustrate how LLMs can be prompted to deal with NLP problems.

#### 1. Text Classification

Text classification is perhaps one of the most common problems in NLP. Many tasks can be broadly categorized as assigning pre-defined labels to a given text. Here we consider the polarity classification problem in sentiment analysis. We choose polarity classification for illustration because it is one of the most popular and well-defined text classification tasks. In a general setup of polarity classification, we are required to categorize a given text into one of three categories: negative, positive, or neutral. Below is a simple prompt for doing this (for easy reading, we highlight the task description in the prompt).

```
Analyze the polarity of the following text and classify it as positive, negative,  
or neutral.
```

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

The polarity of the text can be classified as negative.

<sup>2</sup>See <https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results>.

To make the example complete, we show the response generated by the LLM (underlined text).

Although the answer is correct, the LLM gives this answer not in labels but in text describing the result. The problem is that LLMs are designed to generate text but not to assign labels to text and treat classification problems as text generation problems. As a result, we need another system to map the LLM’s output to the label space (call it **label mapping**), that is, we extract “negative” from “The polarity of the text can be classified as negative”. This is trivial in most cases because we can identify label words via simple heuristics. But occasionally, LLMs may not express the classification results using these label words. In this case, the problem becomes more complicated, as we need some way to map the generated text or words to predefined label words.

One method to induce output labels from LLMs is to reframe the problem as a cloze task. For example, the following shows a cloze-like prompt for polarity classification.

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

The polarity of the text is negative

We can use LLMs to complete the text and fill the blank with the most appropriate word. Ideally, we wish the filled word would be positive, negative, or neutral. However, LLMs are not guaranteed to generate these label words. One method to address this problem is to constrain the prediction to the set of label words and select the one with the highest probability. Then, the output label is given by

$$\text{label} = \arg \max_{y \in Y} \Pr(y|\mathbf{x}) \quad (9.1)$$

where  $y$  denotes the word filled in the blank, and  $Y$  denotes the set of label words {positive, negative, neutral}.

Another method of using LLMs to generate labels is to constrain the output with prompts. For example, we can prompt LLMs to predict within a controlled set of words. Here is an example.

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

What is the polarity of the text?

Just answer: positive, negative, or neutral.

Negative

Sentiment analysis is a common NLP problem that has probably been well understood by LLMs through pre-training or fine-tuning. Thus we can prompt LLMs using simple instructions to perform the task. However, for new classification problems, it may be necessary to provide additional details about the task, such as the classification standards, so that the LLMs can perform correctly. To do this, we can add a more detailed description of the task and/or demonstrate classification examples in the prompts. To illustrate, consider the following example.

Analyze the polarity of the following text and classify it as positive, negative, or neutral. Here's what each category represents:

**Positive:** This indicates that the text conveys a positive emotion or attitude. For example, texts expressing happiness, satisfaction, excitement, or admiration are considered positive.

**Negative:** This refers to a text that expresses a negative emotion or attitude. It encompasses feelings of sadness, anger, frustration, or criticism.

**Neutral:** Neutral sentiment is used to describe texts that do not exhibit clear positive or negative emotions but instead convey informational, factual, or indifferent tones.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

What is the polarity of the text?

Negative

While it seems straightforward to use LLMs for classification problems, there are still issues that have not been well addressed. For example, when dealing with a large number of categories, it remains challenging to effectively prompt LLMs. Note that if we face a very

difficult classification problem and have a certain amount of labeled data, fine-tuning LLMs or adopting “BERT + classifier”-like architectures is also desirable.

## 2. Information Extraction

Many NLP problems can be regarded as information extraction problems, involving the identification or extraction of specific pieces of information from unstructured text. This information can include named entities, relationships, events, and other relevant data points. The goal of information extraction is to transform raw data into a format that can be easily analyzed and used in various downstream applications.

As information extraction covers a wide range of problems, we cannot discuss them all here. Instead, we start with the task of named entity recognition — a task that has long been a concern in NLP. Named entity recognition is a process that detects and classifies key information in text into specific groups. These key pieces of information, known as named entities, typically include proper names and are categorized into distinct classes such as people, locations, organizations, dates, monetary values, and percentages. Consider first a simple example of extracting person names from a text<sup>3</sup>.

Identify all person names in the provided text.

Text:

Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it’s the latter. “I think the UK is doing perfectly well but we’ll see more people going to Europe,” he says of 2024, adding that there’s “allegedly a slight plateau of American demand for the UK.”

The person name identified in the provided text is:

Tom Jenkins

Then, we can extract all named entities.

---

<sup>3</sup>The text is from <https://edition.cnn.com/travel>

Identify and classify all named entities in the provided text into categories such as person names, locations, dates, and organizations. List each entity with its type on one line.

Text:

Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

1. Tom Jenkins - Person Name
2. European Tourism Organisation - Organization
3. UK - Location
4. Europe - Location
5. 2024 - Date

Given these identified named entities, we can further process the text using other information extraction tasks. For example, we can identify the relationships between these named entities (call it **relation extraction**). The corresponding prompt is shown as follows.

Given a text and a list of named entities identified within it, analyze and describe the relationships between these entities. Explain how each entity is contextually related to others.

Text:

Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

Named Entity:

Tom Jenkins, European Tourism Organisation, UK, Europe, 2024

Tom Jenkins - European Tourism Organisation: Tom Jenkins is the CEO of the European Tourism Organisation. This indicates a professional association where Tom Jenkins likely has a significant role in directing the organization's activities, especially those related to tourism in Europe.

...

If LLMs have been fine-tuned with instruction following for information extraction tasks, it is relatively easy to perform various information extraction tasks. For example, the following is a prompt template for information extraction.

```
You will be provided with a text. Your task is to {*task-description*}
```

```
Text: {*text*}
```

```
_____
```

Here task-description is the instruction for the task. Examples of the instruction include

*extract the keywords from the text.*

*extract the key events from the text.*

*detect the coreference and anaphoric links between entities from the text.*

and so on. Of course, we can provide a more detailed description of the task to prompt LLMs more accurately.

### 3. Text Generation

In NLP, the term *text generation* is used broadly to cover a wide range of different problems, many of which focus on generating text based on given text. These problems can be roughly categorized into two classes: 1) continual writing based on the input text (call it **text completion**), and 2) transformation of the input text into another text (call it **text transformation**). Here, we illustrate the prompts for the former and consider the prompts for the latter in the following discussion.

A simple example of text completion is continuing to write a story starting with a few given sentences.

```
You are a writer who is great at writing stories. You are provided with a few sentences to start with. Please continue the story from the provided sentences.
```

Start of the story:

One sunny afternoon, Lily discovered a tiny, golden key hidden under a rock in her backyard. Curious, she picked it up and wondered what it might unlock. Just then, her cat, Whiskers, began to meow excitedly, leading her towards the old oak tree at the edge of the garden.

Lily followed Whiskers, her heart pounding with excitement and a touch of nervousness. The old oak tree had always seemed mysterious to her, its gnarled branches casting long shadows even on the brightest days. As they reached the tree, Whiskers darted ahead and started pawing at something buried beneath a pile of autumn leaves ...

Likewise, we can create a conversation given the conversational context.

You are provided with a conversation between a tourist and a taxi driver. Please complete the conversation.

Tourist: Could you take me to the downtown museum, please?

Taxi Driver: Of course! First time in the city?

Tourist: Yes, any must-see places around?

Taxi Driver: Definitely check out the botanical gardens after the museum. They're nearby and quite beautiful.

Tourist: Great, thanks for the tip!

Taxi Driver: No problem. Enjoy your visit!

Another application of text completion is generating text based on user requirements. The requirements can include style, tone, length, and any specific content that the text should contain. For example, users may need a formal report, a creative story, or a piece of programming code. Here is a prompt for generating a five-character regulated Chinese poem.

You are a poet. Please write a traditional Chinese poem. Here is the requirement.

Theme: Spring's rejuvenation

Structure: Five-character regulated poem

Emotion: Convey a sense of happiness and renewal through vivid spring imagery.

春意盎然

翠柳风中舞，

红花雨后新。

山明水又绿，

天宽鸟自频。

If the LLM is trained to generate language and code, we can prompt it to perform code completion tasks. Here is an example.

Please write a Python function to calculate the average of a list of numbers.

```
def calculate_average(numbers):  
    if numbers: # Check if the list is not empty  
        return sum(numbers) / len(numbers)  
    else:  
        return 0 # Return 0 if the list is empty
```

#### 4. Text Transformation

Typical text transformation tasks include machine translation, summarization, text style transfer, and so on. The prompts used to perform these tasks are not significantly different from the ones presented previously. Here, we present a few examples to showcase these prompts.

Translate the following text from English to Spanish.

Text:

The quick brown fox jumps over the lazy dog.

El rápido zorro marrón salta sobre el perro perezoso.

Summarize the following article in no more than 50 words:

Article:

In recent years, urban areas have been facing significant challenges related to sustainability and environmental management. Cities are experiencing higher rates of pollution, increased traffic congestion, and greater demands on infrastructure. This growth has led to numerous environmental issues, including elevated levels of air and water pollution, increased waste production, and strained public services ...

Urban areas are grappling with sustainability challenges, such as rising pollution, traffic congestion, and infrastructure demands ...

Rewrite this text in a formal tone.

Text:

Hey, what's up? Long time no see!

Hello, how have you been? It has been quite some time since we last met!

## 5. Question Answering

The question-answering format is inherently simple. For a given question, there is an answer that corresponds to it. For example, in open-domain question answering, we expect the system to return an answer in response to a user-submitted question. Prompt templates for general-purpose question answering can be:

```
{*question*}
```

\_\_\_\_\_

```
Question: {*question*}
```

```
Answer: _____
```

```
Q: {*question*}
```

```
A: _____
```

Question answering is important in NLP because many problems can be framed as question-answering tasks. In particular, many recent reasoning tasks are defined in the form of question answering. For example, in the MMLU benchmark [Hendrycks et al., 2021], each example consists of a multiple-choice question, and LLMs are required to select the correct answer. See the following for an example prompt for answering a question in this dataset.

$(Z, *)$  is a group with  $a * b = a + b + 1$  for all  $a, b$  in  $Z$ . The inverse of  $a$  is

(A) 0

(B)  $-2$

(C)  $a - 2$

(D)  $(2 + a) * -1$

D

Another widely-used benchmark is the GSM8K dataset [Cobbe et al., 2021]. It consists of thousands of grade school math word problems. For each problem, we use LLMs to return a solution in natural language. For example, the following is a prompt used for solving a problem in GSM8K.

DEMO Q: Jerry's two daughters play softball on different teams. They each have 8 games this season. Each team practices 4 hours for every game they play. If each game lasts for 2 hours, how many hours will Jerry spend at the field watching his daughters play and practice altogether?

A: Jerry will spend 8 games \* 2 hours per game =  $\ll 8 * 2 = 16 \gg$  16 hours watching one daughter play her games. He will spend  $16 * 2 = \ll 16 * 2 = 32 \gg$  32 hours watching both daughters play their games. He will spend 8 games \* 4 hours of practice =  $\ll 8 * 4 = 32 \gg$  32 hours watching one daughter practice. He will spend  $32 * 2 = \ll 32 * 2 = 64 \gg$  64 hours watching both daughters practice. He will spend a total of 32 hours watching games + 64 hours watching practice =  $\ll 32 + 64 = 96 \gg$  96 hours. ##### 96

Q: Mary bought six apples from the store. From the apples she bought, for each that Mary ate, she planted two trees from the remaining ones. How many apples did Mary eat?

A: She planted eight trees. This means she used half of that amount of apples, which is 8 trees / 2 trees/apple =  $\ll 8 / 2 = 4 \gg$  4 apples. That means that she planted four of the six apples she bought, leaving only 6 apples - 4 apples =  $\ll 6 - 4 = 2 \gg$  2 apples to be eaten. ##### 2

USER Q: Boris has 100 apples. Beck has 23 fewer apples than Boris. If Boris gives Beck 10 apples, how many fewer apples does Beck have than Boris now?

A: Boris starts with 100 apples and gives Beck 10 apples, leaving him with 100 apples - 10 apples =  $\ll 100 - 10 = 90 \gg$  90 apples. Initially, Beck has 23 fewer apples than Boris, giving him a starting count of  $100 - 23 = \ll 100 - 23 = 77 \gg$  77 apples. After receiving 10 apples from Boris, Beck's total comes to 77 apples + 10 apples =  $\ll 77 + 10 = 87 \gg$  87 apples. Consequently, Beck now has  $90 - 87 = \ll 90 - 87 = 3 \gg$  3 fewer apples than Boris. ##### 3

Here a few-shot prompt is adopted. The LLM learns from these demonstrations of problem-solution pairs not only the way of problem-solving but also the way of formatting the output. For example, the final result of calculation follows the ##### token, and  $\ll \dots \gg$  annotates the detailed calculation steps (called **calculation annotation**)<sup>4</sup>.

<sup>4</sup>During prediction, a calculator is used when we see  $\ll \dots \gg$ . More specifically, once the LLM encounters “=” in a  $\ll \dots \gg$ , then the calculator calculates the expression on the left-hand side of “=”. This method helps reduce the calculation errors made by LLMs.

## 9.2 Advanced Prompting Methods

So far in this chapter, we have introduced the basic concepts related to LLM prompting and presented a number of prompts for NLP tasks. We now consider several techniques for enhancing the effectiveness of prompting.

### 9.2.1 Chain of Thought

We have encountered the concept of **chain of thought (CoT)** several times in this chapter and previous ones [Wei et al., 2022c; Chowdhery et al., 2022]. CoT methods provide a simple way to prompt LLMs to generate step-by-step reasoning for complex problems, thereby approaching tasks in a more human-like manner. Rather than coming to a conclusion directly, the CoT methods instruct LLMs to generate reasoning steps or to learn from demonstrations of detailed reasoning processes provided in the prompts. To illustrate CoT, we consider the problem of algebraic calculation, as commonly described in the literature. Suppose we are given an algebraic problem

*Calculate the average of the numbers 2, 4, and 6.*

We can consider it as the question and prompt an LLM to answer it.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: The answer is 6.

It seems difficult for the LLM to directly give a correct answer. A simple improvement is to add demonstrations of similar problems in the prompt, and thus the LLM can learn from these demonstrations.

Q: Please calculate the average of the numbers 1, 3, 5, and 7.

A: The answer is 4.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: The answer is 7.

The problem here is that, although we have shown a similar question-answer pair, it remains difficult for the LLM to reason out the correct answer. In CoT, not only can LLMs learn from the correspondence between questions and answers but they may gain more from detailed problem-solving steps that are used to derive the answers. To do this, we can incorporate some reasoning steps into the prompt to obtain a CoT prompt.

Q: Please calculate the mean square of the numbers 1, 3, 5, and 7.

A: Calculate the square of each number:  $1^2 = 1$ ,  $3^2 = 9$ ,  $5^2 = 25$ , and  $7^2 = 49$ . Sum the squares,  $1 + 9 + 25 + 49 = 84$ . There are 4 numbers in total. Divide the sum by the number of items,  $84/4 = 21$ . The answer is 21.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Calculate  $2 + 4 + 9$ , which equals 15. There are three numbers. Divide the total sum by the count, resulting in  $15/3 = 5$ . The answer is 5.

Here we highlight the reasoning steps in green. By providing a detailed reasoning process for a similar problem, the LLM learns to reason and thereby generates a problem-solving path that leads to the correct answer.

There are several benefits of using CoT prompting. First, CoT allows LLMs to decompose complex problems into smaller, sequential reasoning steps. This somewhat mirrors human problem-solving behaviors, making it particularly effective for tasks requiring detailed, multi-step reasoning. Second, CoT makes the reasoning process more transparent and interpretable. Since all reasoning steps are visible, we can understand and interpret how a conclusion was reached. Third, if users can follow the logic behind the reasoning process, they will be more likely to trust the predictions of an LLM. This is particularly important when applying LLMs in fields like medicine, education, and finance. Fourth, CoT is an in-context learning approach, and thus, it is applicable to most well-trained, off-the-shelf LLMs. Moreover, CoT provides efficient ways to adapt LLMs to different types of problems. It can even inspire more creative solutions by exploring various alternative reasoning paths, which might not be obvious when arriving at a conclusion directly.

The method described above requires providing one or more examples of CoT reasoning, typically called the few-shot CoT method. By contrast, the zero-shot CoT method does not require such examples. It instead prompts LLMs to reason step-by-step by incorporating specific instructions in prompts. For example, below is a zero-shot CoT prompt.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Let's think step-by-step.

We have three numbers: 2, 4, and 9. Add these numbers together,  $2 + 4 + 9 = 15$ . Determine how many numbers there are, which in this case is three. The average is calculated by dividing the total sum by the number of elements. Completing the division gives  $15/3 = 5$ . So the answer is 5.

Following the instruction “Let’s think step by step”, the LLM is prompted to generate detailed reasoning steps. As discussed in [Kojima et al. \[2022\]](#)’s work, prompting with such instructions may result in LLMs generating only the reasoning steps without a clear conclusion. In this case, a second round of prompting can be used to extract the answer from these reasoning

steps. For example, [Kojima et al. \[2022\]](#) create a second prompt which combines both the input and output in the first round of prompting. Using this combined input, the LLM can continue its reasoning process and then generate the correct answer. Furthermore, it is possible to prompt LLMs to reason using instructions other than “Let’s think step by step”, such as “Let’s think logically” and “Please show me your thinking steps first”.

While we have illustrated CoT methods using an algebraic reasoning problem, these methods can be applied to a variety of different problems. Typical problem-solving scenarios for CoT include mathematical reasoning, logical reasoning, commonsense reasoning, symbolic reasoning, code generation, and so on. See [Figure 9.1](#) for more examples of applying CoT in various tasks.

CoT today is one of the most active fields of prompt engineering. This has not only led to improved performance for LLM prompting but has opened the door to a wide range of methods for studying and verifying reasoning capabilities of LLMs. Although we have focused on the basic idea of CoT in this section, it can be improved in several ways. For example, we can consider the reasoning process as a problem of searching through many possible paths, each of which may consist of multiple intermediate states (i.e., reasoning steps). In general, we wish the search space to be well-defined and sufficiently large, so that we are more likely to find the optimal result. For this reason, an area of current LLM research is aimed at designing better structures for representing reasoning processes, allowing LLMs to tackle more complex reasoning challenges. These structures include tree-based structures [[Yao et al., 2024](#)], graph-based structures [[Besta et al., 2024](#)], and so on. By using these compact representations of reasoning paths, LLMs can explore a wider range of decision-making paths, analogous to System 2 thinking<sup>5</sup>. Another line of research focuses on prompting LLMs with multi-round interactions. This involves decomposing complex problems into sub-problems, verifying and refining model outputs, employing model ensembling, and so on. Note that these methods and the issues involved are not limited to CoT. In fact, they are often used as more general approaches to improving LLMs, while CoT can be seen as a way to test the capabilities of LLMs. We will see discussions of some of these issues in the following subsections.

Before leaving our discussion of CoT, we should consider its practical limitations. One of them is the need for detailed, multi-step reasoning demonstrations in few-shot CoT scenarios, which may be difficult to obtain, either automatically or manually. Also, there is no standard method for breaking down complex problems into simpler problem-solving steps. This often heavily depends on the user’s experience. In addition, errors in intermediate steps can also affect the accuracy of the final conclusion. For further discussion on the pros and cons of CoT, the interested reader can refer to recent surveys on this topic [[Chu et al., 2023](#); [Yu et al., 2023b](#); [Zhang et al., 2023a](#)].

---

<sup>5</sup>System 1 and System 2 thinking, as described by [Kahneman \[2011\]](#), represent two different modes of cognitive processing. System 1 is fast, automatic, intuitive, and emotional. This mode of thinking operates effortlessly and quickly, and is often what guides our daily decisions, judgments, and impressions. System 2 is slow, deliberate, and analytical. It is activated when we need to perform complex computations.

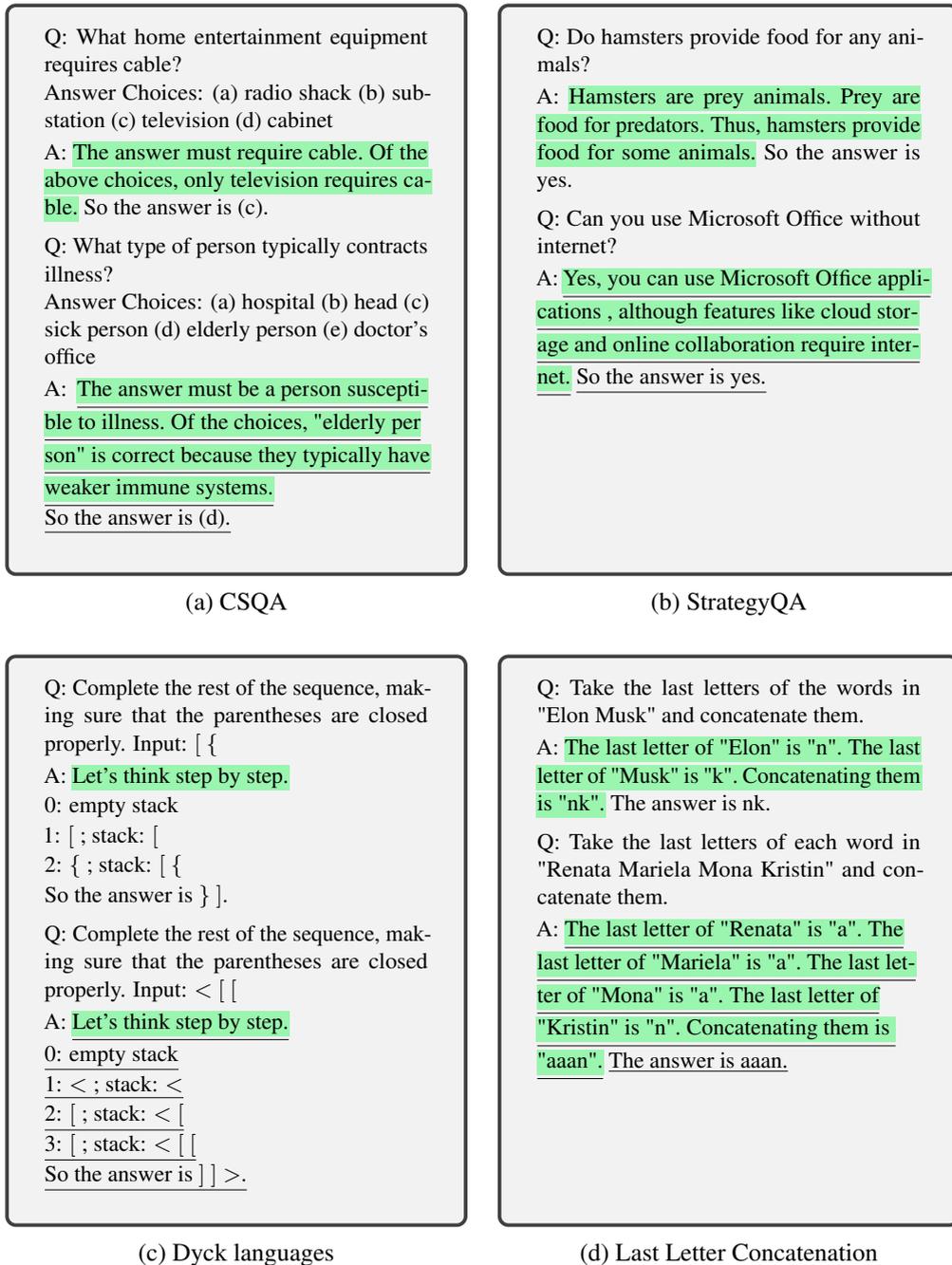


Figure 9.1: CoT in four different reasoning tasks, including CSQA, StrategyQA, Dyck languages, and Last Letter Concatenation. The CoT parts are highlighted in green.

## 9.2.2 Problem Decomposition

We have seen that LLMs can benefit from solving a complex problem by breaking it down into simpler problem-solving tasks. Such an approach can be seen as an example of a broader paradigm known as **problem decomposition**, which has been extensively explored and dis-

cussed in psychology and computer science. From the psychological perspective, complex problem-solving refers to a process of addressing a problem using knowledge that helps overcome the barriers of the problem<sup>6</sup>. There are generally no standard or clear paths to a solution for a complex problem. However, it is often advantageous to employ strategies that decompose the problem, thereby making it easier to tackle the corresponding sub-problems with less effort. For example, consider writing a blog about the risks of AI. If we simply prompt an LLM with the instruction “Please write a blog about the risks of AI”, the LLM may generate a blog with arbitrary structures and writing styles. A better method, instead, could be to outline the blog and provide more detailed information about each section. Consider the following prompt

You are a blog writer. Please follow the provided outline below to write a blog about the risks of AI.

- Introduction  
Introduce AI, its relevance, and the importance of understanding its risks for youth.
- Privacy Concerns  
Discuss how AI might compromise personal privacy through interactions online.
- Misinformation  
Explore AI’s role in spreading misinformation and influencing young people’s decisions.
- Cyberbullying  
Highlight how AI tools can be utilized in cyberbullying and the impact on mental health.
- Tips for Safe AI Use  
Offer guidelines for responsible AI usage and promote critical thinking.
- Conclusion  
Recap main points and encourage proactive engagement with AI ethics.

Here we give the title and major points for each section. Then, the LLM can use this structure to break down the writing task by filling in content for these sections. Note that the way to structure the blog can be provided by humans or even generated automatically. For example, we can use the LLM to first generate the outline, and then ask it to follow this outline to complete the writing.

In computer science, decomposing complex problems is a commonly used strategy in software and hardware system design. A well-known example is the divide-and-conquer paradigm, which is often used to design algorithms for computation problems that can be reduced to simpler, more manageable problems. For example, consider a problem of determining whether

---

<sup>6</sup>A relatively formal definition can be found in [Frensch and Funke \[2014\]](#)’s book: *complex problem-solving occurs to overcome barriers between a given state and a desired goal state by means of behavioral and/or cognitive, multi-step activities.*

a document discusses the risks of AI. We can instruct the LLM with the following prompt.

```
You are provided with a text. Please determine whether it discusses the risks  
of AI.
```

```
{*document*}
```

```
_____
```

If the document is long, the computation will be expensive. Alternatively, we can divide the document into relatively short segments and perform the same task on each segment. These segments can be processed in parallel to further reduce the computational cost. Next, we determine the relevancy of each segment to the topic of AI risks. The final output is then generated using another prompt.

```
Your task is to determine whether a text discusses the risks of AI. This text  
has been divided into segments, and you have obtained the relevancy of each  
segment to the topic of AI risks. Based on this, please provide your final  
result.
```

```
Segment 1: {*relevancy-to-the-topic1*}
```

```
Segment 2: {*relevancy-to-the-topic2*}
```

```
Segment 3: {*relevancy-to-the-topic3*}
```

```
...
```

```
_____
```

Now let us return to a more general discussion of problem decomposition in prompting. While problem decomposition can be applied to various NLP problems, it has been more extensively discussed and tested in reasoning tasks recently. For complex reasoning tasks, we often need a multi-step reasoning path to reach a correct conclusion. We can use LLMs to achieve this in three different ways. First, LLMs can directly reach the conclusion. In other words, they can predict without explicit reasoning processes, and there is a hidden and uninterpretable reasoning mechanism. Second, LLMs are prompted to generate a multi-step reasoning path that leads to the conclusion, like CoT. However, we run LLMs just once, and all intermediate steps in reasoning are generated in a single prediction. Third, we break down the original problem into a number of sub-problems, which are either addressed in separate runs of LLMs or tackled using other systems. Here we focus our attention on the third approach, which is closely related to problem decomposition. Note, however, that a more comprehensive discussion could cover all these approaches, while the first two have been discussed to some extent in this chapter.

A general framework for problem decomposition involves two elements.

- **Sub-problem Generation.** This involves decomposing the input problem into a number of sub-problems.
- **Sub-problem Solving.** This involves solving each sub-problem and deriving intermediate and final conclusions through reasoning.

These two issues can be modeled in different ways, leading to various problem decomposition methods. One approach is to treat them as separate steps in a two-step process. For example, consider the blog writing task described at the beginning of this subsection. In the first step, we decompose the entire problem into sub-problems all at once (i.e., outline the blog). In the second step, we solve the sub-problems either sequentially or in another order (i.e., fill in content for each section as needed). The final output of this process combines the results from solving each sub-problem. While this method is simple and straightforward, it assumes that the problem is compositional, making it more suitable for tasks like writing and code generation.

However, many real-world problems require complex reasoning. One key characteristic of these problems is that the reasoning steps may not be fixed. The reasoning path can vary for different problems, and each step of reasoning may depend on the outcomes of prior steps. In such cases, it is undesirable to use fixed sub-problem generation in advance. Instead, sub-problems should be generated dynamically based on the input problem, and, if possible, generated on the fly during the reasoning process. This makes problem decomposition more challenging compared with designing divide-and-conquer algorithms. Ideally, we would like to jointly design both the systems for sub-problem generation and sub-problem solving. But a more practical and widely used approach is to adopt separate models for these tasks. A straightforward way to achieve this is to adapt an LLM for these tasks by either prompting or tuning the model.

Here we consider a method based on the above idea, called **least-to-most prompting** [Zhou et al., 2023b]. The motivation for this method arises from the challenges of solving difficult reasoning problems — those that cannot be addressed by simply generalizing from a few examples. For these problems, a more effective problem-solving strategy is to follow a progressive sequence of sub-problems that systematically lead to the conclusion. More specifically, in the least-to-most prompting method, sub-problem generation is performed by prompting an LLM with instructions and/or demonstrations. For example, below is a 2-shot prompt for sub-problem generation in least-to-most prompting.

**TASK** Your task is to decompose a problem into several sub-problems. You will be given a few examples to illustrate how to achieve this.

**DEMO** Q: In a community, 5% of the population are infants, 15% are children, 40% are adults, and 40% are seniors. Which group makes up the largest portion of the population?

A: To answer the question “Which group makes up the largest portion of the population?”, we need to know: “How many percent are infants?”, “How many percent are children?”, “How many percent are adults?”, “How many percent are seniors?”.

Q: Alice, Bob, and Charlie brought beads for their group project in their craft class. Alice has twice as many beads as Bob, and Bob has five times as many beads as Charlie. If Charlie has 6 beads, how many beads can they use for their craft project?

A: To answer the question “How many beads can they use for their craft project?”, we need to know: “How many beads does Bob have?”, “How many beads does Alice have?”.

**USER** Q: The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius. What was the duration of the environmental study?

A: To answer the question “What was the duration of the environmental study?”, we need to know: “When did the environmental study start?”, “When did the environmental study end?”.

By learning from the examples, the LLM can generate two sub-problems for answering the new problem “What was the duration of the environmental study?” (highlighted in blue and orange). Given these sub-problems, we solve them sequentially. For each sub-problem, we take all previously-generated QA pairs as context, and then produce the answer. For the example above, we need to answer the first sub-problem by prompting the LLM, like this

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

**SUB-PROB1** Q: When did the environmental study start?

A: The environmental study started in 2015.

Once we have the answer to the first sub-problem, we proceed to the second one. This time, we include both the first sub-problem and its corresponding answer in the input.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

Finally, we use the LLM to solve the original problem given the answers to all the sub-problems.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

FINAL Q: What was the duration of the environmental study?

A: The duration of the environmental study was 5 years.

The least-to-most method offers a basic approach to prompting LLMs to generate and solve sub-problems separately. We can improve it in several ways. One simple improvement is to apply various advanced prompting techniques, which do not require changes to the problem decomposition framework. For example, we can incorporate CoT into the prompting to enhance the reasoning performance of sub-problem generation and solving.

Another improvement is to explore methods for better decomposing problems and organizing problem-solving paths. To describe these approaches, we will use the symbol  $p_0$  to denote the input problem, and use the symbols  $\{p_1, \dots, p_n\}$  to denote the sub-problems corresponding to  $p_0$ . For least-to-most prompting, we decompose  $p_0$  into  $\{p_1, \dots, p_n\}$ , given by

$$\{p_1, \dots, p_n\} = G(p_0) \quad (9.2)$$

where  $G(\cdot)$  denotes the function of sub-problem generation. Then, we solve the sub-problems  $\{p_1, \dots, p_n\}$  sequentially, resulting in a sequence of answers  $\{a_1, \dots, a_n\}$ . For answering the  $i$ -th sub-problem  $p_i$ , we include both the original problem  $p_0$  and all previously-seen problem-

answer pairs in the context for prediction. The answer  $a_i$  is given by

$$a_i = S_i(p_i, \{p_0, p_{<i}, a_{<i}\}) \quad (9.3)$$

where  $p_{<i} = \{p_1, \dots, p_{i-1}\}$  and  $a_{<i} = \{a_1, \dots, a_{i-1}\}$ .  $S_i(\cdot)$  denotes the function that solves the sub-problem  $p_i$  given the context  $\{p_0, p_{<i}, a_{<i}\}$ . The last step is to generate the answer to the original problem  $p_0$ , which can be expressed in a similar manner to Eq. (9.3).

$$a_0 = S_0(p_0, \{p_{\leq n}, a_{\leq n}\}) \quad (9.4)$$

One way to refine this model is to modify the  $G(\cdot)$  function so that the model can dynamically generate answers. Instead of generating all sub-problems at one time, we can generate each of them during problem-solving [Dua et al., 2022]. To do this, we can replace Eq. (9.2) with

$$p_i = G_i(p_0, \{p_{<i}, a_{<i}\}) \quad (9.5)$$

Hence we obtain a sub-problem generation model that operates in a step-by-step manner. At each step  $i$ , we first generate the sub-problem  $p_i$  by prompting an LLM with the original problem  $p_0$  and the problem-solving history  $\{p_{<i}, a_{<i}\}$ . We then generate the answer  $a_i$  for this sub-problem using the same or a different LLM, based on the same contextual information (see Eq. (9.3)). This method effectively expands the reasoning capacity of LLMs by allowing them to dynamically generate and solve sub-problems in intermediate reasoning steps. As a result, the reasoning paths are not fixed in advance, and the models can choose and adapt their reasoning strategies during problem-solving.

Another way to improve the above model is to focus on developing better sub-problem solvers. In our previous discussion, we restricted  $S_i(\cdot)$  to LLMs that are prompted to solve the sub-problem  $p_i$ . In fact, we can expand this function to any system that is capable of addressing the sub-problem. For example,  $S_i(\cdot)$  could make calls to IR systems, thereby allowing us to access a broader range of data for problem-solving. Another example is using  $S_i(\cdot)$  as a calculator to accurately compute results in mathematical problem-solving. If the sub-problem  $p_i$  is complex and requires multiple intermediate problem-solving steps, it is also possible to further decompose  $p_i$  into smaller sub-problems. For example,  $S_i(\cdot)$  can be defined as a recursive program that generates and solves sub-problems. This incorporates recursion into problem-solving and allows us to address problems by iteratively decomposing them. As a result, we can define a hierarchical structure for problem-solving [Khot et al., 2023].

If we generalize the above formulation a bit further, we can consider it as a reinforcement learning problem. A typical method is to model a problem-solving process as a decision making process. In each step of this process, an action is taken based on the current state. These actions can include all functions for sub-problem generation and solving (i.e.,  $G_i(\cdot)$  and  $S_i(\cdot)$ ). Thus, the action sequence corresponds to a problem-solving path. Since the discussion of reinforcement learning problems is beyond the scope of this chapter, we skip the precise description of this learning task. Nevertheless, developing an agent or controller to determine

when and how to generate and solve a sub-problem is also a natural choice.

In NLP, problem decomposition is related to a long line of research on multi-hop question answering [Mavi et al., 2024]. This task requires the system to gather and combine information from multiple pieces of text to provide an accurate answer to a complex question. For example, to answer the question “What is the capital of the country where Albert Einstein was born?”, we need to know “Where Albert Einstein was born?” and “What’s the capital of Germany?”. Earlier work in this area and related ones has investigated the issue of problem decomposition, though the methods might not be based on LLMs. For example, a popular method is to develop an additional neural model to generate simpler questions that address different aspects of the original question [Andreas et al., 2016; Talmor and Berant, 2018; Min et al., 2019]. This question generator can create questions in a batch or sequential manner.

Broadly speaking, problem decomposition is also related to the compositionality issue in NLP [Drozdov et al., 2022; Press et al., 2023]. For example, in semantic parsing, we map natural language sentences into structured meaning representations by breaking them down into constituent parts and understanding the sentences based on the meanings of these parts and the rules used to combine them. In early studies of this field, highly compositional sentences were considered easier for testing systems, as it is relatively straightforward to decompose such sentences and compose the meanings of their parts. However, the task becomes much more difficult when more generalization is required for modeling compositionality in new data. In this case, we want systems to have improved abilities of **compositional generalization**. In more recent research on LLMs, this issue has been frequently discussed in compositional reasoning tasks, such as SCAN<sup>7</sup>, as it is considered an important aspect of testing the language understanding and reasoning abilities of LLMs. This also presents new tasks for developing and examining problem decomposition methods.

In LLMs, one interesting application of problem decomposition is tool use. In some cases, it is necessary to integrate external tools into LLMs to access accurate data not available during training or fine-tuning. For example, LLMs can integrate with APIs to fetch real-time data such as weather updates, stock market prices, or news feeds, enabling them to provide up-to-date responses to user queries. When using tools, LLM predictions might include markers that indicate where and how to call external APIs. This requires decomposing the problem into sub-problems, with some handled by the LLMs and others by external tools. More detailed discussions on this topic will be presented in Section 9.2.5.

### 9.2.3 Self-refinement

In many cases, predictions of LLMs can be inaccurate or incorrect. Given that current LLMs can perform tasks like refinement and correction, it makes sense to explore methods for these models to self-refine their outputs. Self-refinement is a common phenomenon in human psychological activities and daily behavior. For example, when designing a product, a designer

---

<sup>7</sup>The SCAN tasks (Simplified versions of the ComMAI Navigation tasks) are designed to evaluate the ability of LLMs to perform compositional generalization [Lake and Baroni, 2018]. They involve translating natural language commands into a sequence of actions. For example, a command “jump opposite left and walk thrice” can be translated into the action sequence “LTURN LTURN JUMP WALK WALK WALK”.

might first create a basic prototype, then refine the design after evaluation and testing to enhance user experience and functionality. The refinement can be iterated several times until the design is satisfactory. The idea of predict-then-refine can also be found in NLP. One early example is Brill's tagger [Brill, 1992], where an initial POS tagging result of a sentence can be iteratively refined using a rule-based system. In the era of deep learning, a good deal of work on sequence-to-sequence problems, such as grammar correction and text rewriting, can also be seen as examples on this theme.

We can prompt LLMs to do self-refinement. Consider a simple example where we first use an LLM to translate a text from Chinese to English and then use the same model to refine the translation. Here is how we can prompt the model to translate:

Translate the following text from Chinese to English:

Chinese:

一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今5800年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。

English:

A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggest that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Then, we can prompt it to refine the translation:

You are provided with a Chinese text along with its English translation. Please refine the translation to improve its accuracy and fluency if necessary.

Chinese:

一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今5800年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。

English:

A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggests that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Refinement:

A series of **archaeological** discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings **suggest** that, starting from around 5800 years ago, **various** regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Here modifications to the input translation are highlighted in green. This process can be iterative: at each step, we use the translation from the previous step as input, and prompt the LLM to produce a potentially better translation.

In the above example, the refinement of the initial model output was achieved by prompting the LLM with a simple instruction like “Please refine it!”. However, the refinement is based solely on the instruction-following ability of the LLM, and there is no guidance or supervision on how and where to improve the model output. A more effective approach would be to consider feedback on specific aspects that need refinement. For example, we can prompt the LLM with “Please correct all the grammatical errors in the translation”, so that the model can focus more on grammatical error correction during refinement.

A general framework of self-refinement with LLMs involves three steps [Madaan et al., 2024].

- **Prediction.** We use an LLM to produce the initial model output.
- **Feedback Collection.** We obtain feedback on the model output.
- **Refinement.** We use the LLM to refine the model output based on the feedback.

The last two steps can be repeated multiple times, which leads to an iterative self-refinement process. In this framework, receiving accurate and detailed feedback is critical as it guides the subsequent refinement step. In general, feedback can be obtained in various ways [Fernandes et al., 2023; Pan et al., 2024]. We can manually examine the model’s predictions to identify

issues. Also, we can use a feedback model to automatically generate feedback. For instance, a reward model trained on some labeled data can help evaluate the model output and return a score indicating its quality.

Since the focus of this chapter is on LLM prompting, we consider using LLMs to provide feedback. To illustrate, let us consider an example of generating good responses to user questions:

- First, we prompt an LLM to answer the input question, as usual.

Generate a response to the question: “What are some major environmental concerns today?”

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

- Then, using the following prompt, we prompt the same LLM again to get feedback on the generated response.

You will be provided with a question and its corresponding response. Evaluate the accuracy and depth of this response, and identify any factual inaccuracies or areas where additional details could enhance understanding.

Question:

What are some major environmental concerns today?

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

Feedback:

The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:

1. Include specific examples of environmental issues.
2. Add information about major pollutants.
3. Specify the regions most affected.

- Given the feedback the LLM produces, we prompt it to refine the response.

You will be provided with a question, its corresponding response, and some feedback on this response.

Question:

What are some major environmental concerns today?

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

Feedback:

The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:

1. Include specific examples of environmental issues.
2. Add information about major pollutants.
3. Specify the regions most affected.

Now refine the response based on the provided feedback.

Refined Response:

Key environmental concerns include climate change, evidenced by rising global temperatures and severe weather patterns; significant air and water pollution in urban and industrial areas; rampant deforestation in regions like the Amazon, causing biodiversity loss; and ocean degradation, highlighted by coral reef bleaching and widespread overfishing.

Ideally, if a strong LLM is adopted, we would like to have it perform all three steps without extra training. On the other hand, if we have enough labeled data for the task of interest, we can enhance the performance of the LLM using supervised learning. For example, we can fine-tune the LLM to better adapt it to refinement tasks, or alternatively, use task-specific models, which may not necessarily be based on LLMs [Welleck et al., 2023; Schick et al., 2023]. In a broader sense, improving LLMs for self-refinement tasks can be seen as an alignment issue. For example, it has been found that some self-correction abilities can be activated through RLHF [Ganguli et al., 2023]. However, discussing these issues is beyond the scope of this chapter. Further discussion can be found in Chapter 10.

In LLMs, self-refinement is related to several concepts that reveal the psychological aspects of these models, such as the ability to self-reflect. A view is that if LLMs are capable of self-reflection, their predictions can become more accurate and even possess self-correcting capabilities. This self-reflection can be activated in various ways, for example, by prompting these LLMs to engage in more in-depth and careful thinking, or by providing examples from

which the models can learn and reflect. To illustrate, we consider here the **deliberate-then-generate (DTG)** method presented in Li et al. [2023a]’s work, where LLMs are prompted to deliberate. In DTG, we are given an initial model output which may contain errors. LLMs are then prompted to identify the error types of this model output and provide an improved output. Below is a template of DTG prompting for Chinese-to-English translation tasks.

Given the Chinese sentence: {*\*source\**}  
 The English translation is: {*\*target\**}  
 Please first detect the type of error, and then refine the translation.  
 Error Type:

We aim to first predict the error type (red), and then produce a refined translation (blue). This process of deliberation is guided by the instruction “Please first detect the type of error, and then refine the translation”. It encourages LLMs to initially engage in thoughtful analysis and then give better results. Since error type prediction and refinement are performed in a single run of LLMs, this method incorporates both steps of feedback and refinement into one process.

In the above prompts, we assume that the LLM we use is able to review the input translation and correctly identify its error types. However, this raises new difficulties as the model may not be good at finding errors in translations. This will in turn result in extra fine-tuning or prompting engineering efforts. So a simpler method is to reduce the burden of error identification and use LLMs for deliberation only. To do this, we can replace the input translation with a random translation and assign a default error type. An example of such a prompt is shown below.

Given the Chinese sentence:  
 一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。  
 The English translation is:  
A variety of innovative techniques have redefined the importance of modern art in contemporary cultural studies.  
 Please first detect the type of error, and then refine the translation.  
 Error Type:

In this example, the input translation is not generated by LLMs but is instead randomly sampled from the dataset. So it is simply an incorrect translation for the source sentence, and we can set the error type accordingly. The LLMs then generate a new translation by taking both the source sentence and the incorrect translation as input. The design of this prompt

can also be considered as activating the learning capabilities of LLMs through “negative evidence” [Marcus, 1993], thereby enabling them to reflect and produce better outcomes through contrastive analysis. Nevertheless, this method does not rely on any feedback and can enhance the performance of a single LLM prediction via simple prompting.

Note that while DTG is non-iterative, iterative learning and refinement are commonly used in NLP. An advantage of these iterative approaches is that they mimic human learning and problem-solving, where continuous feedback and adjustments lead to progressively improved outcomes. Iterative methods can be applied to a range of LLM prompting problems. For example, in problem decomposition, one can incorporate new sub-problems and their solutions into the context at each step, and thus LLMs can progressively approach the solution of the original problem. On the other hand, iterative methods raise several issues that are absent in non-iterative methods, for example, errors in earlier steps may negatively impact subsequent problem-solving, and determining when to stop iterating often requires additional engineering effort.

### 9.2.4 Ensembling

Model ensembling for text generation has been extensively discussed in the NLP literature. The idea is to combine the predictions of two or more models to generate a better prediction. This technique can be directly applicable to LLMs. For example, we can collect a set of LLMs and run each of them on the same input. The final output is a combined prediction from these models.

For LLM prompting, it is also possible to improve performance by combining predictions based on different prompts. Suppose we have an LLM and a collection of prompts that address the same task. We can run this LLM with each of the prompts and then combine the predictions. For example, below are three different prompt templates for text simplification.

Make this text simpler.

{\*text\*}

\_\_\_\_\_

Condense and simplify this text.

{\*text\*}

\_\_\_\_\_

Rewrite for easy reading.

{\*text\*}

\_\_\_\_\_

Each of these prompts will lead to a different prediction, and we can consider all three predictions to generate the final one.

Formally, let  $\{\mathbf{x}_1, \dots, \mathbf{x}_K\}$  be  $K$  prompts for performing the same task. Given an LLM  $\Pr(\cdot|\cdot)$ , we can find the best prediction for each  $\mathbf{x}_i$  using  $\hat{y}_i = \arg \max_{y_i} \Pr(y_i|\mathbf{x}_i)$ . These predictions can be combined to form a “new” prediction:

$$\hat{y} = \text{Combine}(\hat{y}_1, \dots, \hat{y}_K) \quad (9.6)$$

Here  $\text{Combine}(\cdot)$  is the combination model, which can be designed in several different ways. For example, we can select the best prediction by voting or by identifying the one that overlaps the most with others. Another method for model combination is to perform model averaging during token prediction. Let  $\hat{y}_j$  be the predicted token at the  $j$ -th step for model combination. The probability of predicting  $\hat{y}_j$  is given by

$$\hat{y}_j = \arg \max_{y_j} \sum_{k=1}^K \log \Pr(y_j|\mathbf{x}_k, \hat{y}_1, \dots, \hat{y}_{j-1}) \quad (9.7)$$

The interested reader can refer to Chapter 5 for more details of these methods.

In ensembling for LLM prompting, it is generally advantageous to use diverse prompts so that the combination can capture a broader range of potential responses. This practice is common in ensemble learning, as diversity helps average out biases and errors that may be specific to any single model or configuration. From the Bayesian viewpoint, we can treat the prompt  $\mathbf{x}$  as a latent variable, given the problem of interest,  $p$ . This allows the predictive distribution of  $\mathbf{y}$  given  $p$  to be written as the distribution  $\Pr(\mathbf{y}|\mathbf{x})$  marginalized over all possible prompts

$$\Pr(\mathbf{y}|p) = \int \Pr(\mathbf{y}|\mathbf{x}) \Pr(\mathbf{x}|p) d\mathbf{x} \quad (9.8)$$

The integral computes the total probability of  $\mathbf{y}$  by considering all possible values of  $\mathbf{x}$ , weighted by their likelihoods given  $p$ . Here  $\Pr(\mathbf{y}|\mathbf{x})$  is given by the LLM, and  $\Pr(\mathbf{x}|p)$  is the prior distribution of prompts for the problem. This is a good model because the integral effectively accounts for the uncertainty in the choice of  $\mathbf{x}$ , ensuring that the final predictive distribution  $\Pr(\mathbf{y}|p)$  is robust and encompasses all potential variations and biases in the prompts. However, computing this integral directly can be computationally infeasible due to the potentially infinite space of  $\mathbf{x}$ . One approach to addressing this issue is to employ methods like Monte Carlo sampling, which approximate the integral using a manageable, finite number of prompts.

While the Bayesian treatment is mathematically well-defined, it is common practice in NLP to assume a non-informative or uniform prior and focus instead on constructing a set of diverse prompts. Consequently, the output can be computed using a straightforward combination model, as described in Eq. (9.6). The issue of creating high-quality, diverse prompts has been studied in CoT and other in-context learning areas. Most of the research focuses on

incorporating a variety of demonstration examples across different prompts. Here, we list some of these methods.

- Given a problem, we manually create a number of demonstrations and use different ones for different prompts.
- Given a problem, we use LLMs to automatically generate demonstrations and prompts.
- Given a prompt, we create different prompts by changing the order of demonstrations in the prompt.
- Given a prompt, we use LLMs to generate a number of similar prompts.
- Given a prompt, we transform it into other forms, e.g., translating it into other languages.

Of course, in practice, we can combine these methods to achieve greater diversity. An underlying assumption here is that diverse prompts can lead to diverse model outputs. This is particularly the case when the problem we deal with is relatively new and difficult. For stronger and more robust LLMs, the variance in output for similar prompts might not be large. In this case, the benefit of involving multiple prompts can be modest.

In addition to providing diverse prompts for LLMs, another approach is to make use of the inherent variance in the outputs of LLMs. One simple way to generate multiple outputs is to sample outputs from the hypothesis space. This is straightforward for LLMs because their search algorithms are naturally suited to producing multiple results. For example, if beam search is used, we can easily collect all the complete hypotheses in the beam to generate multiple output candidates. Also, we can modify the search algorithms to implement more powerful sampling methods and explore a wider range of high-quality hypotheses.

The next question is how to combine diverse model predictions. This leads us back to the model combination issue as described in Eq. (9.6). Here we consider the **self-consistency** method, which outputs not the prediction with the highest probability, but rather the one that best aligns with other predictions [Wang et al., 2022d; 2023c]. First, an LLM is prompted with CoT as usual and generates multiple reasoning paths by sampling. Then, the frequency of each answer is counted across these reasoning paths. The final output is the answer that has the highest count. Below shows three different predictions by prompting an LLM with CoT.

Prediction 1 (Correct):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Each coin flip is an independent event with a 50% chance of landing heads (H) and 50% tails (T). Consider the possible outcomes for three flips: HHH, HHT, HTH, HTT, THH, THT, TTH, TTT. Out of these, only HHH, HHT, HTH, and THH are relevant. We are only interested in scenarios with exactly one head: HTT, THT, and TTH. There are 3 such outcomes out of 8 total, giving a probability of 3/8.

Prediction 2 (Correct):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Using a binomial probability formula, where  $p$  (success or heads) is 0.5, and each of the 3 trials (flips) is independent, we look for exactly one success (head). The formula for exactly  $k$  successes in  $n$  trials is given by the binomial coefficient formula  $\binom{n}{k}p^k(1-p)^{n-k}$ . Plugging in the values, we get  $\binom{3}{1}(0.5)^1(0.5)^{3-1} = 3 \cdot 0.5 \cdot 0.25 = 0.375$ .

Prediction 3 (Wrong):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Since each coin flip has a 50% chance of landing heads and a 50% chance of landing tails, and we are looking for the case where only one of the three flips results in heads, we might initially think this is simply a matter of picking which flip is the heads. There are three flips, so one might incorrectly assume that the chance of picking one specific outcome like this would be 1 out of 3. Thus, they might conclude that the probability of exactly one head is  $1/3 = 33.3\%$ .

Predictions 1 and 2 correctly identify the three cases where exactly one head is flipped, both obtaining a probability of 37.5%. The reasoning in Prediction 3 fails to account for the total number of outcomes possible with three coin flips, thus giving a wrong answer of 33.3%. Therefore, we select 37.5% as the final answer because it is the consensus.

Self-consistency provides a criterion for determining the best prediction in a pool of candidates. Since the prompt and the model are fixed in this method, it is not strictly a prompt ensembling method. Instead, it can be seen as an instance of output ensembling methods, also known as hypothesis selection methods, which have long been explored in NLP, particularly for text generation problems [Xiao et al., 2013]. In these methods, multiple outputs are generated by varying model architectures or parameters. Each output is then assigned a score by some criterion, and the outputs are re-ranked based on these scores. There are various ways to define the scoring function, such as measuring the agreement between an output and others, and using a stronger model to rescore each output<sup>8</sup>. Figure 9.2 shows a comparison of different

<sup>8</sup>An interpretation of self-consistency is to view it as a minimum Bayes risk search process. It searches for the best output by minimizing the Bayes risk. More specifically, a risk function  $R(y, y_r)$  is defined on each pair of outputs (denoted by  $(y, y_r)$ ), representing the cost of replacing  $y$  with  $y_r$ . Given a set of outputs  $\Omega$ , the risk of an

ensembling methods for LLMs.

Now, let us briefly review the methods we have discussed so far in this section, such as problem decomposition and self-refinement. It is apparent that these methods enhance decision-making by introducing more “choices” into the reasoning process. To some extent, they all involve evaluating and providing feedback on the results of LLMs. For example, in self-refinement, we need to offer suggestions for improving the prediction of LLMs, and in output ensembling, we select the optimal output from a pool of candidates. In this sense, these methods fall under the broader category of predict-then-verify approaches, where predictions are initially made, then verified and refined. The fundamental problem here involves verifying and evaluating the reasoning results or intermediate steps. This issue is somewhat related to the problem of training reward models in RLHF, although RLHF addresses a different aspect. In fact, the development of verifiers has been explored and implemented in reasoning with LLMs. Most work, rather than developing heuristic-based inference-time algorithms, focuses on learning verifiers in a supervised manner. A straightforward method is to train verifiers as binary classifiers, such as classifying an answer as correct or incorrect, although these verifiers are typically used as scoring models. Given a reasoning path for a problem, the verifiers can be used to score either the entire path (called outcome-based approaches) [Cobbe et al., 2021], or each individual reasoning step (called process-based approaches) [Uesato et al., 2022; Lightman et al., 2024].

### 9.2.5 RAG and Tool Use

RAG is generally employed when standard LLMs, which rely solely on pre-trained knowledge, lack accuracy and depth in the generated text. By drawing from external databases and documents, RAG can significantly improve the quality of responses, ensuring they are both contextually relevant and factually correct. Such an approach is particularly useful in scenarios that require high factual accuracy and up-to-date information, such as complex question answering.

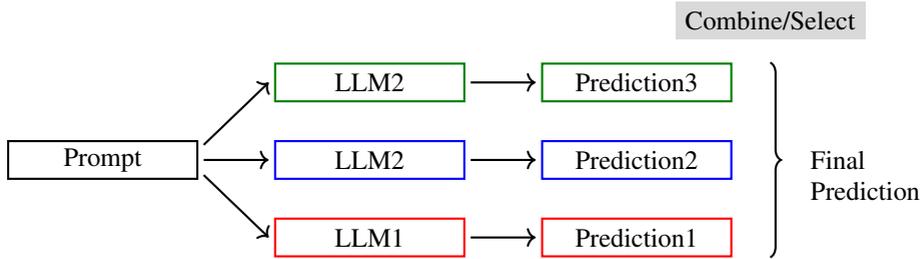
The concept of RAG has been mentioned several times in the previous sections and chapters. For completeness, we outline the key steps involved in RAG here.

- We prepare a collection of texts which are treated as an additional source of knowledge we can access.
- We retrieve relevant texts for a given query.
- We input both the retrieved texts and the query into an LLM, which is then prompted to produce the final prediction.

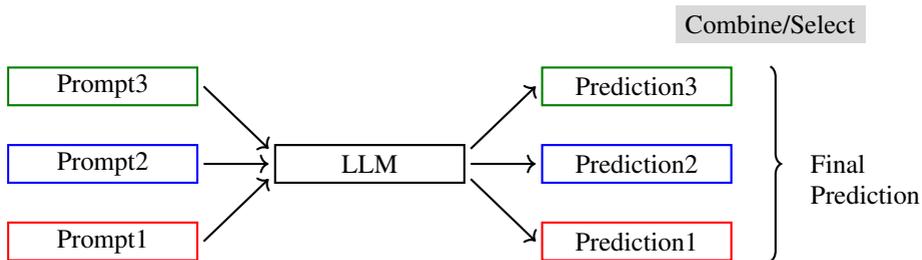
Steps 1 and 2 can be implemented by using an external information retrieval system. For example, we can store the collection of texts in a vector database and then retrieve the most similar texts through vector-based search techniques. Since information retrieval is not the

output  $\mathbf{y} \in \Omega$  is given by

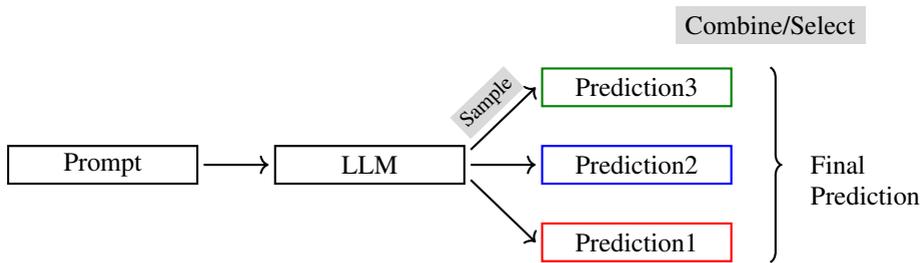
$$\begin{aligned} \text{Risk}(\mathbf{y}) &= \mathbb{E}_{\mathbf{y}_r \sim \Pr(\mathbf{y}_r|\mathbf{x})} R(\mathbf{y}, \mathbf{y}_r) \\ &= \sum_{\mathbf{y}_r \in \Omega} R(\mathbf{y}, \mathbf{y}_r) \cdot \Pr(\mathbf{y}_r|\mathbf{x}) \end{aligned} \quad (9.9)$$



(a) Model Ensembling



(b) Prompt Ensembling



(c) Output Ensembling

Figure 9.2: Ensembling methods for LLMs. In standard model ensembling (a), multiple LLMs varying in architectures or parameters are used. Each LLM receives the same prompt and produces a prediction. These predictions are combined to generate the final prediction. In prompt ensembling (b), we have one LLM and multiple prompts. The LLM produces a prediction for each prompt, and these predictions are combined as usual. In output ensembling (c), the LLM samples multiple predictions over the prediction space given a prompt. It can be seen as a method to boost the performance of the LLM itself. Note that these ensembling methods can be combined to increase the diversity of predictions. For example, we can use both prompt ensembling and output ensembling to obtain more diverse predictions.

focus of this chapter, we will assume that such systems are available off-the-shelf and use them directly.

Here we present how to prompt LLMs to make use of retrieved texts. To illustrate, consider

an example of using LLMs to answer the following question.

*Where will the 2028 Olympics be held?*

We can simply input this question into an online search engine. It will then return the relevant pieces of text found on the internet, for example,

(Wikipedia)

*The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28, is an upcoming international multi-sport event scheduled to take place from July 14-30, 2028, in the United States. ...*

(The Sporting News)

*In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after hosting the Summer Games in 1932 and 1984. It will also be the first time the United States has hosted an Olympic Games since the 2002 Winter Games in Salt Lake City. ...*

...

We can use these retrieved texts as additional context, and prompt an LLM to generate a response based on these texts. Below is an example RAG prompt.

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.

Question:

Where will the 2028 Olympics be held?

Relevant Text 1:

The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28 ...

Relevant Text 2:

In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after ...

...

The 2028 Olympics will be held in Los Angeles.

This prompt assumes that the provided texts are relevant to the question and expects the LLM to generate a faithful response using these texts. However, the information retrieval system may sometimes provide irrelevant or incorrect texts, which may lead the LLM to produce an incorrect answer. One straightforward way to address this issue is to improve the accuracy of the information retrieval system. Nevertheless, as with most AI systems, errors may still occur. Therefore, it is also necessary to enhance the robustness of the LLM, so that it

can make reasonable predictions even when the input is inaccurate. Below is a new prompt that enables the LLM to be more faithful to the facts, and allows it to choose not to answer questions when the information provided is inaccurate.

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.

Please note that your answers need to be as accurate as possible and faithful to the facts. If the information provided is insufficient for an accurate response, you may simply output "No answer!".

Question:

Where will the 2028 Olympics be held?

Relevant Text 1:

The 2024 Summer Olympics, officially the Games of the XXXIII Olympiad and branded as Paris 2024, were an international multi-sport event ...

...

No answer!

In this example, the LLM refuses to answer because the provided information is insufficient and irrelevant to the question.

Both RAG and fine-tuning are common methods for adapting LLMs using task-specific data. Standard RAG is training-free and can be directly applied to LLMs. To further improve RAG, it is also possible to fine-tune LLMs, though this will require some training effort. For example, we can fine-tune LLMs using human-labelled data to supervise them in learning to refuse to answer. Note that, while the examples shown above seem simple, RAG is not trivial. From the prompt engineering perspective, different use cases may require different prompts, though our somewhat “greedy” goal is to develop a universal prompting strategy that can adapt to different tasks. In many cases, we need to control how much we depend on the retrieved context to make predictions. Sometimes, LLMs must derive responses strictly from the provided texts, while at other times, they may need to generate responses using their pre-trained knowledge if the provided texts are insufficient. There are many aspects of RAG, such as improvements to the retrieval systems, that cannot be covered in this chapter. Interested readers can refer to surveys of RAG techniques for more information [Li et al., 2022d; Gao et al., 2023c].

One reason we discuss RAG here is that it can be broadly regarded as an instance of the general problem decomposition framework (see Section 9.2.2). RAG divides problem-solving into two steps. In the first step, we collect relevant and supporting information for a given query from various knowledge sources. In the second step, we use LLMs to generate responses based on the collected information. If we extend the concept of problem decomposition further, we will find that many tasks requiring the use of external systems or tools can be treated as

similar problems. One such example is tool use in LLMs. In many applications, LLMs need to employ external databases, APIs, and even simulation tools to generate accurate responses. For example, LLMs can access real-time data from financial markets to provide up-to-date investment advice or integrate with healthcare databases to offer personalized medical insights. This integration extends the capabilities of LLMs by allowing them to interact with, and in some contexts, influence or control external systems. Consequently, LLMs function more as **autonomous agents** rather than mere text generators [Franklin and Graesser, 1996].

The issue of tool use is broad and vast. Here we narrow our discussion to tasks that can be facilitated by calling external APIs to solve some of the sub-problems [Parisi et al., 2022; Gao et al., 2023b]. Consider again the example of asking an LLM to answer “Where will the 2028 Olympics be held?”. Suppose the LLM can access a web search tool. We can then prompt the LLM to answer the question with web search, like this

Your task is to answer the following question. You may use external tools, such as web search, to assist you.

Question:

Where will the 2028 Olympics be held?

The information regarding this question is given as follows:

```
{tool: web-search, query: "2028 Olympics"}
```

So the answer is: Los Angeles

Here `{tool: web-search, query: "2028 Olympics"}` indicates a request to the web search system using the query “2028 Olympics”. When the LLM sees this string, it executes a web search and uses the result to replace the string. Then, in subsequent steps of prediction, the LLM uses this web search result as context to produce the correct answer.

Consider another example where we ask the LLM to solve a mathematical problem.

**Problem:**

A swimming pool needs to be filled with water. The pool measures 10 meters in length, 4 meters in width, and 2 meters in depth. Calculate the volume of the pool in cubic meters and then determine how many liters of water are needed to fill it (considering 1 cubic meter equals 1000 liters).

**Solution:**

To solve this problem, the LLM needs to first calculate the volume of the pool by using the formula for the volume of a rectangular prism:  $\text{Length} \times \text{Width} \times \text{Depth}$ . Therefore, The volume is  $10\text{ m} \times 4\text{ m} \times 2\text{ m} = \{\text{tool: calculator, expression: } 10 * 4 * 2\} \text{ m}^3$ . Next, to find out how many liters of water are needed, the LLM multiplies the volume in cubic meters by 1000 (since 1 cubic meter equals 1000 liters). Thus,  $80 \times 1000 = \{\text{tool: calculator, expression: } 80 * 1000\}$  liters.

Here the string `{tool: calculator, expression: 10 * 4 * 2}` triggers the invocation of a mathematical interpreter to calculate the result of the expression. Note that the result (i.e., 80) will replace `{tool: calculator, expression: 10 * 4 * 2}` and can be referred to in the following token predictions. For example, in the last step of problem-solving, 80 is used instead of `{tool: calculator, expression: 10 * 4 * 2}`.

A key difference between the tool use examples here and the previously discussed RAG examples is that in tool use, external functions can be called during inference. In contrast, in RAG, the retrieved texts are provided before the prediction process begins. However, from the language modeling perspective, they are actually doing the same thing: before generating the final result, we use external tools, either manually or automatically, to obtain sufficient and relevant context. A high-level interpretation of these approaches is that they both rely on an “agent” that can determine where and how to call external functions to generate the context necessary for prediction.

An issue with tool use is that the original LLMs are not trained to generate the necessary markers for tool use. Therefore, we need to fine-tune the LLMs to adapt them for these tasks [Schick et al., 2024]. As this chapter focuses on prompting, we will not present the details of this fine-tuning process. To put it simply, we first need to annotate data. For each fine-tuning example, we replace parts of the output that require the use of external tools with predefined commands or markers. Then, we use this labeled data to fine-tune the parameters of the LLM as usual. As a result, the LLM can gain the ability to generate commands for calling external tools. During inference, we can execute these tool use commands in the model outputs to get assistance from external tools.

## 9.3 Learning to Prompt

So far in this chapter, we have considered several basic prompting strategies and various refinements to them. However, all the prompts we have discussed were designed manually. This leads to a number of problems: First, designing high-quality prompts is inherently difficult and requires substantial manual effort. For example, extensive experimentation with different prompts is often needed to identify the most effective ones. Since different LLMs may respond better to certain types of prompts, developing universally effective prompts can be even more resource-intensive. Second, manual prompt design relies heavily on human expertise, which can limit the diversity of approaches and overlook potentially effective prompts that are not immediately obvious to humans. Third, prompts created by humans can be complex and redundant, leading to longer inputs for LLMs and higher computational costs.

In this section, we discuss techniques for automated prompting. These methods aim to automatically create, optimize, and represent prompts so that the downstream tasks can be addressed more effectively and efficiently. In particular, we consider three issues here.

- How can we automate the process of designing and optimizing prompts for LLMs?
- Are there other forms of representing prompts beyond strings, and how can we learn such representations?
- How can we make prompts more concise and compact, thereby reducing their complexity and length?

Note that there are many settings in which we can investigate these issues. For example, we might specify that prompts are developed specifically for a particular LLM, or that the development is independent of the LLM used. These settings can lead to different methods and application scenarios, but these methods may overlap in some ways. In the following discussion, we will cover several different scenarios and discuss the connections between various methods.

### 9.3.1 Prompt Optimization

Given that prompt design is difficult and labor-intensive, it is desirable to use machine learning models to discover the optimal prompt for a specific task (call it **automatic prompt design** or **prompt optimization**). This approach can broadly be regarded as an instance of **automated machine learning (AutoML)**, which aims to reduce or eliminate the need for expert-driven manual design of machine learning models. Although our focus here is on the design of prompts, prompts themselves are discrete structures. Therefore, designing prompts is very similar to designing machine learning models, such as discrete model architectures. Perhaps one of the most related fields is **neural architecture search (NAS)**, where the most optimal neural networks are identified by exploring a space of possible neural networks [Zoph and Le, 2016; Elsken et al., 2019a]. If we consider prompt optimization as a search process, then we can describe a general prompt optimization framework involving the following components:

- **Prompt Search Space.** This defines all possible prompts that the algorithms can explore. For example, one can edit some seed prompts to generate a set of diverse candidate

prompts.

- **Performance Estimation.** Once a prompt is chosen, it needs to be evaluated. For example, a straightforward way is to input it to an LLM and measure its performance on a validation set.
- **Search Strategy.** The search process is generally the same as that used in many AI systems. At each step, the system explores a set of promising prompts in the search space and evaluates them. This process continues as more prompts are explored. The outcome of the search is the best-performing prompt observed until the search stops.

This is a very general framework, and different prompt optimization systems can vary in their design of each component. A widely-used approach is to use LLMs as the basis to develop these components. Initially, a few prompts are provided. Then, the following process is iterated until a stopping criterion is met: 1) the prompts are evaluated on a validation set; 2) a candidate pool is maintained by keeping only the most promising prompts; and 3) new prompts are created by employing LLMs to infer similar prompts from this candidate pool. One benefit of this approach is that it allows us to use off-the-shelf LLMs to perform the tasks mentioned above without the need for substantial system development. To achieve this, we can prompt or fine-tune LLMs to adapt them to these tasks. Here we consider [Zhou et al. \[2023c\]](#)'s method for illustrating LLM-based prompt optimization. It involves the following steps.

- **Initialization.** Let  $C$  represent the pool of the candidate prompts we intend to explore. The first step is to add initial prompts into  $C$ . We can do this in several ways. A simple method is to create such prompts by hand for a given task. However, in many cases where humans have limited knowledge about how to write effective prompts for the task, developing prompts becomes challenging. In these cases, it is desirable to use LLMs to generate prompts. For example, we can directly instruct LLMs to produce prompts, providing them with a description of the task.

You are given a task to complete using LLMs. Please write a prompt to guide the LLMs.

{\*task-description\*}

\_\_\_\_\_

This method is straightforward, but it still requires a human-provided description of the task. An alternative method is to use LLMs to generate prompts given examples of the input and output of the task. Here is a prompt template.

You are provided with several input-output pairs for a task. Please write an instruction for performing this task.

Input: {*\*input1\**} Output: {*\*output1\**}

Input: {*\*input2\**} Output: {*\*output2\**}

...

\_\_\_\_\_

As such, LLMs can infer the corresponding instruction for the task from the provided inputs and outputs.

- **Evaluation.** Once we obtain the candidate pool  $C$ , we need to evaluate the prompts in  $C$ . One method is to feed each prompt into an LLM and assess the results on the downstream task. For example, we can evaluate the output of the LLM given an input using a pre-defined metric, or alternatively, use the log-likelihood of the output as a measure of the quality of the prompt.
- **Pruning.** If  $C$  contains a large number of prompts, it is reasonable to prune the unpromising prompts within it, thus reducing the computational burden in subsequent steps. This is a standard pruning problem. Given the evaluation score for each prompt, a simple method is to keep only a certain percentage of the prompts and discard the rest.
- **Expansion.** Expansion is a key operation in search algorithms used to explore different states in the search space. The expansion operation here can be defined as a function

$$C' = \text{Expand}(C, f) \quad (9.10)$$

where  $C'$  is the set of new prompts generated from  $C$  using the model  $f$ . If we consider  $f$  as an LLM, we can perform the expansion operation by instructing  $f$  to generate new and relevant prompts based on  $C$ . Below is an example.

Below is a prompt for an LLM. Please provide some new prompts to perform the same task.

Input: {*\*prompt\**}

\_\_\_\_\_

Then, we replace  $C$  with  $C'$ . The steps of evaluation, pruning and expansion can be repeated, and so we can gradually explore a wider range of prompts.

In prompt optimization, the expansion step plays a key role, as it defines how we explore the search space, and our goal is to find optimal results with minimal effort. One improvement to this step is to treat the problem as a paraphrasing task. A simple method is to apply off-the-shelf paraphrasing systems, either based on LLMs or other models, to transform input prompts

into semantically equivalent forms [Jiang et al., 2020]. Alternatively, we can define specific edit operations, such as insertions and modifications, for each token. A given prompt can be edited into new prompts by applying these operations [Prasad et al., 2023]. Also, further evaluation and pruning can be applied to filter out low-quality prompts. In addition to framing prompt generation as a paraphrasing problem, we can improve the quality of prompts during expansion by learning from feedback [Pryzant et al., 2023]. This approach is somewhat related to the self-refinement issue discussed in Section 9.2.3. An LLM can be used to generate feedback on an input prompt, which is then revised based on this feedback. This feedback-and-revision cycle can be repeated multiple times until the result converges or the desired outcome is achieved.

Another approach to prompt optimization is to apply classic optimization techniques. For example, the problem can be framed as an evolutionary computation problem, where prompts are treated as candidates that evolve generation by generation as the optimization progresses [Guo et al., 2024]. Since many powerful optimization algorithms have been developed in related fields, they can be directly applied to this problem.

In practice, we might be tempted to use existing LLM APIs to implement the steps described above. Such an approach, however, would be strongly dependent on the inference and in-context learning abilities of the LLMs. If these LLMs are not strong and lack adaptation to the tasks, they may introduce errors into search, for example, generating incorrect prompts during expansion. In such cases, it is preferable to train models that are better suited to the tasks. One approach in this research direction appeals to reinforcement learning, which has been widely used in solving discrete decision making and optimization problems. For example, Deng et al. [2022] developed a prompt generator by integrating an FFN-based adaptor into an LLM. The prompt generator is trained as a typical policy network, but only the parameters of the adaptor are updated while the remaining parameters of the model are kept unchanged. During training, the reward is obtained by testing the generated prompts using another LLM, similar to the evaluation method as discussed above. Once the training is complete, the prompt generator is then employed to generate new prompts.

Note that, in our discussion here, prompts are simply seen as sequences of tokens, and the output of prompt optimization is such a sequence. However, in a strict sense, prompts have complex structures and include different fields such as user input, instruction, and demonstration. While our discussed approaches are mostly general, much work in prompt optimization has focused on learning better instructions for prompting. Specifically, the goal is to generate instructions that effectively guide LLMs based on a given task. Of course, the concept of prompt optimization can also be extended to learning other parts of prompts. For example, there has been substantial research interest in learning to select or generate demonstrations in CoT [Liu et al., 2022; Rubin et al., 2022; Zhang et al., 2023b]. One of the differences between learning instructions and learning demonstrations is that generating high-quality demonstrations using LLMs is relatively easy and the focus of learning demonstrations is typically on how to sample appropriate demonstrations from a pool of candidates. In contrast, the difficulty in learning instructions is partly because pre-trained LLMs are not suited to predict the quality of instructions, and testing these instructions on downstream

tasks is computationally expensive. This makes the optimization methods costly to apply, and exploring a wide variety of instructions poses significant challenges.

### 9.3.2 Soft Prompts

Although developing natural language prompts, either manually or automatically, is a straightforward and widely applied approach, it presents some problems. One problem is that natural language prompts can be complex and lengthy, resulting in significant computational burdens when processed via LLMs. In many applications, users may need to perform a task repeatedly, and inputting the same long prompt into the LLMs a large number of times is clearly inefficient. Another problem is that while prompts are typically represented as discrete token sequences (call them **hard prompts**) in regular LLM input, the LLMs encode them as low-dimensional real-valued vectors. This raises the question of whether there are more compact and efficient ways to represent prompts.

In this subsection, we introduce the concept of **soft prompts**, which can be viewed as hidden, distributed representations of prompts. When prompting LLMs, we are concerned with communicating tasks or questions to elicit the desired responses. We can define hard prompts as explicit, predefined text sequences that users input directly into LLMs to guide the responses. In contrast, we can think of soft prompts as implicit, adaptable prompting patterns embedded within LLMs. Unlike hard prompts, which are expressed in natural language and should be understandable for humans, soft prompts are encoded in a format that is more comprehensible to the model rather than to humans. To illustrate, consider a simple prompt

Translate the sentence into Chinese.

Consider it done!

—

Here, the instruction “Translate the sentence into Chinese” can be seen as a hard prompt, denoted by the token sequence  $c_1 \dots c_5$ . By feeding these tokens into an LLM, they are transformed into a sequence of real-valued vectors  $\mathbf{h}_1 \dots \mathbf{h}_5$ , each corresponding to a token. We can roughly think of  $\mathbf{h}_1 \dots \mathbf{h}_5$  as a soft prompt, as illustrated in Figure 9.3.

While the above example shows that soft prompts can be generated by transforming hard prompts, there is not necessarily a direct correspondence between them. In fact, we do not even need to interpret soft prompts using meaningful text. They are instead simply hidden states in LLMs and can be learned as standard parameters of the models through continuous optimization. Such a treatment allows us to explore prompting methods beyond text. As another benefit, soft prompts provide dense, low-dimensional, and learnable representations for encoding how we guide LLMs to generate specific outputs. The training and application of these representations require significantly lower computational costs than those required for processing long hard prompts. This approach would be of great practical value in LLM inference applications where the same prompt is repeatedly used.

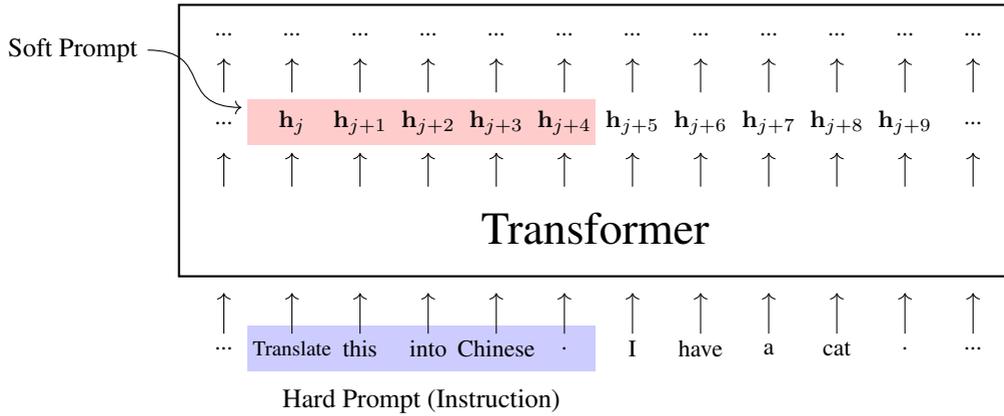


Figure 9.3: Illustration of hard and soft prompts. Here the hard prompt is the instruction we input to the LLM for performing the task. The LLM encodes this instruction as usual, and the intermediate representations corresponding to the instruction can be viewed as some sort of soft prompt.

## 1. Adapting LLMs with Less Prompting

One obvious way to adapt an LLM for a particular task is to simply fine-tune the model using labeled data. This leads to a variety of LLM alignment methods, such as supervised fine-tuning, which update the model parameters by aligning the responses to given prompts with supervision signals. Fine-tuned LLMs embed task-related information in model parameters, and thus these models can respond correctly when dealing with similar prompts with those in fine-tuning.

If we take this idea further, we can expect LLMs to absorb the knowledge about prompting of a task as much as possible during fine-tuning. Consequently, the prompting information is partially captured in the model parameters, and the fine-tuned LLMs can perform the task with less prompting. Here we consider a simple form of prompt, where only an instruction (denoted by  $\mathbf{c}$ ) and a user input (denoted by  $\mathbf{z}$ ) are included. A prompt can be expressed using the following tuple

$$\mathbf{x} = (\mathbf{c}, \mathbf{z}) \quad (9.11)$$

Given a set of prompt-response pairs  $\mathcal{D} = \{(\mathbf{x}, \mathbf{y})\}$ , the objective of fine-tuning is to minimize the total loss incurred over this set. A popular method is to minimize the negative log-likelihood (i.e., maximize the log-likelihood) with respect to the model parameters  $\theta$ :

$$\begin{aligned} \hat{\theta} &= \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\mathbf{y} | \mathbf{x}) \\ &= \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\mathbf{y} | \mathbf{c}, \mathbf{z}) \end{aligned} \quad (9.12)$$

where  $\Pr_{\theta}(\cdot|\cdot)$  is the probability predicted by an LLM with the parameters  $\theta$ <sup>9</sup>.

In general, the instruction in each fine-tuning example should follow the guideline of prompt design, for example, a good instruction should be as clear as possible and provide a detailed description of the task. However, the method described in the above equation does not restrict the instruction to any particular form. This flexibility allows us to instruct LLMs in any way that we want. Consider an example where we intend to instruct LLMs to translate an English sentence into Chinese. Of course, as mentioned earlier in this chapter, we can prompt LLMs using the instruction

*Translate the following sentence from English to Chinese.*

If we want the instruction to be simpler, we may rephrase it into a simpler form

*Translate this into Chinese.*

Even, we can define the instruction as a single phrase

*Translate!*

With certain fine-tuning effort, we can adapt LLMs to follow any of these instructions. From an efficient prompting perspective, there are computational advantages in simplifying instructions in prompting. For example, we can use simple instructions like “Translate!” to perform tasks that would typically require more complex and detailed instructions. This can make subsequent prompting during inference much easier. On the other hand, fine-tuning LLMs with overly simplified instructions may be harmful to the generalization of the models. Since simplified instructions can lead to a loss of information, it is more likely that the LLMs will overfit the fine-tuning data and fail to generalize beyond those instructions. In scenarios involving both complex and simplified instructions for fine-tuning, this problem is more severe because the labeled data available for fine-tuning is usually limited, and accommodating a variety of instructions is costly.

An alternative way to adapt LLMs for simplified instructions is through knowledge distillation. As an example, we consider the context distillation method [Snell et al., 2022]. The goal of this method is to learn a student model that can make use of simplified instructions from a well-trained instruction-following teacher model. Figure 9.4 shows an illustration of this approach. Building the teacher model follows a standard fine-tuning process: we first collect a certain amount of data that includes instructions, user inputs, and correct responses, and then we continue to train a pre-trained model with this dataset. For building the student model, we need to construct a new dataset  $\mathcal{D}'$  where each sample is a tuple consisting of an instruction, a corresponding simplified instruction, and a user input, denoted by  $\mathbf{x}' = (\mathbf{c}, \mathbf{c}', \mathbf{z})$ . Knowledge distillation is performed by minimizing a loss function defined on the outputs of the teacher

<sup>9</sup>In practice, we initialize  $\theta$  with the parameters obtained from pre-training, and then adjust  $\theta$  moderately to ensure that the results after fine-tuning do not deviate too much from the pre-trained results.

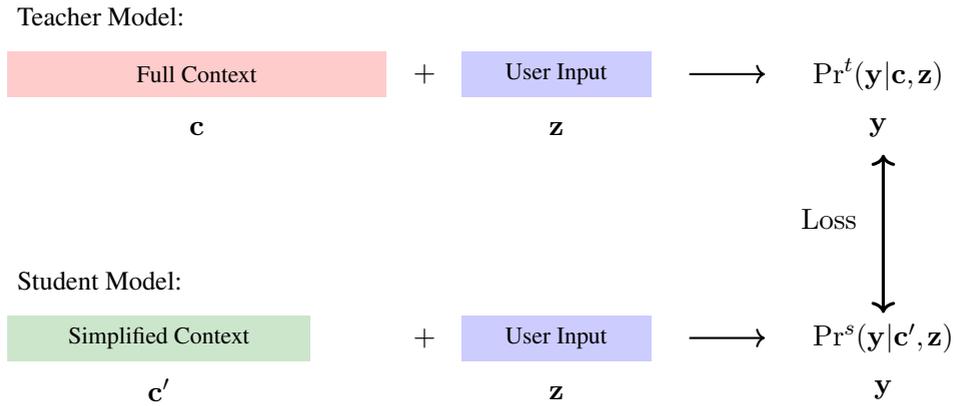


Figure 9.4: Illustration of context distillation [Snell et al., 2022]. The teacher model is a standard LLM, which takes both the context and the user input as model input and produces a prediction as model output. Then, we simplify the context (e.g., simplifying the instruction in prompting) and use the student model to make predictions based on the simplified context and the user input. The student model is trained by minimizing the loss between the predictions produced by the two models.

and student models

$$\hat{\theta} = \operatorname{argmin}_{\theta} \sum_{\mathbf{x}' \in \mathcal{D}'} \operatorname{Loss}(\Pr^t(\cdot|\cdot), \Pr_{\theta}^s(\cdot|\cdot), \mathbf{x}') \quad (9.13)$$

where  $\Pr^t(\cdot|\cdot)$  denotes the pre-trained teacher model, and  $\Pr_{\theta}^s(\cdot|\cdot)$  denotes the student model with the parameters  $\theta$ . To keep the notation simple we will write  $\operatorname{Loss}(\Pr^t(\cdot|\cdot), \Pr_{\theta}^s(\cdot|\cdot), \mathbf{x})$  as  $\operatorname{Loss}$  for short. A commonly-used loss is the sequence-level loss, which has the basic form:

$$\operatorname{Loss} = \sum_{\mathbf{y}} \Pr^t(\mathbf{y}|\mathbf{c}, \mathbf{z}) \log \Pr_{\theta}^s(\mathbf{y}|\mathbf{c}', \mathbf{z}) \quad (9.14)$$

But this function is computationally infeasible because it requires summing over an exponentially large number of outputs. A variant of this method is to train the student model using outputs generated by the teacher model. For each sample, we use the teacher model to produce an output  $\hat{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y}} \log \Pr^t(\mathbf{y}|\mathbf{c}, \mathbf{z})$ . Then we consider  $\hat{\mathbf{y}}$  as the target for learning, and the loss function is given by

$$\operatorname{Loss} = \log \Pr_{\theta}^s(\hat{\mathbf{y}}|\mathbf{c}', \mathbf{z}) \quad (9.15)$$

Alternatively, we can minimize the distances between the probability distributions outputted by the two models [Askell et al., 2021]. For example, the loss function can be defined as the KL divergence between the two output distributions

$$\operatorname{Loss} = \operatorname{KL}(\mathbf{P}^t \parallel \mathbf{P}_{\theta}^s) \quad (9.16)$$

where

$$P^t = \Pr^t(\cdot | \mathbf{c}, \mathbf{z}) \quad (9.17)$$

$$P_\theta^s = \Pr_\theta^s(\cdot | \mathbf{c}', \mathbf{z}) \quad (9.18)$$

Although we have restricted ourselves to knowledge distillation for instructions, the approaches discussed here are general. By learning from the outputs of the teacher model, the knowledge in prompting can be distilled into the parameters of the student model. Therefore, the distilled model can be considered as encoding some sort of soft prompt. This method can be applied to many other problems in prompt learning, such as compressing long contexts and learning soft prompts as specific components of LLMs.

## 2. Learning Soft Prompts for Parameter-efficient Fine-tuning

Updating all parameters is a common method for adapting LLMs to tasks of interest. Although fine-tuning is considered computationally cheaper than pre-training, it is still costly to apply in practice. This issue motivates the development of parameter-efficient fine-tuning methods, which aim to minimize the number of parameters that need to be updated.

One approach, known as **prefix fine-tuning**, is to append a series of trainable vectors, or prefixes, at the beginning of the input of each Transformer layer [Li and Liang, 2021]. These prefixes can be thought of as soft prompts that serve as additional context to guide the behavior of the model under specific tasks. During fine-tuning, we need only to learn the prefixes for embedding task-specific knowledge. Thus, this method is efficient because it only modifies a small part of the model rather than adjusting the entire set of model parameters.

Specifically, let the input of a layer at depth  $l$  be denoted by  $\mathbf{H}^l = \mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l$ . The output of the layer can be expressed as

$$\mathbf{H}^{l+1} = \text{Layer}(\mathbf{H}^l) \quad (9.19)$$

In prefix fine-tuning, we extend the sequence  $\mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l$  by adding a few vectors at the beginning, which we denote as  $\mathbf{p}_0^l \mathbf{p}_1^l \dots \mathbf{p}_n^l$ . Hence  $\mathbf{H}^l$  can be written in the form

$$\mathbf{H}^l = \underbrace{\mathbf{p}_0^l \mathbf{p}_1^l \dots \mathbf{p}_n^l}_{\text{trainable}} \underbrace{\mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l}_{\text{previous layer output}} \quad (9.20)$$

The output of the layer is the last  $m + 1$  representations.

$$\begin{aligned} \overline{\mathbf{H}}^{l+1} &= \text{Layer}(\mathbf{H}^l)[-m-1:] \\ &= \mathbf{h}_0^{l+1} \mathbf{h}_1^{l+1} \dots \mathbf{h}_m^{l+1} \end{aligned} \quad (9.21)$$

where  $[-m-1:]$  denotes the slicing operation that extracts the last  $m + 1$  elements of a sequence. Given  $\overline{\mathbf{H}}^{l+1}$ , the input of the next layer can be expressed in the same form of Eq.

(9.20):

$$\begin{aligned} \mathbf{H}^{l+1} &= \mathbf{p}_0^{l+1} \mathbf{p}_1^{l+1} \dots \mathbf{p}_n^{l+1} \overline{\mathbf{H}}^{l+1} \\ &= \mathbf{p}_0^{l+1} \mathbf{p}_1^{l+1} \dots \mathbf{p}_n^{l+1} \mathbf{h}_0^{l+1} \mathbf{h}_1^{l+1} \dots \mathbf{h}_m^{l+1} \end{aligned} \quad (9.22)$$

Here each  $\mathbf{p}_i \in \mathbb{R}^d$  can be seen as a learnable parameter. During training,  $\mathbf{p}_0^l \mathbf{p}_1^l \dots \mathbf{p}_n^l$  are trained as usual, and the parameters of the original Transformer model are kept fixed.

Figure 9.5 shows an illustration of prefix fine-tuning for a translation task. Here, only the prefix vectors  $\mathbf{p}_0^l$  and  $\mathbf{p}_1^l$  are updated by receiving the error gradients from the output (i.e., the Chinese translation). By adjusting these vectors for the translation task, the model adapts accordingly. This makes  $\mathbf{p}_0^l$  and  $\mathbf{p}_1^l$  serve as prompts which activate the LLM to perform the task without needing explicit input prompts like “Translate the following sentence from English to Chinese”. At test time, we prepend the optimized  $\mathbf{p}_0^l$  and  $\mathbf{p}_1^l$  to the layer, and the LLM will then translate the input sentence. Note that prefix fine-tuning introduces additional  $L \times n \times d$  parameters, where  $L$  is the number of layers,  $n$  is the number of prefixes, and  $d$  is the dimensionality of each prefix. However, this number is much smaller compared to the total number of parameters in the LLM, making the fine-tuning process highly efficient.

While prefix fine-tuning is simple, it still requires modifications to LLMs. Alternatively, separating soft prompts from the LLMs allows us to preserve the original model architecture, making it more efficient for deployment across different tasks without the need to adjust the core model. One such method is prompt tuning [Lester et al., 2021]. Like prefix fine-tuning, prompt tuning incorporates trainable vectors so that LLMs can adapt to given tasks by adjusting these vectors. However, prompt tuning differs in that it modifies only the embedding layer.

Recall that in LLMs each input token  $z_i$  is represented by an embedding  $\mathbf{e}_i$ . These embeddings are generally learned through a token embedding model and are then used as the real inputs to the LLMs, replacing the symbolically represented tokens. In prompt tuning, a number of pseudo embeddings  $\mathbf{p}_0 \dots \mathbf{p}_n$  are added at the beginning of the token embedding sequence. So the actual input to the LLMs can be expressed as

$$\underbrace{\mathbf{p}_0 \mathbf{p}_1 \dots \mathbf{p}_n}_{\text{trainable}} \quad \underbrace{\mathbf{e}_0 \mathbf{e}_1 \dots \mathbf{e}_m}_{\text{token embeddings}}$$

Note that a pseudo embedding needs not to correspond to any token in natural language. Instead these embeddings can be seen as “soft prompt embeddings” that serve to condition the LLMs. By training soft prompt embeddings on task-specific data, they learn to interact adaptively with the token embeddings  $\mathbf{e}_0 \dots \mathbf{e}_m$  and guide the behavior of LLMs. Since prompt tuning does not change the underlying parameters of pre-trained LLMs, it is considered a lightweight and efficient method of fine-tuning, improving task-specific performance while maintaining their generalization capabilities. See Figure 9.6 for an illustration of prompt tuning.

Since  $\mathbf{p}_0 \mathbf{p}_1 \dots \mathbf{p}_n$  is itself a sequence, we can employ sequence models to better represent it. For example, a Transformer model can encode this sequence, and the resulting representation can then be used as the input to the LLM. In other words, we can develop an additional model for encoding soft prompts. Another way to improve prompting is by combining soft and

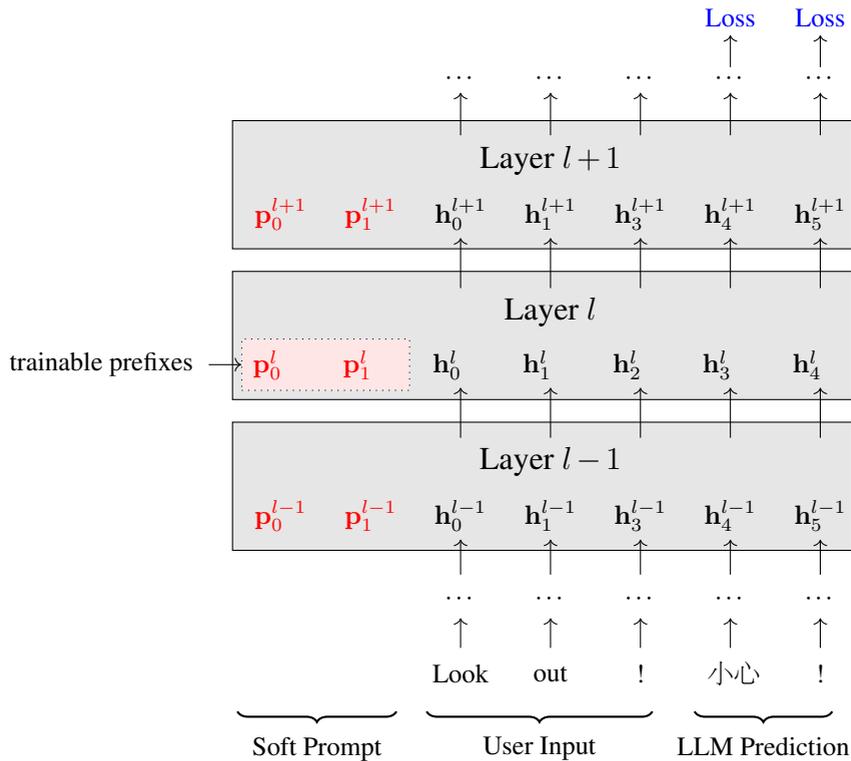
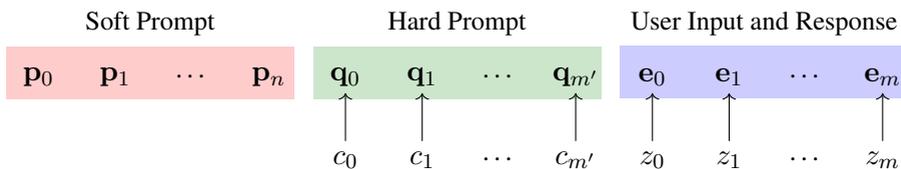


Figure 9.5: Illustration of prefix fine-tuning for a translation task (Look out!  $\rightarrow$  小心!). For each layer, we add two prefixes  $p_0^l$  and  $p_1^l$  at the beginning. The LLM is trained to minimize the loss on the predictions given the input. During this process, only the prefixes are optimized while the rest of the parameters remain fixed. Therefore, the model can adapt to the given task in a very efficient manner. At inference time, the LLM works with optimized prefixes, and can perform the task without the need of explicit hard prompts.

hard prompts, thereby taking advantage of both types [Liu et al., 2023c]. In the embedding sequence, we can arrange or intersperse these prompts. This would result in different prompt patterns. For example, a simple pattern that uses both two types of prompt is



where  $c_0 \dots c_{m'}$  denotes the hard prompt and  $q_0 \dots q_{m'}$  denotes the corresponding embedding sequence.

Here we have considered methods for inserting soft prompts in LLMs. But we skip the details of training these soft prompts and assume that the reader is familiar with the standard supervised learning process, that is, maximizing the likelihood of the correct model output

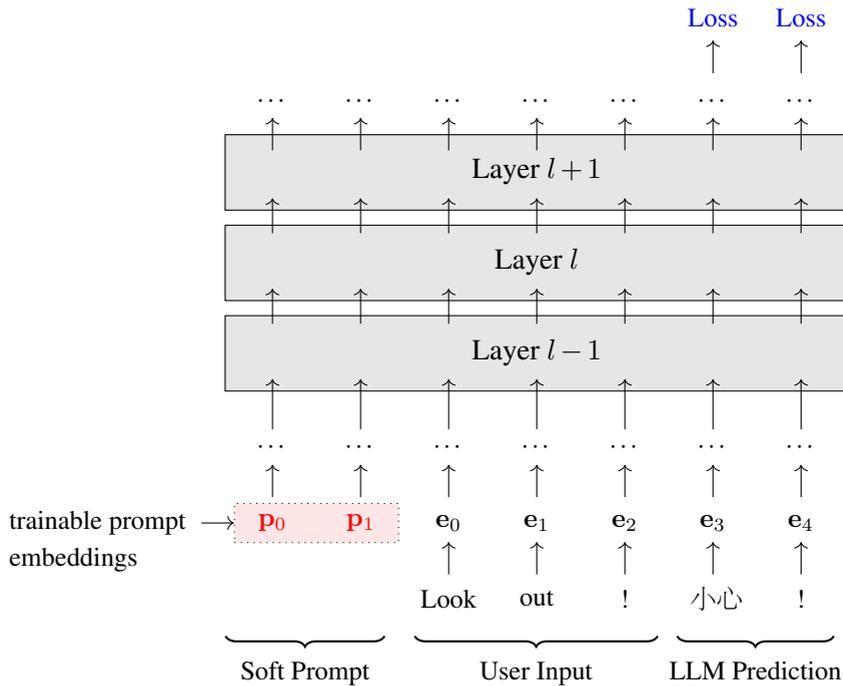


Figure 9.6: Illustration of prompt tuning for a translation task (Look out!  $\rightarrow$  小心!). Instead of using fixed textual prompts, soft prompts are learnable embeddings that are added at the beginning of the embedding sequence. During fine-tuning, only these prompt embeddings are optimized to efficiently adapt the LLM to the given task. Once optimized, the prompt embeddings are used to instruct the LLM to perform the task as new data arrives.

given the model input. In fact, learning soft prompts can be related to many issues in LLM fine-tuning. For example, if we consider it as a context compression problem, we can apply the knowledge distillation methods described previously. In [Mu et al. \[2024\]](#)'s work, prompts are compressed and represented as a few pseudo tokens, which are appended to each input sequence. The embeddings of these pseudo tokens are optimized to mimic the predictions of a standard-prompted model. In other words, the prompting knowledge is distilled from a teacher model into the pseudo tokens.

Broadly speaking, many parameter-efficient fine-tuning methods can be thought of as learning some sort of soft prompt [[Lialin et al., 2023](#)]. When we fine-tune a part of an LLM for a task, this process can essentially be seen as injecting task-related prompting information into that specific part of the model. Another widely-used approach to parameter-efficient fine-tuning is to add an adaptor layer between the existing model layers. This approach allows us to fine-tune only the adaptor layer on specific tasks without altering the underlying architecture or retraining the entire model. In this sense, adaptor layers can be viewed as soft prompts that encode prompting and task-related information and interact with the original LLM to help it adapt. To summarize, [Figure 9.7](#) shows a comparison of different methods of using soft prompts in LLMs.

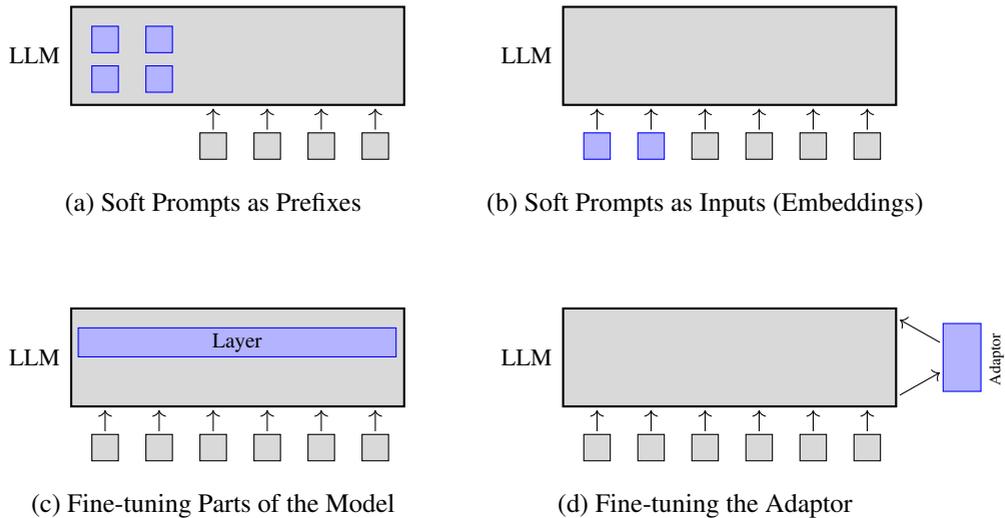


Figure 9.7: Illustrations of using soft prompts in LLMs. Here tunable soft prompts are shown in blue, and components whose parameters are fixed during fine-tuning are shown in gray. In sub-figure (a), soft prompts are prefixes appended to each layer of the LLM. In sub-figure (b), soft prompts are used as input embeddings for the LLM. In sub-figures (c) and (d), soft prompts are broadly treated as components of the model that are fine-tuned for task adaptation.

### 3. Learning Soft Prompts with Compression

Another approach to learning soft prompts is from the perspective of compression. As a simple example, consider the problem of approximating a long context using a continuous representation [Wingate et al., 2022]. Suppose we have a user input  $\mathbf{z}$  and its context  $\mathbf{c}$  (such as long instructions and demonstrations). Now we want to develop a compressed representation of the context, denoted by  $\sigma$ , such that the prediction based on  $\mathbf{z}$  and  $\sigma$  is as close as possible to the prediction based on  $\mathbf{z}$  and  $\mathbf{c}$ . This goal can be expressed in the form

$$\hat{\sigma} = \underset{\sigma}{\operatorname{argmin}} s(\hat{\mathbf{y}}, \hat{\mathbf{y}}_{\sigma}) \quad (9.23)$$

where  $\hat{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{c}, \mathbf{z})$  and  $\hat{\mathbf{y}}_{\sigma} = \operatorname{argmax}_{\mathbf{y}} \Pr(\mathbf{y}|\sigma, \mathbf{z})$  are the LLM predictions given the full context and the compressed context, respectively. The function  $s(\cdot, \cdot)$  typically represents a loss or similarity measure, aiming to minimize the difference in predictions between the two context representations.

One general framework for achieving this is knowledge distillation, where  $\hat{\mathbf{y}}$  and  $\hat{\mathbf{y}}_{\sigma}$  can be seen as the predictions of the teacher model and the student model, respectively. This formalization links our discussion to the context distillation problem discussed earlier. The training objective can be obtained by analogy with Eqs. (9.15) and (9.16). For example, a simple training objective is given by

$$\hat{\sigma} = \underset{\sigma}{\operatorname{argmax}} \log \Pr(\hat{\mathbf{y}}|\sigma, \mathbf{z}) \quad (9.24)$$

Alternatively, we can minimize the KL divergence between the output distributions, giving

$$\hat{\sigma} = \operatorname{argmin}_{\sigma} \operatorname{KL}(\operatorname{Pr}(\cdot|\mathbf{c}, \mathbf{z}) \parallel \operatorname{Pr}(\cdot|\sigma, \mathbf{z})) \quad (9.25)$$

The difference with the models in Eqs. (9.15) and (9.16) is that here the compressed context is represented as real-valued vectors (call them **prompt embeddings**), rather than as normal tokens. By applying the above methods, we distill the context from the token sequence  $\mathbf{c}$  into the embeddings  $\sigma$ . Note that the teacher model  $\operatorname{Pr}(\cdot|\mathbf{c}, \mathbf{z})$  and the student model  $\operatorname{Pr}(\cdot|\sigma, \mathbf{z})$  may not share the same architecture or model settings. In practice, we generally wish for the teacher model to be stronger, while the student model should be smaller and more efficient.

While compressing full context into continuous representations is a straightforward approach to learning soft prompts, it requires a teacher model that can deal with long input sequences. In many cases, however, the context is so long that applying an LLM is too costly or infeasible. Modeling long input sequences can fall under the broad family of efficient methods for long-context LLMs. Many techniques have been developed to address this issue. For example, one can use a fixed-size KV cache to store the past information at each step during inference. Efficient Transformer architectures and long-context LLMs have been intensively discussed in this book. For more detailed discussions of these topics, interested readers can refer to Chapters 6 and 8.

There are also methods specifically designed to compress long context into soft prompts. Here we consider [Chevalier et al. \[2023\]](#)'s method as an example. The basic idea is that we learn soft prompts gradually by accumulating the fixed-size context representation over the context sequence. Given a long context, we first divide it into a number of segments  $\mathbf{z}^1, \dots, \mathbf{z}^K$ . We then process these segments in sequence, each time generating a representation of the context we have processed so far, denoted by  $\sigma^{<i+1}$ . To do this, a few summary tokens  $\langle \mathbf{g}_1 \rangle, \dots, \langle \mathbf{g}_\kappa \rangle$  are introduced. At each step, we take a segment  $\mathbf{z}^i = z_1^i \dots z_{m_i}^i$ , along with the previous context representation  $\sigma^{<i}$  and the summary tokens  $\langle \mathbf{g}_1 \rangle, \dots, \langle \mathbf{g}_\kappa \rangle$  as input, and use an LLM to produce the corresponding hidden representation sequence at the last Transformer layer. An example of this process is illustrated in Figure 9.8.

Here  $\sigma^{<i}$  is essentially a memory. The model operates in an RNN fashion. Each time we take a segment and update this memory by encoding both the previous memory state and the segment. Therefore, the  $\sigma^{<i}$  produced at the last segment is a representation of the entire context sequence. The Transformer model for learning these representations can be a standard LLM but we need to fine-tune it to adapt to this context representation task.

Note that here we simply consider *prompt* and *context* as similar terms, even though they are not the same. Although we are somewhat “misusing” the concept *prompt*, we can often view it as a type of context. From this perspective, the methods discussed here can be applied to general text compression problems.

### 9.3.3 Prompt Length Reduction

While soft prompts provide dense, hidden representations, they are not directly interpretable. The lack of interpretability can be a significant barrier for users trying to understand how their

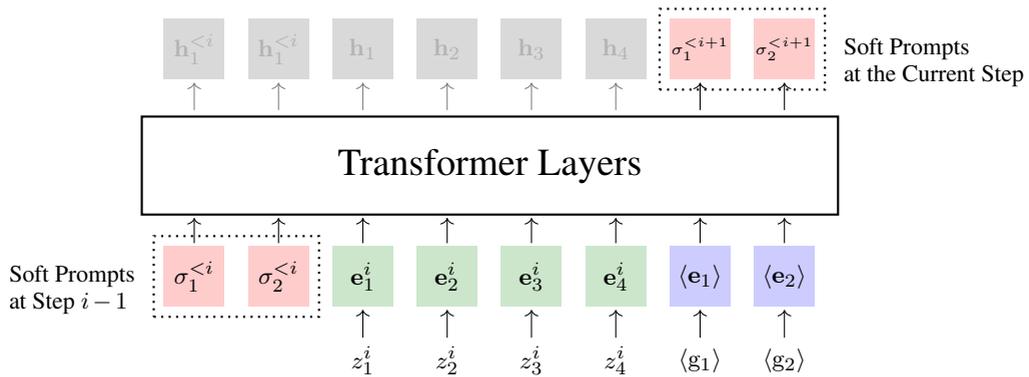


Figure 9.8: Illustration of compressing a context segment into soft prompts ( $\kappa = 2$  and  $m_i = 4$ ). The input to the LLM includes the soft prompts from the previous step ( $\sigma_1^{<i>i-1</i>}$  and  $\sigma_2^{<i>i-1</i>}$ ), the tokens of the segment ( $z_1, z_2, z_3$ , and  $z_4$ ), and the summary tokens ( $\langle g_1 \rangle$  and  $\langle g_2 \rangle$ ). Given these, the LLM operates as usual. We then extract the outputs at the last Transformer layer that correspond to the summary tokens. These outputs can be viewed as the soft prompts that accumulated up to this segment.

inputs influence LLM outputs. Moreover, although soft prompts are efficient for fine-tuning and deployment, they are inflexible and do not allow for easy adjustments without extensive fine-tuning or modification. This inflexibility can limit their utility in dynamic environments where prompt changes are frequently needed.

One alternative way to develop efficient prompts is to simplify the text used for prompting. For example, below is a prompt for answering questions on healthcare and finance.

The task involves developing a language model capable of understanding and responding to user inquiries across various domains, with a particular emphasis on healthcare and finance. Considering the broad range of potential queries, from the specifics of medical diagnoses to the nuances of financial regulations, the model must ensure a comprehensive understanding and accurate responses.

Question:

What are the best practices for using artificial intelligence in diagnosing cardiovascular diseases?

\_\_\_\_\_

We can simplify the task description by deleting the unimportant parts.

*The task involves developing a language model capable of understanding and responding to user inquiries across various domains, with a particular emphasis on healthcare and finance. Considering the broad range of potential queries, from the specifics of medical diagnoses to the nuances of financial regulations, The model must ensure a comprehensive understanding and accurate responses.*

We can also paraphrase it as a shorter text.

*The task involves developing a language model focused on healthcare and finance, capable of understanding and accurately responding to a wide range of user inquiries.*

This problem can be viewed as a classic NLP issue — text simplification. So the methods used can be general and not restricted to the problem of simplifying prompts. There are many ways to achieve this. One simple method is to define some heuristics and identify redundant words that can be eliminated without losing essential information. For example, we can examine each token in a sequence in terms of its contribution to the overall meaning and remove those that provide minimal value [Li et al., 2023c; Jiang et al., 2023b]. Another method involves framing the problem as a sequence-to-sequence task. With labeled data for text simplification, we can train an encoder-decoder model to transform each input text into its simplified form. In addition, given that many LLMs have been fine-tuned and aligned to perform text simplification tasks, it is straightforward to use these models to simplify prompts. For example, we can prompt an LLM to simplify a text under certain constraints, such as limiting the length of the simplified text.

## 9.4 Summary

In this chapter, we have discussed a variety of issues related to LLM prompting. Our discussion has focused mainly on two aspects:

- How to design basic prompts to guide the predictions of LLMs and refine these prompts for more effective and efficient problem-solving?
- How to automate the design and representation of prompts?

Solutions to these issues involve both general prompt designs and more advanced techniques, such as CoT and prompt learning, which have been explored extensively in recent research.

In NLP, prompting can be viewed as a technology that has evolved along with LLMs, and in a sense, it has opened the door to the practical application of these models in an impressive range of problem domains. In fact, if we expand the concept of prompts to some extent, it can be traced back to the early days of machine learning and NLP. For example, many NLP systems use hand-crafted features and templates to “prompt” specific tasks. Imagine developing a feature to indicate whether a text is formal or informal. We can feed this feature into a machine translation system to condition the translation on the type of the input text.

The widespread use of the modern concept of prompts began with the rise of large pre-trained models in the field of NLP. Initially, these models, such as BERT, were adapted to specific downstream tasks mainly through fine-tuning. However, researchers soon discovered that by designing specific "prompts" — adding certain words or sentences to the input — the models could be triggered to respond to specific tasks without extensive fine-tuning. This motivated the NLP community to develop and apply universal foundation models that can be prompted to address various tasks without changing the underlying architecture and the pre-training procedure.

Prompting approaches were first experimented with smaller models and later demonstrated impressive capabilities with large models like GPT-3, which could generate high-quality text in response to simple prompts across various tasks. As prompting technology evolved, prompt engineering emerged as a critical area of research. As discussed in this chapter, it broadly involves designing effective prompts to maximize model performance, encompassing both hand-crafted and automatically generated prompts. More recent research has explored how to enhance the effectiveness of prompting through techniques like few-shot learning, zero-shot learning, and CoT reasoning, enabling LLMs to work effectively across a wide range of scenarios. A general discussion of prompting can be very broad, and we cannot cover all details in this chapter. For more advanced techniques of prompting, the reader can refer to recent surveys. Topics include in-context learning [Li, 2023; Dong et al., 2022], CoT [Chu et al., 2023; Yu et al., 2023b; Zhang et al., 2023a], efficient prompting [Chang et al., 2024], and general prompt engineering [Liu et al., 2023d; Chen et al., 2023a].

Note that although we would ideally like to develop general prompting methods without adjusting model architectures and parameters, the results of prompting generally depend heavily on the quality and size of the given LLMs. For stronger models, such as commercialized online LLMs, simple prompts may be sufficient to instruct these models to perform tasks correctly. In this case, prompt engineering is relatively easy, though we still need certain efforts to make LLMs work properly. By contrast, if the LLMs are not powerful enough, we may need to carefully design the prompts to achieve the desired results. In many cases, fine-tuning is still necessary to adapt the models to sophisticated prompting strategies.



# Chapter 10

## Alignment

Alignment is not a new concept in NLP, but its meaning varies across different domains and over time. In traditional NLP, the term *alignment* typically refers to the tasks that link corresponding elements in two sets, such as aligning words between a Chinese sentence and an English sentence. As LLMs become increasingly important in NLP research, this term is more broadly used to refer to aligning model outputs with human expectations. The problem that alignment addresses is that the output of a model may not align with the specific goals or contexts intended by users. For example, pre-trained LLMs may not be able to follow user instructions because they were not trained to do so. Another example is that LLMs may generate harmful content or perpetuate biases inherent in their training data. This poses new challenges in ensuring that LLM outputs are not only accurate and relevant, but also ethically sound and non-discriminatory.

Simply pre-training LLMs can result in a variety of alignment problems. Our ultimate goal is to resolve or mitigate all these problems to ensure LLMs are both accurate and safe. There is an interesting issue here: since large language models are trained on vast amounts of data, we have reason to believe that if we have sufficient data covering a variety of tasks and aligned with human preferences, pre-training could make LLMs accurate and safe enough, perhaps even eliminating the need for alignment. However, the reality is that it is nearly impossible to gather data that encompasses all tasks or adequately represents human preferences. This makes it difficult to achieve model alignment through pre-training alone, or at least, at this stage, alignment remains a very necessary and critical step in the development of LLMs.

In this chapter, we will focus on alignment methods for LLMs. We will begin by discussing the general alignment tasks. Then we will consider two widely-used approaches, known as **instruction alignment** and **human preference alignment**, respectively. The former resorts to supervised fine-tuning techniques and guides the LLMs to generate outputs that adhere closely to user instructions. On the other hand, the latter typically relies on reinforcement learning techniques, where the LLMs are trained based on feedback from humans. While these methods are motivated by different goals, they are commonly used together to develop well-aligned LLMs.

## 10.1 An Overview of LLM Alignment

Alignment can be achieved in several different ways. We need different methods for LLM alignment because this problem is itself complicated and multifaceted, requiring a blend of technical considerations. Here we consider three widely-used approaches to aligning LLMs.

The first approach is to fine-tune LLMs with labeled data. This approach is straightforward as it simply extends the pre-existing training of a pre-trained LLM to adapt it to specific tasks. An example of this is **supervised fine-tuning (SFT)**, in which the LLM is further trained on a dataset comprising task-specific instructions paired with their expected outputs. The SFT dataset is generally much smaller compared to the original training set, but this data is highly specialized. The result of SFT is that the LLM can learn to execute tasks based on user instructions. These tasks can either be ones previously encountered in SFT, or new tasks similar to those. For example, by fine-tuning the LLM with a set of question-answer pairs, the model can respond to specific questions, even if not directly covered in the SFT dataset. This method proves particularly useful when it is relatively easy to describe the input-output relationships and straightforward to annotate the data.

The second approach is to fine-tune LLMs using reward models. One difficulty in alignment is that human values and expectations are complex and hard to describe. In many cases, even for humans themselves, articulating what is ethically correct or culturally appropriate can be challenging. As a result, collecting or annotating fine-tuning data is not as straightforward as it is with SFT. Moreover, aligning LLMs is not just a task of fitting data, or in other words, the limited samples annotated by humans are often insufficient to comprehensively describe these behaviors. What we really need here is to teach the model how to determine which outputs are more in line with human preferences, for example, we not only want the outputs to be technically accurate but also to align with human expectations and values. One idea is to develop a reward model analogous to a human expert. This reward model would work by rewarding the LLM whenever it generates responses that align more closely with human preferences, much like how a teacher provides feedback to a student. To obtain such a reward model, we can train a scoring function from human preference data. The trained reward model is then used as a guide to adjust and refine the LLM. This frames the LLM alignment task as a reinforcement learning task. The resulting methods, such as **reinforcement learning from human feedback (RLHF)**, have been demonstrated to be particularly successful in adapting LLMs to follow the subtleties of human behavior and social norms.

The third approach is to perform alignment during inference rather than during training or fine-tuning. From this perspective, prompting in LLMs can also be seen as a form of alignment, but it does not involve training or fine-tuning. So we can dynamically adapt an LLM to various tasks at minimal cost. Another method to do alignment at inference time is to rescore the outputs of an LLM. For example, we could develop a scoring system to simulate human feedback on the outputs of the LLM (like a reward model) and prioritize those that receive more positive feedback.

The three methods mentioned above are typically used in sequence once the pre-training is complete: we first perform SFT, then RLHF, and then prompt the LLM in some way

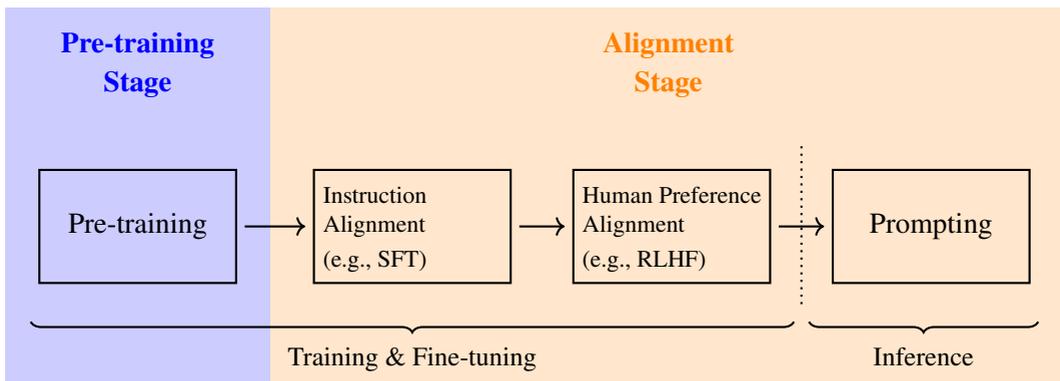


Figure 10.1: Schematic illustration of the pre-train-then-align method for developing LLMs. In the pre-training stage, we train an LLM on vast amounts of data using next token prediction. Then, in the alignment stage, we align the LLM to user instructions, intents, and preferences. This includes instruction alignment, human preference alignment, and prompting.

during inference. This roughly divides the development of LLMs into two stages — the pre-training stage and the alignment stage. Figure 10.1 shows an illustration of this. Since prompting techniques have been intensively discussed in the previous chapter, we will focus on fine-tuning-based alignment methods in the rest of this chapter.

## 10.2 Instruction Alignment

One feature of LLMs is that they can follow the prompts provided by users to perform various tasks. In many applications, a prompt consists of a simple instruction and user input, and we want the LLM to follow this instruction to perform the task correctly. This ability of LLMs is also called the instruction-following ability. For example, below is a prompt where we want the LLM to extract key points and provide a concise summary for a lengthy article.

Instruction	Summarize this text in three sentences.
Input	Daylight Savings Time (DST) - the process of moving clocks forward by one hour in the summer - was started in Germany in 1916. During World War One it was a way to save ...
Output	_____

This task requires the LLM to understand the instruction “Summarize this text in three sentences” and perform the summarization accordingly. However, LLMs are typically trained for next-token prediction rather than for generating outputs that follow instructions. Applying a pre-trained LLM to the above example would likely result in the model continuing to write the input article instead of summarizing the main points. The goal of instruction alignment

(or **instruction fine-tuning**) is to tune the LLM to accurately respond to user instructions and intentions. The rest of this section will discuss some issues related to instruction alignment, including fine-tuning LLMs to follow instructions, generating or collecting instruction data, and generalizing instruction alignment.

### 10.2.1 Supervised Fine-tuning

One straightforward approach to adapting LLMs to follow instructions is to fine-tune these models using annotated input-output pairs [Ouyang et al., 2022; Wei et al., 2022a]. Unlike standard language model training, here we do not wish to maximize the probability of generating a complete sequence, but rather maximize the probability of generating the rest of the sequence given its prefix (i.e., generating the output given the input). This approach makes instruction fine-tuning a bit different from pre-training. Let  $\mathbf{x} = x_0 \dots x_m$  be an input sequence (e.g., instruction + user input) and  $\mathbf{y} = y_1 \dots y_n$  be the corresponding output sequence. The SFT data is a collection of such input-output pairs (denoted by  $S$ ), where each output is the correct response for the corresponding input instruction. For example, below is an SFT dataset

$\mathbf{x}$ (instruction + user input)	$\mathbf{y}$ (output)
Summarize the following article. Article: In recent years, solar energy has seen unprecedented growth, becoming the fastest-growing ...	{*summary*}
Analyze the sentiment of the following review. Review: I absolutely loved the new dining experience. The food was divine and the service was impeccable.	Positive
Translate the following sentence into French. Sentence: practice indeed helps.	La pratique aide effectivement.
Extract the main financial figures from the following earnings report. Report: The company reported a revenue of \$10 million in the first quarter with a profit margin of 15% ...	Revenue: \$10 million, Profit Margin: 15%
Classify the following email as spam or not spam. Text: Congratulations! You've won a \$500 gift card. Click here to claim now.	Spam
Provide a solution to the following technical issue. Issue: my computer is running slow and often freezes.	First, check for ...

where the instructions are highlighted. This dataset contains instructions and the corresponding outputs for several different NLP problems, and so we can fine-tune an LLM to handle multiple tasks simultaneously.

In SFT, we aim to maximize the probability of the correct output given the input. Consider an LLM with pre-trained parameters  $\hat{\theta}$ . The fine-tuning objective can then be formulated as:

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x}) \quad (10.1)$$

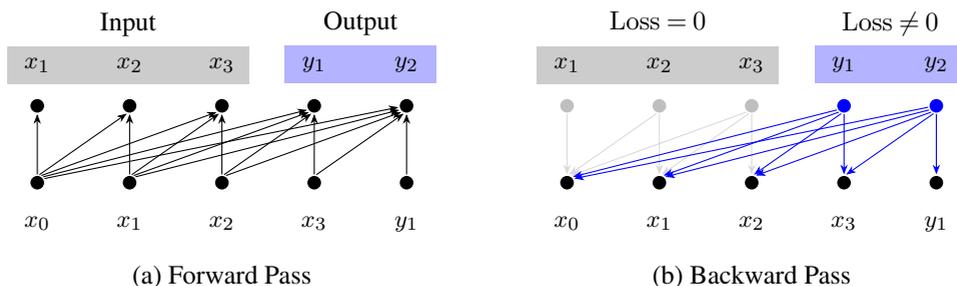


Figure 10.2: Illustration of supervised fine-tuning for LLMs. We concatenate the input and the output into a single sequence. During the forward pass, we run the LLM as usual. During the backward pass, we compute the loss only for the output part and simply set the loss for the input part to 0.

where  $\tilde{\theta}$  denotes the parameters optimized via fine-tuning, and  $\hat{\theta}^+$  represents an adjustment to  $\hat{\theta}$ . Here we will omit the superscript  $+$  and use  $\theta$  to represent  $\hat{\theta}^+$  to keep the notation uncluttered. But the reader should keep in mind that the fine-tuning starts from the pre-trained parameters rather than randomly initialized parameters.

The objective function  $\log \Pr_{\theta}(y_i | \mathbf{x}, \mathbf{y}_{<i})$  is computed by summing the log-probabilities of the tokens in  $\mathbf{y}$ , conditional on the input  $\mathbf{x}$  and all the previous tokens  $\mathbf{y}_{<i}$ :

$$\log \Pr_{\theta}(\mathbf{y} | \mathbf{x}) = \sum_{i=1}^n \log \Pr_{\theta}(y_i | \mathbf{x}, \mathbf{y}_{<i}) \quad (10.2)$$

This formulation is equivalent to minimizing the cross-entropy loss.

Note that minimizing the conditional log-probability  $\log \Pr_{\theta}(\mathbf{y} | \mathbf{x})$  is not a standard language model training problem. If we concatenate  $\mathbf{x}$  and  $\mathbf{y}$  as a single sequence, a more general form of language modeling is based on the joint log-probability  $\log \Pr_{\theta}(\mathbf{x}, \mathbf{y})$ , that is, we minimize the loss over all tokens of the sequence  $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$ . We can write the probability of this sequence using the chain rule

$$\begin{aligned} \log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}}) &= \log \Pr_{\theta}(\mathbf{x}, \mathbf{y}) \\ &= \underbrace{\log \Pr_{\theta}(\mathbf{x})}_{\text{set to 0}} + \underbrace{\log \Pr_{\theta}(\mathbf{y} | \mathbf{x})}_{\text{loss computation}} \end{aligned} \quad (10.3)$$

There are two terms on the right-hand side of the equation. We can simply set the first term  $\log \Pr_{\theta}(\mathbf{x})$  to 0, focusing solely on the second term  $\log \Pr_{\theta}(\mathbf{y} | \mathbf{x})$  for loss computation. As a result, the training can be implemented using standard LLMs. For the sequence  $\text{seq}_{\mathbf{x}, \mathbf{y}}$ , we first run the forward pass as usual. Then, during the backward pass, we force the loss corresponding to  $\mathbf{x}$  to be zero. Figure 10.2 shows an illustration of this process.

By taking  $\log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}})$  as the objective function, we can describe SFT using a regular

form of language model training:

$$\tilde{\theta} = \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}}) \quad (10.4)$$

The problem we considered above is fundamentally a **single-round prediction** problem, where the LLM generates a response based on a single input without any further interaction or feedback from the user. The input is processed, and the output is generated in one go. This is typical in scenarios where a single question is asked, and a single answer is provided, with no follow-up questions or clarifications. However, in practice, we sometimes have to handle multi-round prediction problems, for example, an LLM engages in a dialogue over multiple turns. In this setting, the LLM not only generates responses based on the initial input but also incorporates subsequent inputs that might refine or expand on earlier interactions. For example, we can use the LLM to act as a healthcare assistant chatbot and have a conversation with the user, like this

User I've been feeling very tired lately.

Chatbot I'm sorry to hear that. Besides feeling tired, have you noticed any other symptoms?

User Yes, I'm also experiencing headaches frequently.

Chatbot How long have these symptoms been going on?

User About a week now.

Chatbot It might be good to check in with a healthcare professional. Would you like help setting up an appointment?

User Yes, please. Can it be after work hours?

Chatbot Sure, I can arrange that. There are slots available next Wednesday and Thursday after 5 PM. Which day works better for you?

...

In this task, there are several rounds of conversation, each involving the generation of a response based on the user's request or question and the conversational history. Suppose we have  $K$  rounds of conversation, denoted by  $\{\mathbf{x}^1, \mathbf{y}^1, \mathbf{x}^2, \mathbf{y}^2, \dots, \mathbf{x}^K, \mathbf{y}^K\}$ . Here  $\mathbf{x}^k$  and  $\mathbf{y}^k$  denote the user request and the response, respectively, for each round  $k$ . The log-probability of generating the response can be written as  $\log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^k)$ . Our goal is then to maximize the sum of these log-probabilities

$$\tilde{\theta} = \arg \max_{\theta} \sum_{k=1}^K \log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^k) \quad (10.5)$$

A straightforward implementation of this involves calculating the conditional probability for each  $k$ . However, it requires running the LLM  $K$  times, each time with an increased conversational history to make predictions. A more efficient method is to perform loss computation of all responses in a single run of the LLM. To do this, we represent the conversation as a sequence  $\text{seq}_{\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K} = [\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K]$  (or  $\text{seq}$  for short). The log-probability of this sequence is given by

$$\begin{aligned}
 \log \Pr_{\theta}(\text{seq}) &= \log \Pr_{\theta}(\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K) \\
 &= \underbrace{\log \Pr_{\theta}(\mathbf{x}^1)}_{\text{set to 0}} + \underbrace{\log \Pr_{\theta}(\mathbf{y}^1 | \mathbf{x}^1)}_{\text{loss computation}} + \dots + \\
 &\quad \underbrace{\log \Pr_{\theta}(\mathbf{x}^K | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{y}^{K-1})}_{\text{set to 0}} + \\
 &\quad \underbrace{\log \Pr_{\theta}(\mathbf{y}^K | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K)}_{\text{loss computation}}
 \end{aligned} \tag{10.6}$$

The trick here is that we ignore the loss for generating user inputs (i.e.,  $\log \Pr_{\theta}(\mathbf{x}^1), \dots, \log \Pr_{\theta}(\mathbf{x}^K | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{y}^{K-1})$ ), as illustrated in Figure 10.3. Hence we only compute the probabilities of generating the responses given their conversational histories, in other words, the value on the right-hand side of Eq. (10.6) is actually equal to the value on the right-hand side of Eq. (10.5). As with Eq. (10.4), the training of this multi-round prediction model can be achieved by maximizing the log likelihood over a training dataset  $\mathcal{D}$ :

$$\tilde{\theta} = \arg \max_{\theta} \sum_{\text{seq} \in \mathcal{D}} \log \Pr_{\theta}(\text{seq}) \tag{10.7}$$

While implementing the SFT methods introduced above seems trivial as they are fundamentally the same as regular language model training, there are still issues that need to be considered in practice. For example,

- SFT requires labeled data. This makes SFT quite different from pre-training, where raw text is used as training data and is readily available. As in other supervised machine learning problems, data annotation and selection in SFT are not simple tasks. In general, we wish to develop SFT data that is both substantial in quantity and high in quality, and this data should be highly relevant to the tasks the LLM will perform. On the other hand, there is a need to fine-tune LLMs with less data to minimize computational and data construction costs. Often, the quality of LLMs is highly dependent on the data used in SFT. Thus, such data must be carefully developed and examined. As we will see in later subsections, SFT can be more efficient and effective through more advanced techniques for data construction.
- SFT is still computationally expensive for LLMs due to their large size. As a result, maintaining and updating such models is resource-intensive. For example, applying gradient updates to billions of parameters within an LLM requires significant computational power and memory. This often requires high-performance computing environments,

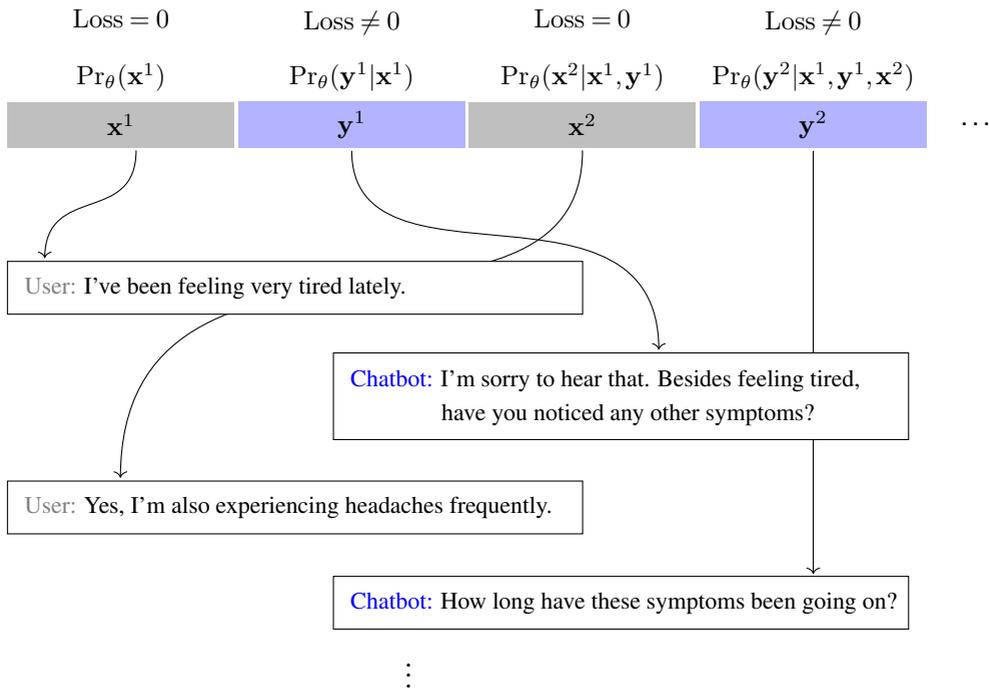


Figure 10.3: Illustration of supervised fine-tuning for conversational models. Here the LLM acts as a chatbot to respond to each request based on the conversational history. The conversation progresses by alternating between the user and the chatbot. In SFT, we treat the entire conversation as a sequence, just like in standard LLMs, but compute the loss only for the responses of the LLM.

which are costly to operate. To address these challenges, various optimization strategies, such as pruning, quantization, and the use of more efficient training algorithms, have been explored. In particular, there has been significant interest in parameter-efficient fine-tuning methods which are designed to maintain state-of-the-art performance without the need for extensive computational resources. We have seen in Chapter 9 that applying techniques like soft prompts can make the fine-tuning process more efficient. For further discussion on parameter-efficient methods, the reader can refer to related papers on this issue [Houlsby et al., 2019; Hu et al., 2022; Han et al., 2024].

- SFT can be regarded as a post-training step following pre-training. It is a separate training phase designed to preserve the advantages of the initial pre-training while incorporating new adjustments. This may seem paradoxical because updating a pre-trained LLM with further data potentially causes the model to forget some of its prior knowledge. Imagine a scenario where we have a large amount of SFT data and extensively fine-tune the LLM. In this case, the LLM could overfit the data, which in turn may reduce generalization performance or cause catastrophic forgetting. A common strategy to mitigate this issue is to employ regularization and early stopping techniques. Another

practical approach is to use a smaller learning rate to gently adjust the weights of the LLM. In addition, fine-tuning with data from diverse sources and problem domains can also be beneficial. Nevertheless, in practice, the SFT step is often carefully examined and requires substantial engineering and experimental efforts to optimize.

## 10.2.2 Fine-tuning Data Acquisition

Fine-tuning data is so important that much recent work in LLM has focused on developing various datasets for instruction fine-tuning. As with most work in machine learning, there are generally two approaches to data acquisition — manual data generation and automatic data generation.

### 1. Manually Generated Data

One straightforward method is to recruit human annotators to create input-output pairs for the tasks of interest. Unlike data annotation in conventional NLP, such as text classification, where annotators simply assign labels to collected texts according to guidelines, creating fine-tuning data for LLMs requires more steps and effort, making it thus more challenging. Suppose we want to obtain fine-tuning data for the English-to-Chinese machine translation task. The first step is to write a prompt template to describe the task and format the problem clearly. For example,

```
Instruction  Translate the text from English to Chinese.
User Input  {*text*}
Output     {*translation*}
```

Then, we collect pairs of source and target texts (i.e., Chinese texts and the corresponding translations), and replace the variables `{*text*}` and `{*translation*}` to generate the fine-tuning samples. For example, given a pair of English and Chinese sentences

```
How's the weather today?  →  今天天气怎么样?
      {*text*}                {*translation*}
```

we can generate a fine-tuning sample using the prompt template, like this

```
Instruction  Translate the text from English to Chinese.
User Input  How's the weather today?
Output     今天天气怎么样?
```

That is,

$x$  = Translate the text from English to Chinese.\n How's the weather today?  
 $y$  = 今天天气怎么样?

We can use this  $(x, y)$  pair to fine-tune the LLM, as described in the previous subsection.

One difficulty here is that there are many, many different ways to write prompt templates for the same task, and different people may produce prompt templates with varying qualities and complexities. Sometimes, we may write prompt templates with overly complex or verbose instructions. Sometimes, we may not even know exactly what the target task is and how to describe it. A widely-adopted strategy is to create prompt templates for existing NLP tasks, given that there have been so many well-established NLP problems and benchmarks [Bach et al., 2022; Wang et al., 2022e; Mishra et al., 2022]. In this case, annotators can be given the original task description and many examples. Then, they can use their own ways to express how to prompt the LLM to perform the tasks. Note that, while such a method can ease the process of creating and writing prompts, we still need annotation frameworks and crowdsourcing systems to manage the work and conduct quality control. For example, we generally need to design annotation guidelines and a unified format for writing prompt templates, especially when many annotators are contributing to the same task. One advantage of inducing prompts from existing NLP tasks is that, once the prompt templates have been developed, it is easy to generate prompts using the annotated samples in the original tasks. For example, given a bilingual dataset for English-to-Chinese translation, we can easily create a number of fine-tuning examples by filling the slots in the above template with the sentence pairs in this dataset.

Another approach is to directly use the naturally existing data available on the internet. A common example is by collecting question-and-answer pairs from QA websites to fine-tune LLMs for open-domain QA tasks [Joshi et al., 2017]. Many benchmarks in QA are built in this way because there are so many types of questions that it is impossible to think of them all by a small group of people. Instead, using data from those websites can ensure that the LLM fine-tuning data is at a good or acceptable level in terms of quantity and quality.

In addition to employing existing resources, another straightforward way to develop a fine-tuning dataset is to crowdsource the data. A simple approach is to allow users to input any question, after which responses are either manually given or automatically generated by an LLM and then manually annotated and corrected. It is thus possible to capture real user behavior and consequently gather inputs and outputs for a large number of “new” problems that traditional NLP tasks do not cover.

An issue related to the construction of the fine-tuning datasets is that we usually want the data to be as diverse as possible. Many studies have found that increasing the diversity of fine-tuning data can improve the robustness and generalization ability of LLMs. For this reason, there has been considerable interest in involving more diverse prompts and tasks in LLM fine-tuning datasets. We will provide further discussion on the generalization of fine-tuning in Section 10.2.4.

## 2. Automatically Generated Data

One limitation of manual data generation is that the quality and diversity largely depend on human experience and creativity. Therefore, if we want LLMs to handle a broad range of tasks, that is, to effectively execute any instruction, relying on human-annotated data for LLM fine-tuning is often inefficient. Moreover, the coverage of such data can be limited, and the data may even contain biases introduced by the annotators themselves. An alternative approach is to generate data automatically. For example, we can collect a number of questions through crowdsourcing, and employ a well-tuned LLM to generate answers to the questions. These question-answer pairs are then used as fine-tuning samples as usual. This method, though very simple, has been extensively applied to generate large-scale fine-tuning data for LLMs.

The above way of producing synthetic fine-tuning data is similar to those used in data augmentation for NLP. If we have an LLM, we can produce a prediction in response to any input. Repeating this process for different inputs allows us to create a sufficient number of fine-tuning samples. Such a method is particularly useful for fine-tuning new LLMs using a well-tuned LLM. However, one disadvantage of this approach is that it relies on human-crafted or collected inputs for data generation, which may turn out to be inappropriate for generalizing LLMs. In many LLM applications, a significant challenge arises from the broad range of users' questions and requests, many of which are not covered in existing NLP tasks and datasets. In these cases, it becomes necessary to generate not only the predictions but also the inputs themselves.

Here we consider **self-instruct** as an example to illustrate how to generate LLM fine-tuning samples [Wang et al., 2023e; Honovich et al., 2023]. The idea is that we can prompt an LLM to create a new instruction by learning from other instructions. Given this instruction, the LLM can then fill in other fields (such as the user input) and produce the predictions. Figure 10.4 shows a schematic illustration of self-instruct. Here we give a brief outline of the key steps involved.

- The self-instruct algorithm maintains a pool of tasks. Initially it contains a number of seed hand-crafted tasks, each with an instruction and input-output sample. As the algorithm proceeds, LLM-generated instructions and samples will be added to this pool.
- At each step, a small number of instructions are drawn from the instruction pool. For example, we can randomly select a few human-written instructions and a few LLM-generated instructions to ensure diversity.

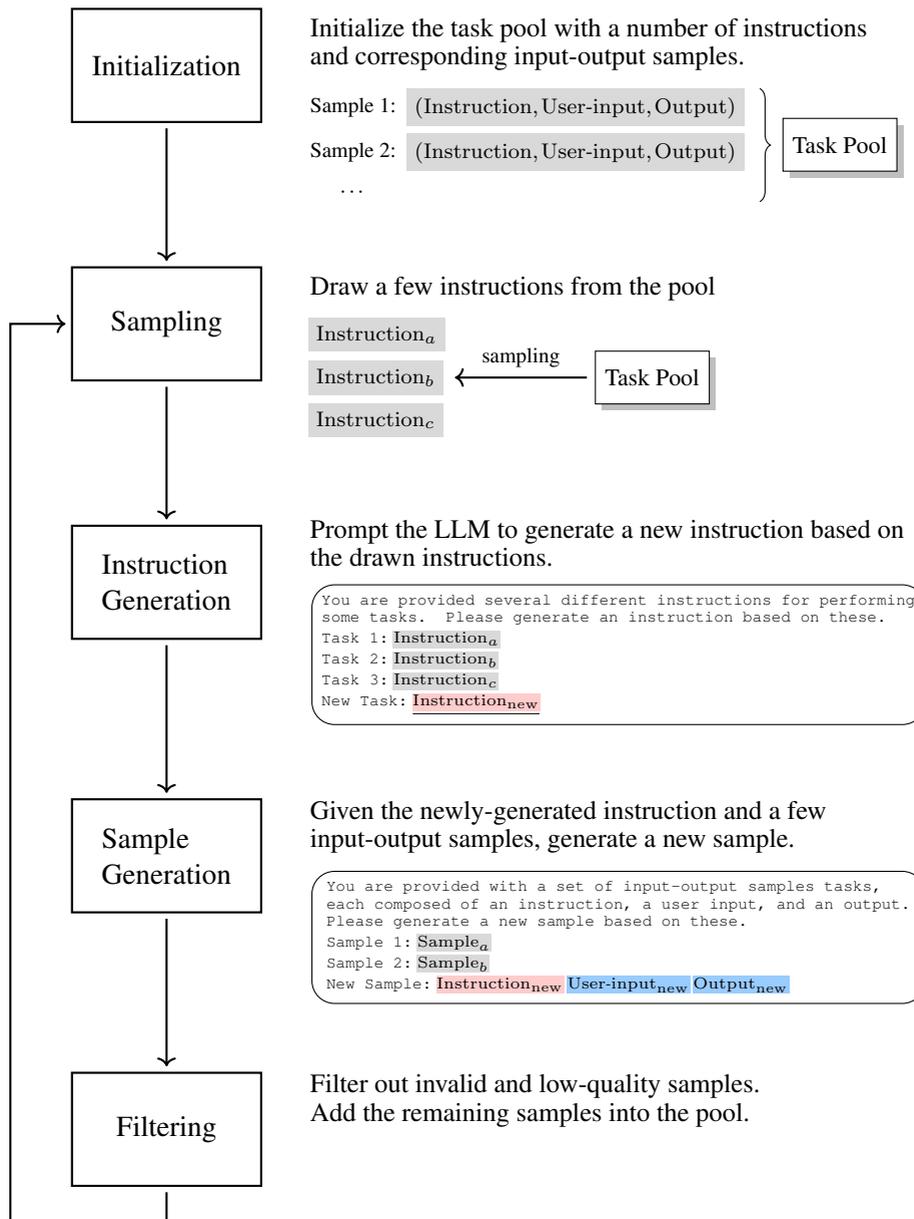


Figure 10.4: Illustration of self-instruct [Wang et al., 2023c]. This method maintains a pool of instructions and corresponding input-output samples. Initially, the pool contains a number of hand-crafted instructions and samples. Each time, we draw a few instructions from the pool. An LLM is then prompted to generate new instructions and samples based on those drawn. Finally, the newly-generated instructions and samples are filtered and added to the pool.

- The selected instructions are then used as demonstration examples. Thus, the LLM can in-context learn from these examples and produce a new instruction. Below is an example template for prompting the LLM.

You are provided several different instructions for performing some tasks. Please generate an instruction based on these.

Task 1: {instruction1}

Task 2: {instruction2}

Task 3: {instruction3}

Task 4: {instruction4}

New Task: \_\_\_\_\_

- Given the generated instruction, the LLM is then prompted to complete the sample by filling in the remaining input fields and generating the corresponding output. Below is a prompt template.

You are provided with a set of input-output samples, each composed of an instruction, a user input, and an output. Please generate a new sample based on these.

Sample 1: {instruction1}

Input: {user-input1}

Output: {output1}

Sample 2: {instruction2}

Input: {user-input2}

Output: {output2}

New Sample: {new-instruction}

\_\_\_\_\_

- This newly-generated sample is examined by some heuristic rules (such as filtering out samples or instructions that are similar to those already in the pool). If it passes, the sample and instruction are added to the pool.

This generation process can be repeated many times to obtain a sufficient number of fine-tuning samples. Note that, above, we just show simple prompt templates for generating instruction and fine-tuning samples. Of course, we can develop better templates to generate more diverse and accurate instruction and fine-tuning samples. For example, for certain tasks like text classification, the LLM may tend to produce biased predictions, for example, most generated samples belong to a single class. In such cases, we can adjust the order of generation of different fields. More specifically, we can specify the output (i.e., the class) with some prior, and prompt the LLM to generate user input given both the instruction and the output. This

method resembles **input inversion**, where the LLM generates the input based on the specified output [Longpre et al., 2023].

Using LLM-generated instructions and fine-tuning samples has been a common method for developing LLMs, especially given that manually developing such data is so expensive that most research groups cannot afford it. In several well-tuned LLMs, their fine-tuning datasets include a certain amount of synthetic data, which has proved useful [Ouyang et al., 2022; Taori et al., 2023; Chiang et al., 2023b]. There have been further studies on synthetic data generation for LLM fine-tuning. For example, one can generate more diverse instructions by introducing evolutionary algorithms [Xu et al., 2024], or use synthetic data as supervision signals in a more advanced fine-tuning process [Chen et al., 2024b]. More recently, there has also been considerable interest in using synthetic data in the pre-training stage [Gunasekar et al., 2023; Allal et al., 2024].

In many applications, a real-world scenario is that, given a task, we can collect or annotate a relatively small amount of fine-tuning data, for example, we can recruit experts to create questions for QA tasks in a specific domain. But the quantity and diversity of this data are in general not sufficient. In this case, we can use self-instruct techniques to generate more diverse question-answer pairs, and thus augment the fine-tuning data. This provides a way of bootstrapping the LLM starting from a seed set of fine-tuning samples. Note that using self-generated data is a common practice and has long been applied in NLP. For example, this approach has been successfully used in parsing and machine translation [Charniak, 1997; Senrich et al., 2016a].

### 10.2.3 Fine-tuning with Less Data

With the increasing prominence of instruction fine-tuning, there has been a surge in demand for large-scale, high-quality fine-tuning data. For example, the FLAN fine-tuning dataset, which is compiled from 1,836 tasks, contains 15 million samples [Longpre et al., 2023]. Fine-tuning LLMs with such large datasets is typically a computationally expensive task, especially given that updating the large number of parameters in LLMs is resource-intensive. One approach for mitigating this issue is to explore efficient model training methods, for example, one can use parameter-efficient methods to update only a small portion of the model. However, many fine-tuning datasets contain a large amount of synthetic data, where errors and biases are still inevitable.

Another approach to efficient fine-tuning is to consider only the most relevant and impactful examples for fine-tuning. We can thus reduce the amount of data that needs to be processed while still maintaining the quality of the model updates. There are several methods to achieve this. For example, Zhou et al. [2023a] built an instruction-following dataset containing only 1,000 samples by carefully crafting the prompts and collecting samples from a variety of NLP tasks. They showed that the LLaMa 65B model fine-tuned with this dataset could be competitive with or even better than models with much more fine-tuning effort. This suggests that LLMs can be adapted to respond to diverse tasks without necessarily needing fine-tuning on all types of instruction-following data. Chen et al. [2024a] developed a system based on the GPT-3.5 model to assess the quality of each instruction-following sample. Therefore,

they could select high-quality samples from existing datasets, showing better fine-tuning performance with fewer fine-tuning samples. Researchers have also developed methods to either select or filter out data using heuristics [Zhao et al., 2024; Ge et al., 2024], or to prioritize data that more significantly influences the fine-tuning process [Xia et al., 2024]. In fact, most of these methods can be seen as instances of larger families of data selection and filtering methods. And it is often the case that using higher quality (but maybe less) data is beneficial for training NLP models.

The discoveries in instruction fine-tuning somewhat differ from traditional views in NLP: the ability of models to handle complex problems can be activated with a small amount of annotated data, rather than requiring massive amounts of supervised data for extensive training. One possible explanation is that the ability of generating correct responses given instructions has been learned during pre-training, but such instruction-response mappings are not with high probabilities during inference. Fine-tuning can slightly adjust the models to get them to follow instructions, requiring significantly less training effort than pre-training. This is closely related to what is known as the **superficial alignment hypothesis**, which suggests that learning primarily occurs during pre-training, and the subsequent fine-tuning or alignment phase does not significantly contribute to the underlying knowledge base of an LLM [Zhou et al., 2023a]. Since the core abilities and knowledge of the model are already established from pre-training, effective fine-tuning for alignment with user needs can be achieved with relatively small training fine-tuning effort. This implies the possibility of fine-tuning LLMs with very little data. In another direction, it may not be necessary to restrict fine-tuning to paired instruction-response data. For example, Hewitt et al. [2024] found that instruction-following can be implicitly achieved by fine-tuning LLMs only on responses, without corresponding instructions.

A concept related to the discussion here is sample efficiency. A machine learning method is called **sample efficient** if it can learn effectively from a small number of training examples. In this sense, instruction fine-tuning is sample efficient compared with pre-training. From the perspective of machine learning, sample-efficient methods can be seen as efficient ways to sample the space of data, and are advantageous as they make optimal use of scarce data. Therefore, sampling-based learning techniques, such as many reinforcement learning algorithms, can benefit from these sample efficient approaches. For example, in human preference alignment, we can either efficiently sample preference data via reward models [Liu et al., 2024b] or improve the sampling efficiency in policy learning [Wang et al., 2024].

### 10.2.4 Instruction Generalization

In many machine learning and NLP problems, training a model to generalize is a fundamental goal. For example, in text classification, we expect our model to correctly classify new texts that were not seen during training. However, generalization poses additional challenges in instruction fine-tuning. We expect instruction-fine-tuned LLMs to not only generate appropriate responses for different inputs within a task but also to accurately perform various tasks as described by different instructions. To illustrate this issue, consider an LLM  $\Pr(\mathbf{y}|\mathbf{c}, \mathbf{z})$ , where  $\mathbf{c}$  is an instruction,  $\mathbf{z}$  is a user input, and  $\mathbf{y}$  is the corresponding model output (i.e., the

response). Suppose that the performance of this model is evaluated in terms of a metric, written as  $\text{Performance}(\Pr(\mathbf{y}|\mathbf{c}, \mathbf{z}))$  or  $P(\mathbf{c}, \mathbf{z}, \mathbf{y})$  for short. Informally, when we say this model can generalize within a given task (indicated by the instruction  $\mathbf{c}^*$ ), we mean that there may be a value  $\epsilon$  such that the average performance on new inputs is above this value:

$$\frac{1}{|\mathcal{Z}|} \sum_{\mathbf{z}' \in \mathcal{Z}} P(\mathbf{c}^*, \mathbf{z}', \mathbf{y}') > \epsilon \quad (10.8)$$

where  $\mathcal{Z}$  is the set of new inputs, and  $\mathbf{z}'$  and  $\mathbf{y}'$  are an input in this set and the corresponding output, respectively.

Likewise, we can say that this model can generalize across tasks if the average performance over all instruction-input pairs is above some  $\epsilon$ :

$$\frac{1}{|\mathcal{D}|} \sum_{(\mathbf{c}', \mathbf{z}') \in \mathcal{D}} P(\mathbf{c}', \mathbf{z}', \mathbf{y}') > \epsilon \quad (10.9)$$

where  $\mathcal{D}$  is the set of new instruction-input pairs.

Here, we need to deal with variations in two dimensions: instruction and user input. This makes the generalization problem very complex, because, intuitively, a model needs to learn from a vast number of tasks and different input-output pairs associated with each task to achieve good generalization. As we have discussed several times in this book, achieving such generalization incurs much lower cost than pre-training. In general, fine-tuning LLMs with instruction-response data to some extent can lead to models yielding instruction following on new tasks. Nevertheless, it is typically believed that certain efforts are still needed to adapt LLMs to make them understand and execute instructions broadly.

One way to generalize instruction fine-tuning is to increase the diversity of the fine-tuning data. In earlier studies on instruction fine-tuning, researchers developed many datasets, covering a wide variety of NLP tasks and different instructions for each task [Wang et al., 2022e; Sanh et al., 2022; Longpre et al., 2023]. By transforming these tasks into a unified format, one can fine-tune an LLM with a sufficiently large number of samples, for example, there have been several instruction fine-tuning datasets that involve over 100 NLP tasks and 1M samples. However, these early datasets mostly focus on existing academic problems, but not those that users want to deal with in real-world applications. Much recent work has shifted focus to addressing new and more practical problems. For example, there has been considerable interest in constructing datasets that contain large and complicated demonstrations and responses from SOTA models to real user queries [Wang et al., 2023d; Teknium, 2023].

Perhaps the use of large and diverse fine-tuning datasets has its origins in attempts to scale LLMs in different dimensions. Indeed, scaling laws have been used broadly to motivate the development of a wide range of different instruction-fine-tuned LLMs. And it is reasonable to scale instruction fine-tuning to make an LLM follow broad instructions. From the perspective of LLM alignment, however, scaling instruction fine-tuning might not be efficient to achieve generalization.

One problem is that instruction fine-tuning relies on supervised learning that learns to

generalize and perform tasks based on instruction-response mappings. However, such an approach does not capture subtle or complex human preferences (e.g., tone, style, or subjective quality) because these are hard to encode as explicit instruction-response data. Moreover, the generalization performance is bounded by the diversity and quality of the instruction-response dataset. Given these limitations, we would instead like to employ preference models as an additional fine-tuning step following instruction fine-tuning, so the LLMs can generalize further (see Section 10.3).

Another view is that some instruction-response mappings may already be learned during pre-training, and so the pre-trained LLMs have encoded such mappings. However, since we often do not know exactly what data is used in the pre-training, it is hard to judge whether we need to learn such mappings in the fine-tuning. A related question is whether out-of-distribution generalization is primarily achieved during pre-training or fine-tuning. While directly answering this question is beyond the scope of this chapter, it has been shown that pre-training on large and diverse datasets is effective in improving out-of-distribution performance [Hendrycks et al., 2020; Radford et al., 2021; Gunasekar et al., 2023]. This raises an interesting problem: if an LLM has been well pre-trained at scale, fine-tuning may not be as essential for out-of-distribution generalization, since the model may have already encountered sufficient distributional variation. This prompts researchers to fine-tune LLMs with modest effort or to explore new methods to achieve instruction-following. As discussed in the previous subsection, for example, instruction following can be yielded by fine-tuning on a small number of carefully selected instruction-response pairs [Zhou et al., 2023a], or even by using methods that are not explicitly designed to do so [Kung and Peng, 2023].

The above discussion provides two different strategies: one requires scaling up fine-tuning datasets for larger diversity, the other requires small but necessary fine-tuning datasets for efficient LLM adaptation. However, in practice, involving diverse instructions often helps. In many cases, we need to adapt our LLM for specific purposes. But the LLM, which has possibly encoded broad instruction-following mappings during pre-training, might tend to behave as a general-purpose instruction executor even with modest fine-tuning. An interesting phenomenon is that when fine-tuning on math data, the resulting LLM might not specialize in math outputs. Instead, this model might respond normally to general instructions, for example, it could generate poetry if instructed to do so [Hewitt, 2024]. This is not a bad thing, but it shows that LLMs may not easily change their nature of following general instructions. In this case, additional adaptations with more diverse data may help adjust the way the LLM follows instructions, particularly for those tasks we aim to address.

### 10.2.5 Using Weak Models to Improve Strong Models

So far we have explored a variety of instruction fine-tuning methods based on labeled data. One of the limitations of many such methods is that they require the data to be annotated by humans or generated by strong LLMs, which can provide accurate supervision signals in fine-tuning. However, in many cases, the LLM we have in hand is already strong (or at least is advantageous in specific aspects of problem solving), and thus it is not easy to find a superior model for supervision. Even for human experts, when the problem becomes complex,

providing correct and detailed answers might be difficult, or sometimes infeasible. For example, when faced with an extremely long document, the experts would find it challenging to identify any inconsistencies, subtle biases, or missing key points without conducting an exhaustive and time-consuming review.

One may ask at this point: can we use weak LLMs to supervise strong LLMs? This seems to be a significant challenge, but it may reflect a future scenario where we need to supervise AI systems that are smarter than humans or any other AI systems [Burns et al., 2023b]. The problem of using smaller, less complex models to improve the training of larger, more complex models is also called the **weak-to-strong generalization** problem. While there have not been mature approaches to weak-to-strong generalization, using smaller models to assist stronger models has indeed proven useful in several areas of LLMs.

For instruction fine-tuning, one of the simplest ways of applying weak LLMs is to use these models to generate synthetic fine-tuning data. Suppose we have a collection of inputs  $X$ , where each input includes an instruction and a user input if necessary. For each  $\mathbf{x} \in X$ , we use a weak LLM  $\text{Pr}^w(\cdot)$  to generate a prediction  $\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \text{Pr}^w(\mathbf{y}|\mathbf{x})$ . Then, the strong LLM  $\text{Pr}_{\theta}^s(\cdot)$  can be trained on these generated predictions (see Eq. (10.1)):

$$\tilde{\theta} = \arg \max_{\theta} \sum_{\mathbf{x} \in X} \log \text{Pr}_{\theta}^s(\hat{\mathbf{y}}|\mathbf{x}) \quad (10.10)$$

where  $\theta$  is the model parameters.

The above form transforms the fine-tuning problem into a knowledge distillation problem, in other words, we distill knowledge from the weak model to the strong model. Consequently, we can employ various knowledge distillation methods to achieve this goal. However, explaining weak-to-strong fine-tuning from the perspective of knowledge distillation is not straightforward. A major concern is that the strong model may merely imitate or overfit the errors of the weak model and fail to generalize. For example, the fine-tuned strong model still cannot solve difficult problems that the weak model cannot accurately predict. Fortunately, preliminary experiments in this line of research have shown positive and promising results. For example, Burns et al. [2023a] found that fine-tuning the strong pre-trained GPT-4 model with GPT-2-level supervision could improve generalization across several NLP tasks. To measure how the weak model improves the generalization of the strong model, we define the following terms:

- **Weak Performance** ( $P_{\text{weak}}$ ). This is the test-set performance of the weak model, which can be regarded as the baseline performance.
- **Weak-to-strong Performance** ( $P_{\text{weak} \rightarrow \text{strong}}$ ). This is the test-set performance of the strong model that is fine-tuned with the weak model.
- **Strong Ceiling Performance** ( $P_{\text{ceiling}}$ ). This is the test-set performance of the strong model that is fine-tuned with ground truth data. For example, we fine-tune the strong model with human-annotated predictions and take the resulting model as a ceiling.

Then, the **performance gap recovered (PGR)** can be defined as

$$\text{PGR} = \max \left\{ 0, \frac{P_{\text{weak} \rightarrow \text{strong}} - P_{\text{weak}}}{P_{\text{ceiling}} - P_{\text{weak}}} \right\} \quad (10.11)$$

This metric measures how much of the performance gap between the ceiling model and the weak model can be recovered by the weak-to-strong model. A PGR of 1 indicates that the weak-to-strong fine-tuning can completely close the performance gap, whereas a PGR of 0 indicates no improvement. In [Burns et al. \[2023a\]](#)'s work, it is shown that PGR can be around 0.8 on 22 NLP classification tasks. It should be noted that, while the potential of weak-to-strong fine-tuning is promising, achieving substantial weak-to-strong generalization remains a challenging goal that needs further investigation [[Aschenbrenner, 2024](#)].

Fine-tuning LLMs with weak supervision is just one choice for using small models to improve large models. Although this section primarily focuses on fine-tuning LLMs, we also mention other methods here to give a more complete discussion (see [Figure 10.5](#) for illustrations of these methods).

- Instead of using small models to generate synthetic data, it is also straightforward to incorporate knowledge distillation loss based on these models. For example, a simple loss function that measures the difference between the small and large models can be defined as:

$$\text{Loss}_{\text{kd}} = \text{KL}(\text{Pr}^w(\cdot|\mathbf{x}) \parallel \text{Pr}_\theta^s(\cdot|\mathbf{x})) \quad (10.12)$$

Then, we can add this loss to the original loss of language modeling, and yield the following training objective

$$\tilde{\theta} = \underset{\theta}{\text{argmax}} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \text{Pr}_\theta^s(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Loss}_{\text{kd}} \quad (10.13)$$

where  $\mathcal{D}$  is the set of input and output pairs, and  $\lambda$  is the coefficient of the interpolation. This method can be employed in either the pre-training or fine-tuning phase. We can adjust  $\lambda$  to control how much the small model influences the training. For example, we can gradually decrease  $\lambda$  to make the training rely more on the original language modeling loss as the large model becomes more capable.

- Another approach to involving small models in LLM pre-training and fine-tuning is to use them to do data selection or filtering. Given a sequence, we can compute the likelihood or cross-entropy using a small model. These quantities can then be used as criteria for selecting or filtering data. For example, sequences with low likelihood or high cross-entropy might be excluded from the training set, as they are less aligned with the small model's learned distribution. Conversely, sequences with high likelihood or low cross-entropy can be prioritized, ensuring that the training focuses on more relevant or high-quality data.
- Ensemble learning is a simple and effective way to build a strong model by combining

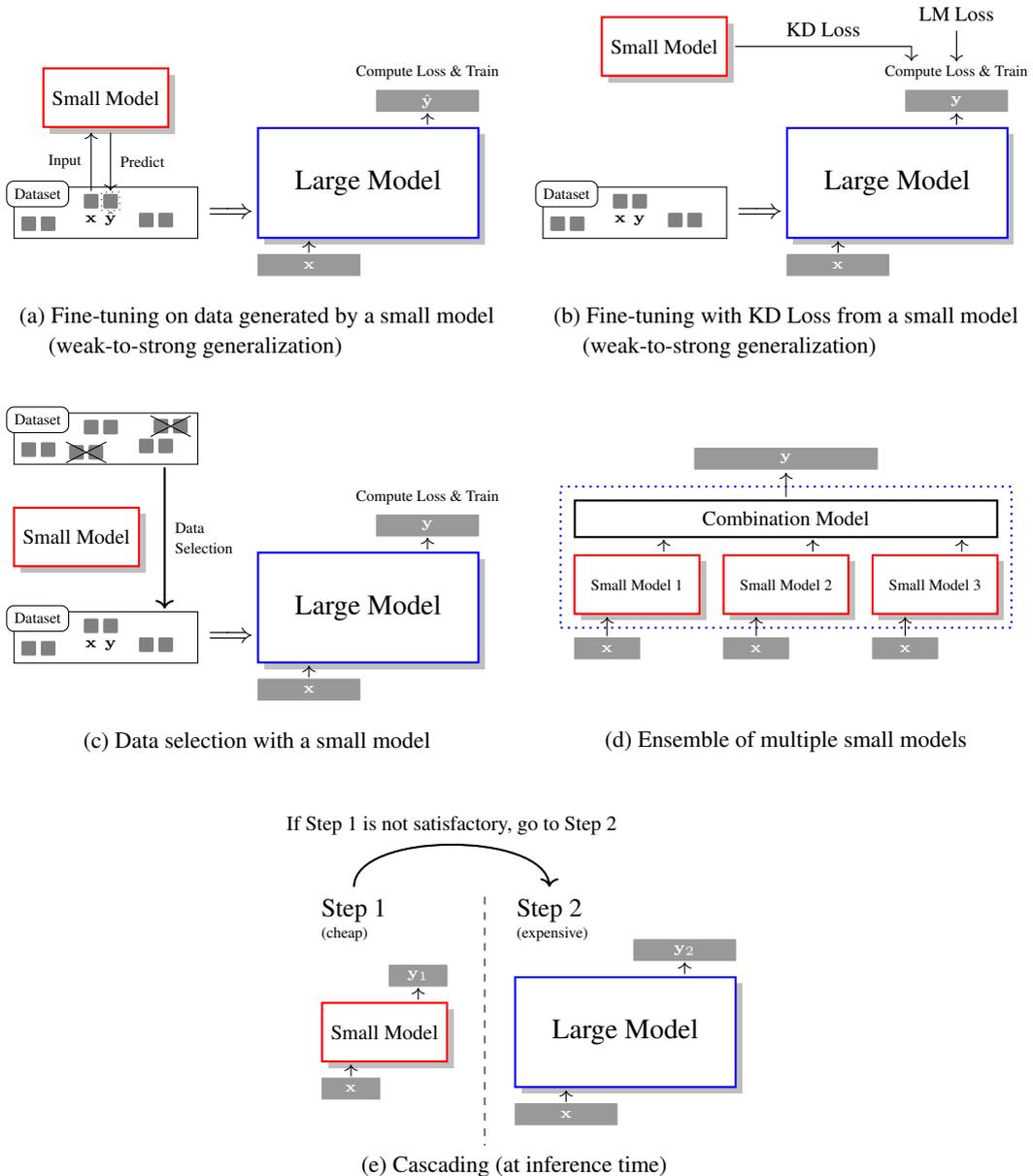


Figure 10.5: Illustrations of using small models to improve large models in LLMs. One approach involves using smaller models for the fine-tuning or pre-training of larger models. This includes generating synthetic data (a), incorporating auxiliary loss (b), and selecting appropriate data (c). Another approach involves combining small models and large models. This includes learning a strong model by aggregating multiple small models (d), and cascading small models with large models (e).

multiple weak models. Applying this technique to LLMs is straightforward. We can aggregate distributions predicted by multiple small models or specialized submodels,

and derive the final prediction from the aggregated results. This aggregation can be done using methods such as majority voting, weighted averaging, or stacking.

- Small models can also be employed at inference time to improve overall efficiency. Suppose we have a large model that is slow but more accurate, and a small model that is fast but less accurate. In model cascading, the small model first processes the input data, quickly generating preliminary results. If these results meet certain pre-defined criteria, they can be directly used. However, if the initial results are not sufficiently good, the input is then passed to the larger, more accurate model to produce a better result. This approach significantly reduces computational costs and latency, as the small model can effectively handle many inputs without access to the large model.

## 10.3 Human Preference Alignment: RLHF

So far in this chapter, we have focused on fine-tuning LLMs using input-output paired labeled data. This approach allows us to adapt LLMs for instruction-following via supervised learning. In many applications, however, LLMs are required not only to follow instructions but also to act in ways that are more aligned with human values and preferences. Consider a scenario where a user asks an LLM how to hack into a computer system. If the LLM is not appropriately aligned, it may respond by providing details on how to perform this illegal activity. Instead, a more desirable response might be to advise the user against engaging in illegal activities and offer a general overview of the consequences of such actions. The difficulty in achieving this is that the ethical nuances and contextual considerations required for an LLM to respond appropriately in such scenarios are not always straightforward to encode into a fine-tuning dataset. What's even more challenging is that, often, humans themselves cannot precisely express their own preferences.

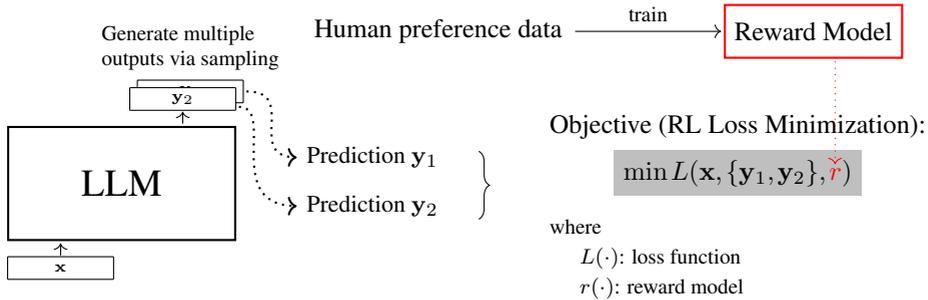
In this section, we discuss an alternative LLM fine-tuning method, called reinforcement learning from human feedback or RLHF for short [Christiano et al., 2017; Stiennon et al., 2020]. The basic idea behind RLHF is that LLMs can learn from comparisons of model outputs using reward models (see Figure 10.6). To do this, we can recruit human experts who indicate their preferences between pairs of outputs generated by the LLM. This preference data is used to train a reward model that can predict the perceived quality of LLM outputs. Once trained, the reward model provides feedback by assigning scores to new outputs that the LLM generates in response to the inputs. The LLM uses these scores to update its parameters through reinforcement learning algorithms. In the rest of this section, we will first introduce the basic knowledge of reinforcement learning to facilitate the discussion, and then discuss methods for training reward models and aligning LLMs with these models.

### 10.3.1 Basics of Reinforcement Learning

We begin by looking at some basic concepts of reinforcement learning. Note that the notation used here slightly differs from that used in the previous sections and chapters because we want to make our description more consistent with those in the reinforcement learning literature.



(a) Supervised fine-tuning (maximizing the prediction probability given the input)



(b) Reinforcement Learning from Human Feedback

Figure 10.6: Supervised fine-tuning vs. reinforcement learning from human feedback. In supervised fine-tuning, we optimize the LLM by maximizing the probability of the prediction given the input. In reinforcement learning from human feedback, we first train a reward model on human preference data (on each pair of predictions, evaluators are asked to choose which one they prefer). Then, we use this reward model to supervise the LLM during fine-tuning.

Nevertheless, we will show how this notation corresponds to the language modeling notation. The reader who is already familiar with reinforcement learning techniques may skip or skim this subsection.

A general reinforcement learning framework describes how an agent interacts with a dynamic environment. This interaction is modeled as a sequence of actions taken by the agent in response to the state of the environment. At each time step, the agent observes the current state, chooses an action based on its policy, performs the action, and then receives feedback from the environment in the form of a reward and a new state. This sequence of observe-act-receive feedback is repeated until the agent achieves its goal.

A reinforcement learning system involves several components:

- **Agent.** This is the learner or decision-maker in reinforcement learning. In the context of LLMs, it can be seen as the LLM itself.
- **Environment.** This includes everything external to the agent with which the agent interacts. But the environment in LLMs is less about a physical or virtual space and more about the framework within which the agent (e.g., an LLM) receives feedback and

learns.

- **State** ( $s$ ). A state represents the current situation of the environment. Given a sequence of tokens for language modeling, a state at a time step can be viewed as the tokens we observed so far, that is, the context tokens we take to predict the next token. For example, we can define  $(\mathbf{x}, \mathbf{y}_{<t})$  as the state when predicting the next token at the time step  $t$ .
- **Action** ( $a$ ). Actions represent possible decisions the agent can make. We can see them as possible predicted tokens in the vocabulary.
- **Reward** ( $R$ ). The reward is the feedback from the environment that evaluates the success of an action. For example,  $r(s, a, s')$  denotes the reward the agent receives for taking the action  $a$  at the state  $s$  and moving to the next state  $s'$ . If the state-action sequence is given, we can denote the reward at the time step  $t$  as  $r_t = r(s_t, a_t, s_{t+1})$ . Also note that if the decision-making process is deterministic, we can omit  $s_{t+1}$  because it can be determined by  $s_t$  and  $a_t$ . In such cases, we can use  $r(s_t, a_t)$  as shorthand for  $r(s_t, a_t, s_{t+1})$ .
- **Policy** ( $\pi$ ). For an LLM, a policy is defined as the probability distribution over the tokens that the LLM predicts, given the preceding context tokens. Formally, this can be expressed as

$$\pi(a|s) = \Pr(y_t|\mathbf{x}, \mathbf{y}_{<t}) \quad (10.14)$$

where  $a$  corresponds to the token  $y_t$ , and  $s$  corresponds to the context  $(\mathbf{x}, \mathbf{y}_{<t})$ . Figure 10.7 illustrates how an LLM can be treated as a policy in the reinforcement learning framework.

- **Value Function** ( $V$  and  $Q$ ). A **state-value function** (or value function, for short) assesses the expected discounted return (i.e., accumulated rewards) for an agent starting from a particular state  $s$  and following a specific policy  $\pi$ . It is defined as:

$$\begin{aligned} V(s) &= \mathbb{E}\left[r(s_0, a_0, s_1) + \gamma r(s_1, a_1, s_2) + \gamma^2 r(s_2, a_2, s_3) + \dots \mid s_0 = s, \pi\right] \\ &= \mathbb{E}\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \mid s_0 = s, \pi\right] \\ &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi\right] \end{aligned} \quad (10.15)$$

where  $\gamma \in [0, 1]$  is the discount factor that adjusts the importance of future rewards,  $s_0 = s$  indicates that the agent starts with the state  $s$ , and the expectation  $\mathbb{E}$  is performed over all possible trajectories (i.e., state-action sequences). Similarly, an **action-value function** (or **Q-value function**) measures the expected return starting from a state  $s$  taking an action  $a$  and thereafter following a policy  $\pi$ , given by

$$Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right] \quad (10.16)$$

where  $a_0 = a$  indicates that the action taken at the initial state is  $a$ .

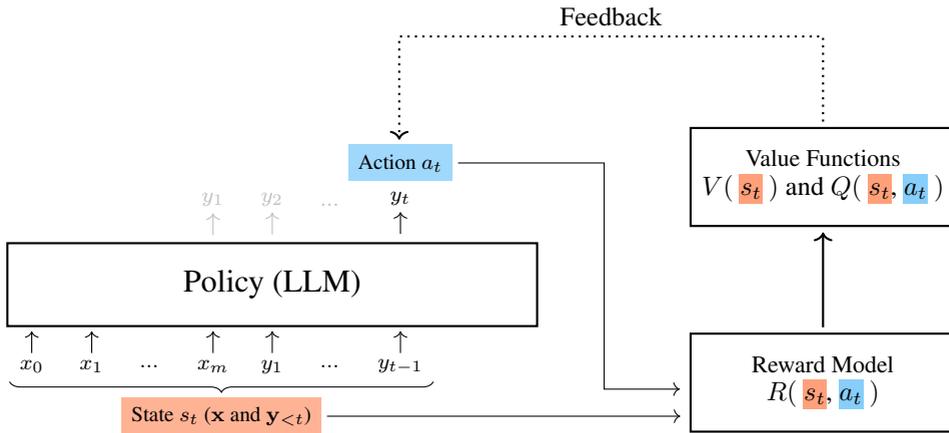


Figure 10.7: LLM as policy in reinforcement learning. At each step  $t$ , the LLM predicts a token  $y_t$  given the model input  $\mathbf{x}$  and the previously-generated tokens  $\mathbf{y}_{<t}$ . This process can be framed as a reinforcement learning problem, where  $y_t$  serves as the action,  $(\mathbf{x}, \mathbf{y}_{<t})$  as the state, and the predicted distribution  $\Pr(y_t|\mathbf{x}, \mathbf{y}_{<t})$  as the policy. Once  $y_t$  is predicted, the LLM inputs both  $(\mathbf{x}, \mathbf{y}_{<t})$  and  $y_t$  to the reward model, which evaluates how effectively the chosen token contributes to achieving the desired textual outcome. This evaluation generates reward scores which are used to compute the value functions  $V(s_t)$  and  $Q(s_t, a_t)$ . These functions then provide feedback to the LLM and guide the policy training.

The goal of reinforcement learning is to learn a policy that maximizes the **cumulative reward** (or **return**) the agent receives over the long run. Given a state-action sequence  $\tau = \{(s_1, a_1), \dots, (s_T, a_T)\}$ <sup>1</sup>, the cumulative reward over this sequence can be written as

$$R(\tau) = \sum_{t=1}^T r_t \quad (10.17)$$

The expectation of this cumulative reward over a space of state-action sequences is given in the form

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim \mathcal{D}} [R(\tau) | \pi_\theta] \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \sum_{t=1}^T r_t \end{aligned} \quad (10.18)$$

where  $\tau \sim \mathcal{D}$  indicates that  $\tau$  is drawn from the state-action sequence space  $\mathcal{D}$ , and the subscript

<sup>1</sup>We assume the state-action sequence begins with  $s_1$  and  $a_1$ , rather than  $s_0$  and  $a_0$ , to align with the notation commonly used in this chapter, where the prediction  $\mathbf{y}$  typically starts from  $y_1$ . Of course, it is also common to denote a state-action sequence as  $\{(s_0, a_0), \dots, (s_T, a_T)\}$  or  $\{(s_0, a_0), \dots, (s_{T-1}, a_{T-1})\}$  in the literature. But this variation in notation does not affect the discussion of the models presented here.

$\theta$  indicates the parameters of the policy.  $J(\theta)$  is also called the **performance function**.

Then the training objective is to maximize  $J(\theta)$ :

$$\tilde{\theta} = \arg \max_{\theta} J(\theta) \quad (10.19)$$

Now, we have a simple reinforcement learning approach: 1) we sample a number of state-action sequences; then, 2) we evaluate each sequence using the performance function; then, 3) we update the model to maximize this performance function. If we take Eq. (10.18) and use gradient descent to optimize the policy, this approach would constitute a form of policy gradient methods [Williams, 1992].

Note that in many NLP problems, such as machine translation, rewards are typically sparse. For instance, a reward is only received at the end of a complete sentence. This means that  $r_t = 0$  for all  $t < T$ , and  $r_t$  is non-zero only when  $t = T$ . Ideally, one might prefer feedback to be immediate and frequent (dense), and thus the training of the policy can be easier and more efficient. While several methods have been proposed to address sparse rewards, such as reward shaping, we will continue in our discussion to assume a sparse reward setup, where the reward is available only upon completing the prediction.

The model described in Eqs. (10.17-10.19) establishes a basic form of reinforcement learning, and many variants and improvements of this model have been developed. Before showing those more sophisticated models, let us take a moment to interpret the objective function  $J(\theta)$  from the perspective of policy gradient. In gradient descent, we need to compute the gradient of  $J(\theta)$  with respect to  $\theta$ :

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta} &= \frac{\partial \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) R(\tau)}{\partial \theta} \\ &= \sum_{\tau \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) \frac{\partial \Pr_{\theta}(\tau) / \partial \theta}{\Pr_{\theta}(\tau)} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) \frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \end{aligned} \quad (10.20)$$

In some cases, we will assume that every sequence in  $\mathcal{D}$  is equally probable (i.e.,  $\Pr_{\theta}(\tau) = 1/|\mathcal{D}|$ ). In this case we can simplify Eq. (10.20) and need only consider the terms  $\frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta}$  and  $R(\tau)$ :

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \sum_{\tau \in \mathcal{D}} \frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \quad (10.21)$$

One advantage of this result is that  $R(\tau)$  does not need to be differentiable, which means that we can use any type of reward function in reinforcement learning.

By treating the generation of the sequence  $\tau$  as a Markov decision process, we can further derive  $\frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta}$ , and obtain:

$$\begin{aligned} \frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} &= \frac{\partial}{\partial \theta} \log \prod_{t=1}^T \pi_{\theta}(a_t | s_t) \Pr(s_{t+1} | s_t, a_t) \\ &= \frac{\partial}{\partial \theta} \underbrace{\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t)}_{\text{policy}} + \frac{\partial}{\partial \theta} \underbrace{\sum_{t=1}^T \log \Pr(s_{t+1} | s_t, a_t)}_{\text{dynamics}} \end{aligned} \quad (10.22)$$

where the gradient is decomposed into two parts: the policy gradient and the dynamics gradient. The policy component,  $\log \pi_{\theta}(a_t | s_t)$ , determines the log-probability of taking action  $a_t$  given state  $s_t$ , and it is parameterized by  $\theta$ . The dynamics component,  $\log \Pr(s_{t+1} | s_t, a_t)$ , represents the log-probability of transitioning to state  $s_{t+1}$  from state  $s_t$  after taking action  $a_t$ . In typical reinforcement learning settings, the dynamics are not directly influenced by the policy parameters  $\theta$ , and thus, their derivatives are often zero. In this case, therefore, Eq. (10.22) can be simplified to:

$$\frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \quad (10.23)$$

In other words, we only concentrate on optimizing the policy without concerning ourselves with the underlying dynamics.

Substituting Eq. (10.23) into Eq. (10.21), and expanding  $R(\tau)$ , we then obtain

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left( \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \sum_{t=1}^T r_t \right) \quad (10.24)$$

While this policy gradient approach is straightforward, it suffers from the problem that the variance of the estimated gradients can be very high, making the learning process noisy and inefficient. One reason for this high variance problem is that rewards can vary greatly across different steps or scenarios. Imagine that in a sequence of action decisions, the reward model tends to assign small rewards to good actions (e.g.,  $R_t = 2$ ) and large penalties to poor actions (e.g.,  $R_t = -50$ ). Such varying reward scales for good and poor actions can result in a very low total reward for the entire sequence, even if it includes good actions.

One simple method for reducing the variance of the gradient is to set a baseline  $b$  and subtract it from  $\sum_{t=1}^T r_t$ , resulting in  $\sum_{t=1}^T r_t - b$ .<sup>2</sup> Here, the baseline can be interpreted as a reference point. By centering the rewards around this baseline, we remove systematic biases in

<sup>2</sup>In fact, the use of a baseline  $b$  does not change the variance of the total rewards  $\sum_{t=1}^T r_t$ . However, it is important to note that while introducing a baseline does not alter the overall variance of the rewards, it helps reduce the variance of the gradient estimates. This is because subtracting the baseline from the total rewards effectively reduces fluctuations around their mean, which makes the gradient estimates more stable. In general, the operation  $\sum_{t=1}^T r_t - b$  centers the rewards around zero (e.g.,  $b$  is defined as the expected value of  $\sum_{t=1}^T r_t$ ), which can lead to reduced variance in the product  $\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) (\sum_{t=1}^T r_t - b)$ .

the reward signal, making the updates more stable and less sensitive to extreme fluctuations in individual rewards.

This policy gradient model with a baseline can be given by

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left( \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \right) \left( \sum_{t=1}^T r_t - b \right) \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[ \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left( \sum_{k=1}^T r_k - b \right) \right] \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[ \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left( \sum_{k=1}^{t-1} r_k + \sum_{k=t}^T r_k - b \right) \right] \quad (10.25)
\end{aligned}$$

Here we write  $\sum_{k=1}^T r_k$  as the sum of two terms  $\sum_{k=1}^{t-1} r_k$  and  $\sum_{k=t}^T r_k$  to distinguish between the rewards accrued before and after the action at time step  $t$ . Note that in Markov decision processes, the future is independent of the past given the present. Therefore, the action taken at time step  $t$  cannot influence the rewards received before  $t$ , or in other words, the rewards prior to  $t$  are already “fixed” by the time the action at  $t$  is chosen. The term  $\sum_{k=1}^{t-1} r_k$  does not contribute to the gradient and can be omitted, leading to a simplified version of Eq. (10.25)

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[ \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left( \sum_{k=t}^T r_k - b \right) \right] \quad (10.26)$$

Also note that removing  $\sum_{k=t}^T r_k$  can further reduce the variance of the gradient.

There are many ways to define the baseline  $b$ . Here we consider the value function of the state  $s_t$ , that is, the estimated value of being in state  $s_t$ :  $V(s_t) = \mathbb{E}(r_t + r_{t+1} + \dots + r_T)$ . Hence we have

$$\begin{aligned}
A(s_t, a_t) &= \sum_{k=t}^T r_k - b \\
&= \sum_{k=t}^T r_k - V(s_t) \quad (10.27)
\end{aligned}$$

where  $\sum_{k=t}^T r_k$  represents the actual return received, and  $V(s_t)$  represents the expected return.  $A(s_t, a_t)$  (or  $A_t$  for short) is called the **advantage** at time step  $t$ , which quantifies the relative benefit of the action  $a_t$  compared to the expected value of following the policy from the state  $s_t$  onward.

By using the advantage function  $A(s_t, a_t)$ , the gradient of  $J(\theta)$  can be written in the form

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left( \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \right) \quad (10.28)$$

This optimization objective corresponds to the **advantage actor-critic (A2C)** method in reinforcement learning [Mnih et al., 2016]. In this method, the actor aims at learning a policy. It updates the policy parameters using Eq. (10.28) to help focus more on actions that are likely to improve performance. The critic, on the other hand, updates its estimation of the value function, which is used to calculate the advantage function  $A(s_t, a_t)$ , thus serving as the evaluator of the policy being learned by the actor.

In the A2C method,  $A(s_t, a_t)$  is typically expressed as the difference of the action-value function  $Q(s_t, a_t)$  and the state-value function  $V(s_t)$

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \quad (10.29)$$

At first glance, this model may seem challenging to develop because it requires two separate sub-models to calculate  $Q(s_t, a_t)$  and  $V(s_t)$  respectively. Fortunately, considering that  $Q(s_t, a_t)$  can be defined as the return  $r_t + V(s_{t+1})$ , we can rewrite Eq. (10.29) as

$$A(s_t, a_t) = r_t + V(s_{t+1}) - V(s_t) \quad (10.30)$$

or alternatively, introduce the discount factor  $\gamma$  to obtain a more general form

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (10.31)$$

$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$  is also called the **temporal difference (TD)** error. What we need is to train a critic network for the value function  $V(s_t)$ , and then use it to compute the advantage function<sup>3</sup>.

Up to this point, we have spent considerable space discussing the basics of reinforcement learning, especially on how to derive the optimization objective for the A2C method. However, reinforcement learning is a vast field, and many technical details cannot be covered here. The interested reader can refer to reinforcement learning books for more details [Sutton and Barto, 2018; Szepesvári, 2010]. Nevertheless, we now have the necessary knowledge to further discuss RLHF. In the subsequent subsections, we will return to the discussion on LLM alignment, demonstrating how to use the A2C method for aligning with human preferences.

### 10.3.2 Training Reward Models

We have shown that reward models play a very important role in the general reinforcement learning framework and form the basis for computing value functions. We now consider the problem of training these reward models.

In RLHF, a reward model is a neural network that maps a pair of input and output token

---

<sup>3</sup>The training loss for the value network (or critic network) in A2C is generally formulated as the mean squared error between the computed return  $r_t + \gamma V(s_{t+1})$  and the predicted state value  $V(s_t)$ . Suppose that the value network is parameterized by  $\omega$ . The loss function is given by

$$\mathcal{L}_v(\omega) = \frac{1}{M} \sum (r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t))^2 \quad (10.32)$$

where  $M$  is the number of training samples, for example, for a sequence of  $T$  tokens, we can set  $M = T$ .

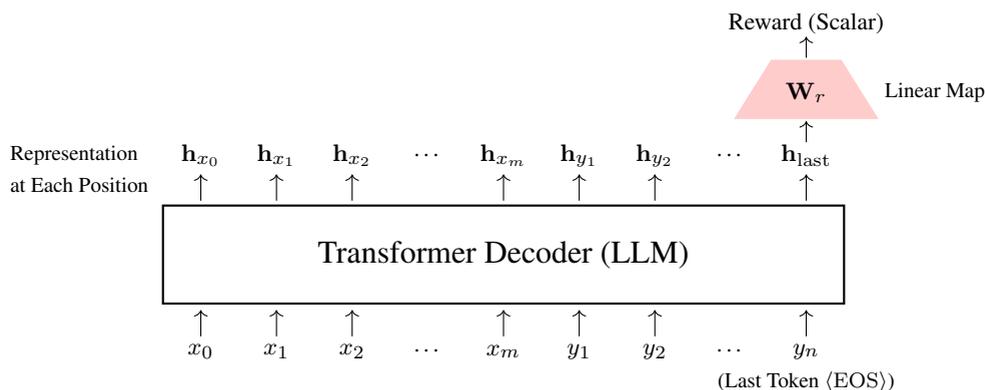


Figure 10.8: Architecture of the reward model based on Transformer. The main component of this model is still an LLM. We use the Transformer decoder as the sequence representation model. We extract the representation of the last position of the decoder as the representation of the entire sequence  $[\mathbf{x}, \mathbf{y}]$ . We then map this representation to a scalar through a linear transformation, which serves as the reward score for  $[\mathbf{x}, \mathbf{y}]$ .

sequences to a scalar. Given an input  $\mathbf{x}$  and an output  $\mathbf{y}$ , the reward can be expressed as

$$r = \text{Reward}(\mathbf{x}, \mathbf{y}) \quad (10.33)$$

where  $\text{Reward}(\cdot)$  is the reward model.  $r$  can be interpreted as a measure of how well the output  $\mathbf{y}$  aligns with the desired behavior given the input  $\mathbf{x}$ . As discussed in the previous subsection, both  $\mathbf{x}$  and  $\mathbf{y}$  are assumed to complete texts. This means that the reward model evaluates the relationship between inputs and outputs that provide full semantic content. For example, when applying the reward model, it assigns a value of 0 (or another predetermined value) at each position  $t$  in the output sequence  $\mathbf{y} = y_1 \dots y_n$ . Only at the final position, when  $t = n$ , does the reward model generate the actual reward score. To keep the notation uncluttered, we will use  $r(\mathbf{x}, \mathbf{y})$  to denote the reward model  $\text{Reward}(\mathbf{x}, \mathbf{y})$  from here on.

There are many ways to implement the reward model. One simple approach is to build the reward model based on a pre-trained LLM. More specifically, we can concatenate  $\mathbf{x}$  and  $\mathbf{y}$  to form a single token sequence  $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$ . We run a pre-trained LLM on this sequence, as usual, and at each position, we obtain a representation from the top-most Transformer layer. Then, we take the representation at the last position (denoted by  $\mathbf{h}_{last}$ ) and map it to a scalar via linear transformation:

$$r(\mathbf{x}, \mathbf{y}) = \mathbf{h}_{last} \mathbf{W}_r \quad (10.34)$$

where  $\mathbf{h}_{last}$  is a  $d$ -dimensional vector, and  $\mathbf{W}_r$  is a  $d \times 1$  linear mapping matrix. This architecture of the reward model is illustrated in Figure 10.8.

To train the reward model, the first step is to collect human feedback on a set of generated outputs. Given an input  $\mathbf{x}$ , we use the LLM to produce multiple candidate outputs  $\{\mathbf{y}_1, \dots, \mathbf{y}_N\}$ .

Human feedback can be obtained in several ways:

- **Pairwise Comparison (Pairwise Ranking).** Given two different outputs, human experts select which one is better.
- **Rating.** Human experts provide a score or rating to each output. This score is often a continuous or discrete numerical value, such as a score on a scale (e.g., 1-5 stars, or 1-10 points). In some cases, the rating might be binary, indicating a “yes/no” or “positive/negative” preference.
- **Listwise Ranking.** Human experts are asked to rank or order the given set of possible outputs.

Here we consider pairwise comparison feedback as it is one of the simplest and most common forms of human feedback used in RLHF. In this setting, each time, two outputs  $(\mathbf{y}_a, \mathbf{y}_b)$  are randomly drawn from the candidate pool  $\{\mathbf{y}_1, \dots, \mathbf{y}_N\}$ . Human experts are then presented with these pairs and asked to decide which output they prefer based on specific criteria, such as clarity, relevance, and accuracy. The human feedback can be encoded as a binary label,  $\mathbf{y}_a \succ \mathbf{y}_b$  for a preference for  $\mathbf{y}_a$ , and  $\mathbf{y}_b \succ \mathbf{y}_a$  for a preference for  $\mathbf{y}_b$ .

One simple and widely used model for describing such pairwise comparisons is the **Bradley-Terry model** [Bradley and Terry, 1952]. It is a probabilistic model that estimates the probability that one item is preferred over another. Adapting this model to the notation used here, we can write the probability that  $\mathbf{y}_a$  is preferred over  $\mathbf{y}_b$  in the form

$$\begin{aligned} \Pr(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) &= \frac{e^{r(\mathbf{x}, \mathbf{y}_a)}}{e^{r(\mathbf{x}, \mathbf{y}_a)} + e^{r(\mathbf{x}, \mathbf{y}_b)}} \\ &= \frac{e^{r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)}}{e^{r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)} + 1} \\ &= \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)) \end{aligned} \quad (10.35)$$

When training the reward model, we want to maximize this preference probability. A loss function based on the Bradley-Terry model is given by

$$\mathcal{L}_r(\phi) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \quad (10.36)$$

where  $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$  is drawn from a human-annotated dataset  $\mathcal{D}_r$  consisting of preference pairs of outputs and their corresponding inputs.  $\phi$  represents the parameters of the reward model, which includes both the parameters of the Transformer decoder and the linear mapping matrix  $\mathbf{W}_r$ . In practice, assuming  $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$  is uniformly sampled from  $\mathcal{D}_r$ , we can replace the expectation with a summation

$$\mathcal{L}_r(\phi) = -\frac{1}{|\mathcal{D}_r|} \sum_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \in \mathcal{D}_r} \log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) \quad (10.37)$$

The goal of training the reward model is to find the optimal parameters  $\hat{\phi}$  that minimize

this loss function, given by

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}} \mathcal{L}_r(\phi) \quad (10.38)$$

Since the reward model itself is also an LLM, we can directly reuse the Transformer training procedure to optimize the reward model. The difference from training a standard LLM is that we only need to replace the cross-entropy loss with the pairwise comparison loss as described in Eq. (10.37). After the training of the reward model, we can apply the trained reward model  $r_{\hat{\phi}}(\cdot)$  to supervise the target LLM for alignment.

It is worth noting that although we train the reward model to perform pairwise ranking, we apply it to score each input-output pair independently during the alignment process. The pairwise ranking objective ensures that the reward model is sensitive to subtle differences between outputs, but we rely on the continuous scores produced by the reward model to guide the optimization of the LLM. An advantage of this approach is that we can choose from or combine various ranking loss functions, and still apply the resulting reward models in the same way as we have done in this subsection. This consistency ensures a unified framework for aligning the LLM, regardless of the specific ranking loss used during reward model training.

### 10.3.3 Training LLMs

Having obtained the reward model, we then train the policy (i.e., the LLM) via the A2C method. Recall from Section 10.3.1 that a state-action sequence or trajectory  $\tau$  can be evaluated by the utility function

$$U(\tau; \theta) = \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \quad (10.39)$$

where  $A(s_t, a_t)$  is the advantage of taking the action  $a_t$  given the state  $s_t$ . An estimate of  $A(s_t, a_t)$  is defined as the TD error  $r_t + \gamma V(s_{t+1}) - V(s_t)$ , where the value function  $V(s_t)$  is trained with the reward model.

Given this utility function, the A2C-based loss function can be written in the form

$$\begin{aligned} \mathcal{L}(\theta) &= -\mathbb{E}_{\tau \sim \mathcal{D}} [U(\tau; \theta)] \\ &= -\mathbb{E}_{\tau \sim \mathcal{D}} \left[ \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \right] \end{aligned} \quad (10.40)$$

where  $\mathcal{D}$  is a space of state-action sequences. As usual, the goal of training the policy is to minimize this loss function

$$\tilde{\theta} = \underset{\theta}{\operatorname{argmin}} \mathcal{L}(\theta) \quad (10.41)$$

If we map the problem back to the language modeling problem and adopt the notation

from LLMs, the loss function can be written as:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}} [U(\mathbf{x}, \mathbf{y}; \theta)] \quad (10.42)$$

where

$$U(\mathbf{x}, \mathbf{y}; \theta) = \sum_{t=1}^T \log \pi_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (10.43)$$

Here  $\pi_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) = \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})$  is the LLM parameterized by  $\theta$ .

In general, we do not have a human annotated input-output dataset  $\mathcal{D}$  in RLHF, but a dataset containing inputs only. The outputs, in this case, are typically the predictions made by the LLM. The loss function is then defined as

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot | \mathbf{x})} [U(\mathbf{x}, \mathbf{y}; \theta)] \quad (10.44)$$

where  $\mathcal{D}$  denotes the input-only dataset, and  $\mathbf{y} \sim \pi_{\theta}(\cdot | \mathbf{x})$  denotes that the output  $\mathbf{y}$  is sampled by the policy  $\pi_{\theta}(\cdot | \mathbf{x})$ .

The above formulation provides a basic form of the A2C method for LLMs. Improved versions of this model are more commonly used in RLHF. In the following discussion, we will still use the reinforcement learning notation to simplify the presentation and will get back the language modeling notation later.

One common improvement of policy gradient methods is to use **importance sampling** to refine the estimation of  $U(\tau; \theta)$ . This can be written as

$$U(\tau; \theta) = \sum_{t=1}^T \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{ref}}}(a_t | s_t)} A(s_t, a_t) \quad (10.45)$$

Here we replace the log-probability  $\log \pi_{\theta}(a_t | s_t)$  with the ratio  $\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{ref}}}(a_t | s_t)}$ .  $\theta_{\text{ref}}$  denotes the parameters of the previous policy (such as an initial model from which we start the training). So  $\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{ref}}}(a_t | s_t)}$ , also called the **ratio function**, can be interpreted as the log-probability ratio between the current policy  $\pi_{\theta}$  and the previous policy  $\pi_{\theta_{\text{ref}}}$  (call it the reference policy). By using the ratio function we reweight the observed rewards based on the likelihood of the actions under the current policy versus the reference policy. When  $\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{ref}}}(a_t | s_t)} > 1$ , the action  $a_t$  is more favored by the current policy compared to the reference policy. By contrast, when  $\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{ref}}}(a_t | s_t)} < 1$ , the action  $a_t$  is less favored by the current policy<sup>4</sup>.

A problem with the model presented in Eq. (10.47) (as well as in Eq. (10.39)) is that the variance in the gradient estimates is often high, making the learning process unstable. To

<sup>4</sup>Consider a more general case where we wish to evaluate the policy using its expected reward (also see Eq. (10.18))

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)] \quad (10.46)$$

Here  $\tau \sim \pi_{\theta}$  means that the sequence  $\tau$  is generated by the policy  $\pi_{\theta}$ . Alternatively, we can write  $J(\theta)$  in another

mitigate this issue, techniques such as clipping are often employed to bound the importance weights and prevent large updates. A clipped version of the utility function (also called the clipped surrogate objective function) is given by

$$U_{\text{clip}}(\tau; \theta) = \sum_{t=1}^T \text{Clip}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) A(s_t, a_t) \quad (10.49)$$

$$\text{Clip}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) = \min\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, \text{bound}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon\right)\right) \quad (10.50)$$

Here the function  $\text{bound}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon\right)$  constrains the ratio function to the range  $[1 - \epsilon, 1 + \epsilon]$ .

A further improvement to the above model is to consider **trust regions** in optimization [Schulman et al., 2015]. In reinforcement learning, a large update to the policy can lead to instability, where the agent may start performing worse after an update. A reasonable idea is to optimize the model in the trust region, which refers to a region around the current parameter estimate where the model is well-behaved. One approach to incorporating trust regions is to impose a constraint on the size of the policy update, ensuring that the current policy does not deviate too significantly from the reference policy. This can be achieved by adding a penalty based on some form of divergence between the current and reference policies to the objective function. A simple form of such a penalty is given by the difference in the log-probability of the sequence  $\tau$  under the current policy versus the reference policy:

$$\text{Penalty} = \log \pi_{\theta}(\tau) - \log \pi_{\theta_{\text{ref}}}(\tau) \quad (10.51)$$

---

form

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] \quad (10.47)$$

It is not difficult to find that the right-hand sides of these equations are essentially the same since  $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] = \sum_{\tau} \text{Pr}_{\theta_{\text{ref}}}(\tau) \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) = \sum_{\tau} \text{Pr}_{\theta}(\tau) R(\tau) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)]$

Note that this equivalence holds only when the expectation is performed over the entire sequence space. In practice, however, we often only sample a relatively small number of sequences using a policy in policy learning. As a result, the sampling method itself matters. Eq. (10.47) offers an interesting manner to separate the sampling and reward computation processes: we first use a baseline policy (with  $\theta_{\text{ref}}$ ) to sample a number of sequences, and then use the target policy (with  $\theta$ ) to compute the expected reward. In this way, we separate the policy used for collecting the data, and the policy used for computing the gradient. This approach avoids the need to directly sample from the policy we are evaluating, which can be beneficial in cases where generating sequences from the target policy is expensive or difficult. In reinforcement learning,  $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) \right]$  is often called a **surrogate objective**.

Eq. (10.47) can also be interpreted from a policy gradient perspective. For  $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) \right]$ , the gradient at  $\theta = \theta_{\text{ref}}$  is given by

$$\frac{\partial}{\partial \theta} \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\text{Pr}_{\theta}(\tau)}{\text{Pr}_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] \Big|_{\theta = \theta_{\text{ref}}} = \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[ \frac{\partial \text{Pr}_{\theta}(\tau) |_{\theta = \theta_{\text{ref}}}}{\partial \theta} R(\tau) \right] \quad (10.48)$$

The right-hand side is a standard form used in policy gradient methods, meaning that we compute the direction of the parameter update at the point  $\theta = \theta_{\text{ref}}$  on the optimization surface.

In practice, this penalty can be approximated by considering only the policy probabilities and ignoring the dynamics. This gives

$$\text{Penalty} = \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) - \sum_{t=1}^T \log \pi_{\theta_{\text{ref}}}(a_t | s_t) \quad (10.52)$$

By including this penalty in the optimization objective, we encourage the current policy to remain close to the reference policy, limiting very large updates that could destabilize learning.

We can incorporate this penalty into the clipped surrogate objective function, and obtain

$$U_{\text{ppo-clip}}(\tau; \theta) = U_{\text{clip}}(\tau; \theta) - \beta \text{Penalty} \quad (10.53)$$

where  $\beta$  is the weight of the penalty. This training method is called **proximal policy optimization (PPO)**, which is one of the most popular reinforcement learning methods used in LLMs and many other fields [Schulman et al., 2017].

Now we can write the objective of training LLMs in the form of PPO.

$$U(\mathbf{x}, \mathbf{y}; \theta) = U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta) - \beta \text{Penalty} \quad (10.54)$$

where

$$U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta) = \sum_{t=1}^T \text{Clip} \left( \frac{\pi_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\pi_{\theta_{\text{ref}}}(y_t | \mathbf{x}, \mathbf{y}_{<t})} \right) A(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (10.55)$$

$$\begin{aligned} \text{Penalty} &= \log \Pr_{\theta}(\mathbf{y} | \mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y} | \mathbf{x}) \\ &= \sum_{t=1}^T \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) - \sum_{t=1}^T \log \Pr_{\theta_{\text{ref}}}(y_t | \mathbf{x}, \mathbf{y}_{<t}) \end{aligned} \quad (10.56)$$

Although the notation here appears a bit tedious, the idea of PPO is simple: we develop an objective by combining the clipped likelihood ratio of the target and reference policies with an advantage function, and then impose a penalty that ensures policy updates are not too large. The PPO-based RLHF is illustrated in Figure 10.9.

To summarize, implementing RLHF requires building four models, all based on the Transformer decoder architecture.

- **Reward Model** ( $r_{\phi}(\cdot)$  where  $\phi$  denotes the parameters). The reward model learns from human preference data to predict the reward for each pair of input and output token sequences. It is a Transformer decoder followed by a linear layer that maps a sequence (the concatenation of the input and output) to a real-valued reward score.
- **Value Model** or **Value Function** ( $V_{\omega}(\cdot)$  where  $\omega$  denotes the parameters). The value function receives reward scores from the reward model and is trained to predict the expected sum of rewards that can be obtained starting from a state. It is generally based on the same architecture as the reward model.

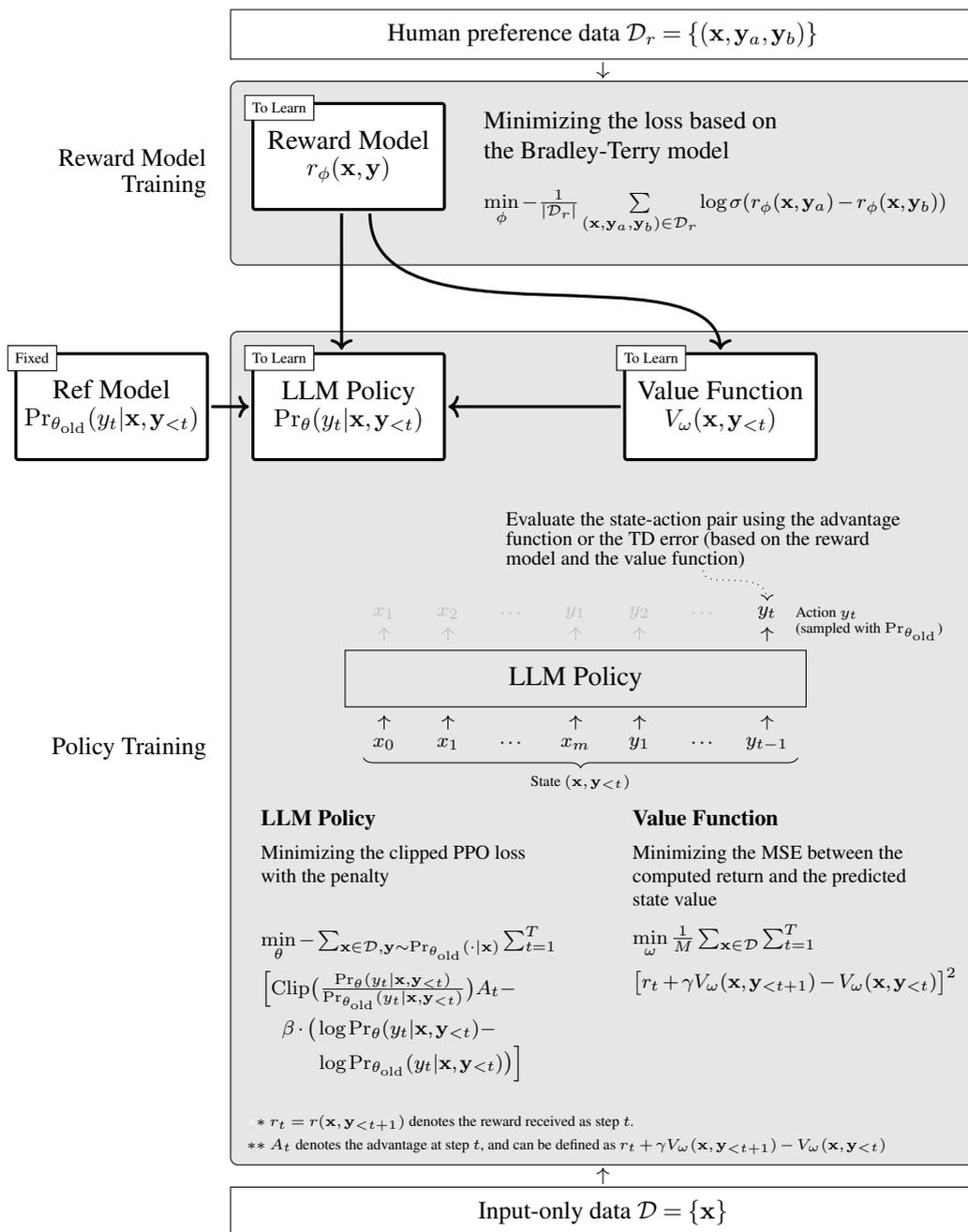


Figure 10.9: Illustration of RLHF. The first step is to collect human preference data and train the reward model using this data. Once the reward model is optimized, along with the reference model, we proceed to train both the policy and the value function. At each prediction step, we compute the sum of the PPO-based loss and update the parameters of the policy. This requires access to the reward model, the reference model, and the value function at hand. At the same time, we update the parameters of the value function by minimizing the MSE loss.

- **Reference Model** ( $\pi_{\theta_{\text{ref}}}(\cdot) = \text{Pr}_{\theta_{\text{ref}}}(\cdot)$  where  $\theta_{\text{ref}}$  denotes the parameters). The reference model is the baseline LLM that serves as a starting point for policy training. In RLHF, it represents the previous version of the model or a model trained without human feedback. It is used to perform sampling over the space of outputs and contribute to the loss computation for policy training.
- **Target Model or Policy** ( $\pi_{\theta}(\cdot) = \text{Pr}_{\theta}(\cdot)$  where  $\theta$  denotes the parameters). This policy governs how the LLM decides the most appropriate next token given its context. It is trained under the supervision of both the reward model and the value model.

In practice, these models need to be trained in a certain order. First, we need to initialize them using some other models. For example, the reward model and the value model can be initialized with a pre-trained LLM, while the reference model and the target model can be initialized with a model that has been instruction fine-tuned. Note that, at this point, the reference model is ready for use and will not be further updated. Second, we need to collect human preference data and train the reward model on this data. Third, both the value model and the policy are trained simultaneously using the reward model. At each position in an output token sequence, we update the value model by minimizing the MSE error of value prediction, and the policy is updated by minimizing the PPO loss.

## 10.4 Improved Human Preference Alignment

In the previous section, we reviewed the basic concepts of reinforcement learning and the general framework of RLHF. In this section, we will discuss some refinements of RLHF and alternative methods to achieve human preference alignment.

### 10.4.1 Better Reward Modeling

In Section 10.3.2, we highlighted the task of learning from human preferences as well as the use of pairwise ranking loss for training reward models. Here we consider more methods for reward modeling. Our discussion will be relatively general, and since the reward model is widely used in many reinforcement learning problems, it will be easy for us to apply the methods discussed here to RLHF and related applications.

#### 1. Supervision Signals

The training of reward models can broadly be seen as a ranking problem, where the model learns to assign scores to outputs so that their order reflects the preferences indicated by humans. There are several methods to train a reward model from the perspective of ranking.

One approach is to extend pairwise ranking to listwise ranking. For each sample in a dataset, we can use the LLM to generate multiple outputs, and ask human experts to order these outputs. For example, given a set of four outputs  $\{y_1, y_2, y_3, y_4\}$ , one possible order of them can be  $y_2 \succ y_3 \succ y_1 \succ y_4$ . A very simple method to model the ordering of the list is to accumulate the pairwise comparison loss. For example, we can define the listwise loss by

accumulating the loss over all pairs of outputs:

$$\mathcal{L}_{\text{list}} = -\mathbb{E}_{(\mathbf{x}, Y) \sim \mathcal{D}_r} \left[ \frac{1}{N(N-1)} \sum_{\substack{\mathbf{y}_a \in Y, \mathbf{y}_b \in Y \\ \mathbf{y}_a \neq \mathbf{y}_b}} \log \Pr(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) \right] \quad (10.57)$$

where  $Y$  is a list of outputs, and  $N$  is the number of outputs in the list.  $\Pr(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})$  can be defined using the Bradley-Terry model, that is,  $\Pr(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) = \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b))$ . Here we omit the  $\phi$  superscript on the  $\Pr(\cdot)$  to keep the notation uncluttered.

An extension to the Bradley-Terry model for listwise ranking could involve a ranking mechanism that takes into account the entire list of outputs rather than just pairwise comparisons. One such model is the **Plackett-Luce model**, which generalizes the Bradley-Terry model to handle multiple items in a ranking [Plackett, 1975]. In the Plackett-Luce model, for each item in a list, we define a “worth” for this item that reflects its relative strength of being chosen over other items. For the reward modeling problem here, the worth of  $\mathbf{y}$  in the list  $Y$  can be defined as

$$\alpha(\mathbf{y}) = \exp(r(\mathbf{x}, \mathbf{y})) \quad (10.58)$$

Then the probability of selecting  $\mathbf{y}$  from  $Y$  is given by

$$\begin{aligned} \Pr(\mathbf{y} \text{ is selected} | \mathbf{x}, Y) &= \frac{\alpha(\mathbf{y})}{\sum_{\mathbf{y}' \in Y} \alpha(\mathbf{y}')} \\ &= \frac{\exp(r(\mathbf{x}, \mathbf{y}))}{\sum_{\mathbf{y}' \in Y} \exp(r(\mathbf{x}, \mathbf{y}'))} \end{aligned} \quad (10.59)$$

Suppose  $\mathring{Y}$  is an ordered list  $\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \dots \succ \mathbf{y}_{j_N}$ . The overall log-probability of this ordered list can be defined as the sum of the conditional log-probabilities at each stage of selection, given by

$$\begin{aligned} \log \Pr(\mathring{Y} | \mathbf{x}) &= \log \Pr(\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \dots \succ \mathbf{y}_{j_N} | \mathbf{x}) \\ &= \log \Pr(\mathbf{y}_{j_1} | \mathbf{x}, \{\mathbf{y}_{j_1}, \mathbf{y}_{j_2}, \dots, \mathbf{y}_{j_N}\}) + \\ &\quad \log \Pr(\mathbf{y}_{j_2} | \mathbf{x}, \{\mathbf{y}_{j_2}, \dots, \mathbf{y}_{j_N}\}) + \\ &\quad \dots + \\ &\quad \log \Pr(\mathbf{y}_{j_N} | \mathbf{x}, \{\mathbf{y}_{j_N}\}) \\ &= \sum_{k=1}^N \log \Pr(\mathbf{y}_{j_k} | \mathbf{x}, \mathring{Y}_{\geq k}) \end{aligned} \quad (10.60)$$

where  $\mathring{Y}_{\geq k}$  represents the subset of the list of outputs that remain unselected at the  $k$ -th stage, i.e.,  $\mathring{Y}_{\geq k} = \{\mathbf{y}_{j_k}, \dots, \mathbf{y}_{j_N}\}$ . Given the log-probability  $\log \Pr(\mathring{Y} | \mathbf{x})$ , we can define the loss function based on the Plackett-Luce model by

$$\mathcal{L}_{\text{pl}} = -\mathbb{E}_{(\mathbf{x}, \mathring{Y}) \sim \mathcal{D}_r} [\log \Pr(\mathring{Y} | \mathbf{x})] \quad (10.61)$$

There are also many other pairwise and listwise methods for modeling rankings, such as RankNet [Burges et al., 2005] and ListNet [Cao et al., 2007]. All these methods can be categorized into a large family of learning-to-rank approaches, and most of them are applicable to the problem of modeling human preferences. However, discussing these methods is beyond the scope of this chapter. Interested readers can refer to books on this topic for more details [Liu, 2009; Li, 2011].

In addition to pairwise and listwise ranking, using pointwise methods to train reward models offers an alternative way to capture human preferences. Unlike methods that focus on the relative rankings between different outputs, pointwise methods treat each output independently. For example, human experts might assign a score to an individual output, such as a rating on a five-point scale. The objective is to adjust the reward model so that its outputs align with these scores. A simple way to achieve pointwise training is through regression techniques where the reward of each output is treated as a target variable. Let  $\varphi(\mathbf{x}, \mathbf{y})$  be the score assigned to  $\mathbf{y}$  given  $\mathbf{x}$  by humans. Pointwise reward models can be trained by minimizing a loss function, often based on mean squared error or other regression losses, between the predicted reward  $r(\mathbf{x}, \mathbf{y})$  and the actual human feedback  $\varphi(\mathbf{x}, \mathbf{y})$ . For example, the loss function could be

$$\mathcal{L}_{\text{point}} = -\mathbb{E}[\varphi(\mathbf{x}, \mathbf{y}) - r(\mathbf{x}, \mathbf{y})]^2 \quad (10.62)$$

While pointwise methods are conceptually simpler and can directly guide the reward model to predict scores, they might not always be the best choice in RLHF. A problem is that these methods may struggle with high variance in human feedback, especially when different experts provide inconsistent scores for similar outputs. Because they focus on fitting to absolute scores rather than relative differences, inconsistencies in scoring can lead to poor model performance. Moreover, fitting to specific scored outputs might discourage generalization, particularly given that training data is often very limited in RLHF. In contrast, methods that consider relative preferences can promote the learning of more generalized patterns of success and failure. Nevertheless, there are scenarios where pointwise methods might still be suitable. For example, in tasks where training data is abundant and the costs of obtaining accurate, consistent annotations are low, pointwise methods can prove effective.

In fact, to make the supervision signal for training the reward model more robust, we can also introduce additional regularization terms into training. For example, if we consider the first term  $U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta)$  in Eq. (10.54) as a type of generalized reward, then the second term (i.e., the penalty term) can be viewed as a form of regularization for the reward model, except that here the goal is to train the policy rather than the reward model. Another example is that Eisenstein et al. [2023] develop a regularization term based on the squared sum of rewards, and add it to the pairwise comparison loss in RLHF:

$$\begin{aligned} \mathcal{L}_{\text{reg}} &= \mathcal{L}_{\text{pair}} + (-\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)]^2) \\ &= -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \text{Pr}_{\phi}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \\ &\quad - \mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)]^2 \end{aligned} \quad (10.63)$$

Optimizing with this regularization term can help mitigate the underdetermination of reward models<sup>5</sup>.

## 2. Sparse Rewards vs. Dense Rewards

As discussed in Section 10.3, the rewards in RLHF are very sparse: they are observed only at the end of sequences, rather than continuously throughout the generation process. Dealing with sparse rewards has long been a concern in reinforcement learning, and has been one of the challenges in many practical applications. For example, in robotics, it often needs to shape the reward function to ease optimization rather than relying solely on end-of-sequence rewards. Various methods have been developed to address this issue. One common approach is reward shaping, where the original function is modified to include intermediate rewards, thereby providing more immediate feedback. Also, one can adopt curriculum learning to sequentially structure tasks in a way that the complexity gradually increases. This can help models to master simpler tasks first, which prepares them for more complex challenges as their skills develop. There are many such methods that can mitigate the impact of sparse rewards, such as Monte Carlo methods and intrinsic motivation. Most of these methods are general and the discussion of them can be found in the broader literature on reinforcement learning, such as Sutton and Barto [2018]’s book.

Although we do not discuss methods for mitigating sparse rewards in detail here, an interesting question arises: why are sparse rewards so successful in RLHF? Recall from Section 10.3.1 that the supervision signal received at each time step  $t$  is not the reward for the current action, but rather some form of the accumulated rewards from  $t$  until the last time step. Such supervision signals are dense over the sequence, because the reward obtained at the end of the sequence can be transferred back to that time step, regardless of which time step it is. In other words, the sparse rewards are transformed into the dense supervision signals. Furthermore, from the perspective of reward shaping, Ng et al. [1999] show that the reward at  $t$  can be defined as

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t, s_{t+1}) + f(s_t, a_t, s_{t+1}) \quad (10.64)$$

where  $r'(\cdot)$  is the transformed reward function,  $r(\cdot)$  is the original reward function, and  $f(\cdot)$  is the shaping reward function. To ensure the optimality of the policy under the transformed reward function, the shaping reward function can be given in the form

$$f(s_t, a_t, s_{t+1}) = \gamma\Phi(s_{t+1}) - \Phi(s_t) \quad (10.65)$$

where  $\Phi(s)$  is called the potential value of the state  $s$ . If we define  $\Phi(s)$  as the common value function as in Eq. (10.15) and substitute Eq. (10.65) into Eq. (10.64), we obtain

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1}) - V(s_t) \quad (10.66)$$

---

<sup>5</sup>A model is called underdetermined if there are multiple alternative sets of parameters that can achieve the same objective.

It is interesting to see that this function is exactly the same as the advantage function used in PPO. This relates advantage-based methods to reward shaping: the advantage is essentially a shaped reward.

On the other hand, one of the reasons for adopting end-of-sequence rewards lies in the nature of the RLHF tasks. Unlike traditional reinforcement learning environments where the agent interacts with a dynamic environment, RLHF tasks often involve complex decision-making based on linguistic or other high-level cognitive processes. These processes do not lend themselves easily to frequent and meaningful intermediate rewards because the quality and appropriateness of the actions can only be fully evaluated after observing their impact in the larger context of the entire sequence or task. In this case, the reward signals based on human feedback, though very sparse, are typically very informative and accurate. Consequently, this sparsity, together with the high informativeness and accuracy of the human feedback, can make the learning both robust and efficient.

### 3. Fine-grained Rewards

For many applications, our objective will be more complex than merely evaluating an entire text. For example, in sentiment analysis, we often do not just determine the sentiment of a text, but need to analyze the sentiment in more detail by associating it with specific aspects of a topic discussed in the text. Consider the sentence "The camera of the phone is excellent, but the battery life is disappointing." In this example, we would need to separately analyze the sentiments expressed about the camera and the battery. Such analysis, known as aspect-based sentiment analysis, helps provide a finer-grained understanding of the customer review compared to general sentiment analysis.

For the problem of reward modeling, we often need to model different parts of a sequence as well. A simple and straightforward way to do this is to divide a sequence into different segments and then compute the reward for each segment [Wu et al., 2023b]. Suppose that an output token sequence  $\mathbf{y}$  can be divided into  $n_s$  segments  $\{\bar{\mathbf{y}}_1, \dots, \bar{\mathbf{y}}_{n_s}\}$  by some criterion. We can use the reward model to evaluate each of these segments. By taking  $\mathbf{x}$ ,  $\mathbf{y}$  and  $\bar{\mathbf{y}}_k$  as input to the reward model, the reward score for the  $k$ -th segment is given by

$$r^k = r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \quad (10.67)$$

Then the reward score for the entire output sequence is given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \quad (10.68)$$

Here  $r(\mathbf{x}, \mathbf{y})$  can be used to train the policy as usual.

A problem with this model is that training reward models at the segment level is not as straightforward as learning from human preferences on entire texts, as it is difficult to obtain segment-level human preference data. For rating-like problems (e.g., we rate a segment according to its level of misinformation), one simple approach is to assign a rating score to each segment and train the reward model using pointwise methods. For example, we can use a

strong LLM to rate the sequences  $\bar{y}_1 \dots \bar{y}_{k-1}$  and  $\bar{y}_1 \dots \bar{y}_k$ , and obtain the scores  $s(\bar{y}_1 \dots \bar{y}_{k-1})$  and  $s(\bar{y}_1 \dots \bar{y}_k)$ . We can then define the score of the segment  $\bar{y}_k$  as the difference between  $s(\bar{y}_1 \dots \bar{y}_k)$  and  $s(\bar{y}_1 \dots \bar{y}_{k-1})$

$$s(\bar{y}_k) = s(\bar{y}_1 \dots \bar{y}_k) - s(\bar{y}_1 \dots \bar{y}_{k-1}) \quad (10.69)$$

Using these segment-level scores, we can train the reward model with a regression loss function

$$\mathcal{L}_{\text{rating}} = -\mathbb{E}_{\bar{y}_k} [s(\bar{y}_k) - r(\mathbf{x}, \mathbf{y}, \bar{y}_k)]^2 \quad (10.70)$$

Sometimes, alignment can be treated as a classification problem, for example, we assess whether a segment has ethical issues. In this case, the segment can be labeled as ethical or unethical, either by humans or using additional classifiers. Given the label of the segment, we can train the reward model using some classification loss function. For example, suppose that  $r(\mathbf{x}, \mathbf{y}, \bar{y}_k) = 1$  if the segment is classified as unethical, and  $r(\mathbf{x}, \mathbf{y}, \bar{y}_k) = -1$  otherwise<sup>6</sup>. The hinge loss of training binary classification models is given by

$$\mathcal{L}_{\text{hinge}} = \max(0, 1 - r(\mathbf{x}, \mathbf{y}, \bar{y}_k) \cdot \hat{r}) \quad (10.71)$$

where  $\hat{r} \in \{1, -1\}$  denotes the ground truth label.

The remaining issue here is how to split  $\mathbf{y}$  into segments. One approach is to define a fixed-length segmentation, where  $\mathbf{y}$  is divided into equal-length chunks. However, this may not always be ideal, as the content of the sequence may not align well with fixed boundaries. An alternative approach is to segment  $\mathbf{y}$  based on specific linguistic or semantic cues, such as sentence boundaries, topic shifts, or other meaningful structures in the text. Such a segmentation can be achieved by using linguistic segmentation systems or prompting LLMs to identify natural breaks in the sequence. Another approach is to use dynamic segmentation methods based on the complexity of the sequence. For example, segments could be defined where there is a significant change in the reward score, which might correspond to shifts in the task being modeled.

#### 4. Combination of Reward Models

A reward model can be viewed as a proxy for the environment. Since the true environment is often too complex or unknown, developing a perfect proxy for the environment is generally not possible. As a result, over-aligning LLMs with this imperfect proxy might lead to decreased performance, known as the **overoptimization problem** [Stiennon et al., 2020; Gao et al., 2023a]<sup>7</sup>. We can also explain this through Goodhart’s law, which states: *when a measure*

<sup>6</sup>To allow the reward model to output categories, we can replace the linear layer described in Section 10.3.2 with a Softmax layer.

<sup>7</sup>This problem is also called **reward hacking** or **reward gaming** [Krakovna et al., 2020; Skalse et al., 2022; Pan et al., 2022], which refers to the phenomenon where the agent attempts to trick the reward model but fails to align its actions with the true intended objectives of the task. Imagine a student who is assigned homework and is rewarded with points or praise for completing it. The student might then find ways to finish the homework

becomes a target, it ceases to be a good measure [Goodhart, 1984].

Addressing the overoptimization problem is not easy, and there is no mature solution yet. The ideal approach might be to develop an oracle reward model that can perfectly capture the true objectives of the task and prevent the agent from “tricking”. However, creating such a model is extremely difficult due to the complexity of the real-world environment, as well as the challenge of defining all the relevant factors that contribute to the desired outcome. Instead, a more practical approach is to combine multiple reward models, thereby alleviating the misalignment between the training objective and the true objective that arises from using a single, specific reward model [Coste et al., 2024].

Given a set of reward models, combining them is straightforward, and in some cases, we can simply treat this problem as an ensemble learning problem. A simple yet common approach is to average the outputs of these models to obtain a more precise reward estimation:

$$r_{\text{combine}} = \frac{1}{K} \sum_{k=1}^K w_k \cdot r_k(\mathbf{x}, \mathbf{y}) \quad (10.72)$$

where  $r_k(\cdot)$  is the  $k$ -th reward model in the ensemble,  $w_k$  is the weight of  $r_k(\cdot)$ , and  $K$  is the number of reward models. This combined reward can then be used to supervise the training of a policy. In fact, there are many ways to combine different models, for example, one can make predictions using Bayesian model averaging or develop a fusion network to learn to combine the predictions from different models. Alternatively, one can frame this task as a multi-objective optimization problem, and use multiple reward models to train the policy simultaneously. These methods have been intensively discussed in the literature on optimization and machine learning [Miettinen, 1999; Bishop, 2006].

In addition to model combination methods, another important issue is how to collect or construct multiple different reward models. One of the simplest approaches is to employ ensemble learning techniques, such as developing diverse reward models from different subsets of a given dataset or from various data sources. For RLHF, it is also possible to construct reward models based on considerations of different aspects of alignment. For example, we can develop a reward model to evaluate the factual accuracy of the output and another reward model to evaluate the completeness of the output. These two models are complementary to each other, and can be combined to improve the overall evaluation of the output. Another approach is to employ different off-the-shelf LLMs as reward models. This approach is simple and practical, as there have been a lot of well-developed LLMs and we just need to use them with no or little modification. An interesting issue, though not closely related to the discussion here, arises: can an LLM that aligns with other LLMs outperform those LLMs? Probably not at first glance. In part, this is because the target LLM merely imitates other LLMs based on limited supervision and thus cannot capture well the nuances of the behaviors of these supervisors. However, given the strong generalization ability of LLMs, this approach can, in fact, be quite beneficial. For example, using open-sourced or commercial LLMs as reward

---

with minimal effort to maximize the reward, such as copying and pasting solutions from the internet or previous assignments, rather than solving the problems themselves.

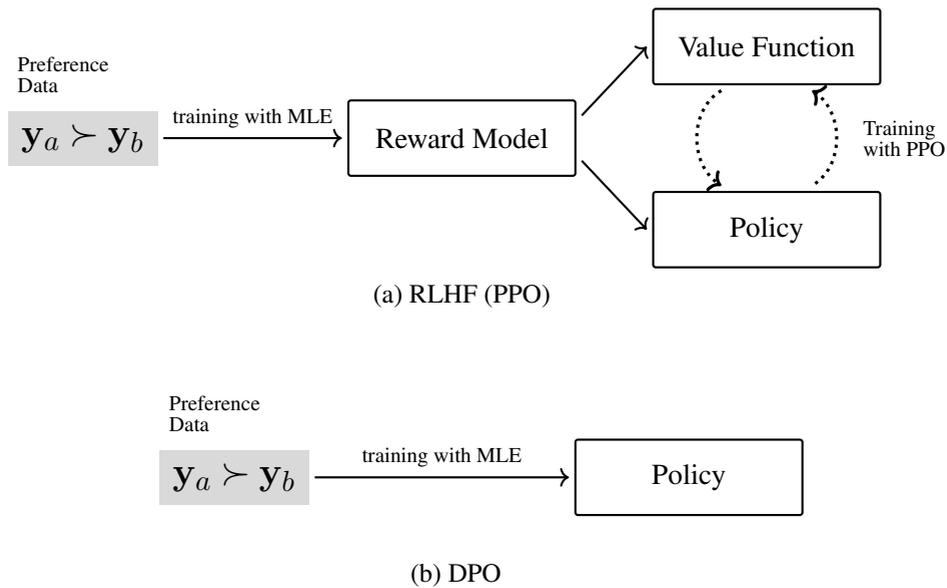


Figure 10.10: Standard RLHF (PPO) vs. DPO. In RLHF, the human preference data is used to train a reward model, which is then employed in training the policy as well as the value function. In DPO, the use of human preference data is more direct, and the policy is trained on this data without the need for reward model training.

models has demonstrated strong performance in aligning LLMs, even achieving state-of-the-art results on several popular tasks [Lambert et al., 2024].

## 10.4.2 Direct Preference Optimization

Although learning reward models is a standard step in reinforcement learning, it makes the entire training process much more complex than supervised training. Training a reliable reward model is itself not an easy task, and a poorly trained reward model can greatly affect the outcome of policy learning. We now consider an alternative alignment method, called **direct preference optimization (DPO)**, which simplifies the training framework by eliminating the need to explicitly model rewards [Rafailov et al., 2024]. This method directly optimizes the policy based on user preferences, rather than developing a separate reward model. As a result, we can achieve human preference alignment in a supervised learning-like fashion. Figure 10.10 shows a comparison of the standard RLHF method and the DPO method.

Before deriving the DPO objective, let us first review the objective of policy training used in RLHF. As discussed in Section 10.3.3, the policy is typically trained by optimizing a loss function with a penalty term. The DPO method assumes a simple loss function where the quality of the output  $y$  given the input  $x$  is evaluated by the reward model  $r(x, y)$ . The training

objective is thus given by

$$\tilde{\theta} = \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \underbrace{-r(\mathbf{x}, \mathbf{y})}_{\text{loss}} + \beta \underbrace{(\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}))}_{\text{penalty}} \right] \quad (10.73)$$

Note that in this optimization problem, only the term  $\pi_{\theta}(\mathbf{y}|\mathbf{x})$  depends on the target policy  $\pi_{\theta}(\cdot)$ . Both the reward model  $r(\mathbf{x}, \mathbf{y})$  and the reference model  $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$  are assumed to be fixed given  $\mathbf{x}$  and  $\mathbf{y}$ . This is a strong assumption compared with PPO, but as will be shown later, it simplifies the problem and crucial for deriving the DPO objective.

Since  $\theta$  is the variable we want to optimize, we rearrange the right-hand side of Eq. (10.73) to isolate  $\pi_{\theta}(\mathbf{y}|\mathbf{x})$  as an independent term:

$$\begin{aligned} \tilde{\theta} &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \beta \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \beta \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) - r(\mathbf{x}, \mathbf{y}) \right] \\ &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \left( \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) + \frac{1}{\beta} r(\mathbf{x}, \mathbf{y}) \right) \right] \\ &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \underbrace{\log \pi_{\theta}(\mathbf{y}|\mathbf{x})}_{\text{dependent on } \theta} - \underbrace{\log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}_{\text{not dependent on } \theta} \right] \quad (10.74) \end{aligned}$$

This equation defines the objective function as the difference between the log-probability distribution of  $y$  and another function of  $y$ . This form of the objective function seems not “ideal”, as we usually prefer to see the difference between two distributions, so that we can interpret this difference as some kind of divergence between the distributions. A simple idea is to convert the second term (i.e.,  $\log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))$ ) into a log-probability distribution over the domain of  $y$ . If we treat  $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))$  as an unnormalized probability of  $y$ , we can convert it into a normalized probability by dividing it by a normalization factor:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right) \quad (10.75)$$

Hence we can define a probability distribution by

$$\pi^*(\mathbf{y}|\mathbf{x}) = \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}{Z(\mathbf{x})} \quad (10.76)$$

We then rewrite Eq. (10.74) as

$$\begin{aligned}
\tilde{\theta} &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}{Z(\mathbf{x})} \right. \\
&\quad \left. - \log Z(\mathbf{x}) \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi^*(\mathbf{y}|\mathbf{x}) - \log Z(\mathbf{x}) \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[ \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi^*(\mathbf{y}|\mathbf{x}) \right] \right. \\
&\quad \left. - \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[ \log Z(\mathbf{x}) \right] \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[ \underbrace{\text{KL}(\pi_{\theta}(\cdot|\mathbf{x}) \parallel \pi^*(\cdot|\mathbf{x}))}_{\text{KL divergence}} - \underbrace{\log Z(\mathbf{x})}_{\text{constant wrt. } \theta} \right] \tag{10.77}
\end{aligned}$$

Since  $\log Z(\mathbf{x})$  is independent of  $\theta$ , it does not affect the result of the  $\arg \min_{\theta}$  operation, and can be removed from the objective. Now we obtain a new training objective which finds the optimal policy  $\pi_{\theta}$  by minimizing the KL divergence between  $\pi_{\theta}(\cdot|\mathbf{x})$  and  $\pi^*(\cdot|\mathbf{x})$

$$\tilde{\theta} = \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[ \text{KL}(\pi_{\theta}(\cdot|\mathbf{x}) \parallel \pi^*(\cdot|\mathbf{x})) \right] \tag{10.78}$$

Clearly, the solution to this optimization problem is given by

$$\begin{aligned}
\pi_{\theta}(\mathbf{y}|\mathbf{x}) &= \pi^*(\mathbf{y}|\mathbf{x}) \\
&= \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}{Z(\mathbf{x})} \tag{10.79}
\end{aligned}$$

Given this equation, we can express the reward  $r(\mathbf{x}, \mathbf{y})$  using the target model  $\pi_{\theta}(\mathbf{y}|\mathbf{x})$ , the reference model  $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$ , and the normalization factor  $Z(\mathbf{x})$ :

$$r(\mathbf{x}, \mathbf{y}) = \beta \left( \log \frac{\pi_{\theta}(\mathbf{y}|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})} + \log Z(\mathbf{x}) \right) \tag{10.80}$$

This is interesting because we initially seek to learn the policy  $\pi_{\theta}(\cdot)$  using the reward model  $r(\mathbf{x}, \mathbf{y})$ , but eventually obtain a representation of the reward model based on the policy. Given the reward model defined in Eq. (10.80), we can apply it to the Bradley-Terry model to

calculate the preference probability (also see Section 10.3.2):

$$\begin{aligned}
 \Pr_{\theta}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) &= \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)) \\
 &= \text{Sigmoid}\left(\beta \left( \log \frac{\pi_{\theta}(\mathbf{y}_a | \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a | \mathbf{x})} + \log Z(\mathbf{x}) \right) - \right. \\
 &\quad \left. \beta \left( \log \frac{\pi_{\theta}(\mathbf{y}_b | \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b | \mathbf{x})} + \log Z(\mathbf{x}) \right) \right) \\
 &= \text{Sigmoid}\left(\beta \log \frac{\pi_{\theta}(\mathbf{y}_a | \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a | \mathbf{x})} - \beta \log \frac{\pi_{\theta}(\mathbf{y}_b | \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b | \mathbf{x})}\right) \quad (10.81)
 \end{aligned}$$

This formula is elegant because it converts the difference in rewards into the difference in ratio functions, and we do not need to calculate the value of  $Z(\mathbf{x})$ . A direct result is that we no longer need a reward model, but only need the target policy and reference model to calculate the probability of preferences. Finally, we can train the target policy by minimizing the following DPO loss function

$$\mathcal{L}_{\text{dpo}}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_{\theta}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \quad (10.82)$$

The form of this loss function is very similar to that used in training reward models in RLHF (see Eq. (10.36)). But it should be noted that the loss function here depends on the parameters of the policy (i.e.,  $\theta$ ) rather than the parameters of the reward model (i.e.,  $\phi$ ).

The main advantage of DPO lies in its simplicity and efficiency. The DPO objective is very straightforward — it directly optimizes for preference-based feedback, rather than relying on separately developed reward models. Moreover, DPO is generally more sample-efficient, as it learns from a fixed dataset without the need for the computationally expensive sampling process used in PPO. This makes DPO a popular method for human preference alignment, especially when developing and applying reward models via reinforcement learning is challenging.

DPO can broadly be viewed as an **offline reinforcement learning** method, where the training data is pre-collected and fixed, and there is no exploration. In contrast, online reinforcement learning methods like PPO, which require exploring new states through interaction with the environment (using the reward model as a proxy), also have their unique advantages. One of the benefits of online reinforcement learning is that it allows the agent to continuously adapt to changes in the environment by learning from real-time feedback. This means that, unlike offline methods, online methods are not constrained by the static nature of pre-collected data and can discover new problem-solving strategies. In addition, exploration can help the agent cover a wider range of state-action pairs, thus improving generalization. This could be an important advantage for LLMs, as generalization is considered a critical aspect in applying such large models.

### 10.4.3 Automatic Preference Data Generation

Although learning from human preferences is an effective and popular method for aligning LLMs, annotating preference data is costly. Using human feedback does not only face the problem of limited scalability, but it may also introduce bias because human feedback is

inherently subjective. As a result, one can turn to AI feedback methods to address these scalability and consistency issues without the limitations associated with human annotators.

As with data generation for instruction fine-tuning, generating preference data using LLMs is straightforward. Given a set of inputs, we first use an LLM to generate pairs of outputs. Then, we prompt the LLM to label the preference between each pair of outputs, along with its corresponding input. Below is an example of prompting the LLM to generate a preference label for a pair of consumer service responses.

Consider a customer service scenario where a customer poses a request. You will review two responses to this request. Please indicate which response is preferred. Note that a good response should be courteous, clear, and concise. It should address the customer’s concern directly, provide helpful information or a solution, and maintain a positive tone.

Request:

*Hello, I noticed that my order hasn’t arrived yet, though it was scheduled to arrive several days ago. Could you please update me on its status? Thank you!*

Response A:

*I’m very sorry for the delay and understand how disappointing this can be. We’re doing our best to sort this out quickly for you.*

Response B:

*Hey, stuff happens! Your package will get there when it gets there, no need to stress.*

Response A is preferred.

Once we collect such preference labels, we can use them, along with the output pair and input, to train the reward model. Of course, we can consider demonstrating a few examples or using advanced prompting techniques, such as CoT, to improve labeling performance. For example, we can include in the prompt an example showing how and why one of the two responses is preferred based on a CoT rationale.

In addition to preference labels, we can also obtain the probability associated with each label [Lee et al., 2023]. A simple method is to extract the probabilities for the label tokens, such as “A” and “B”, from the probabilities output by the LLM. We can then use the Softmax function or other normalization techniques to re-normalize these probabilities into a distribution over the labels. These probabilities of preferred labels can serve as pointwise supervision signals for training the reward model, as discussed in Section 10.4.1.

For data generation, although it is easy to scale up, it is often necessary to ensure the data is accurate and diverse. Here, the data quality and diversity issues involve not only the

labeling of preferences but also the inputs and outputs of the model. Therefore, we often need to use a variety of techniques to obtain large-scale, high-quality data. For example, one can generate diverse model outputs and annotations by using different LLMs, prompts, in-context demonstrations, and so on [Cui et al., 2024]. Dubois et al. [2024] report that the variability in pairwise preference data is important for training LLMs from either human or AI feedback.

While learning from AI feedback is highly scalable and generally objective, this method is more suited to well-defined tasks where objective performance metrics are available. By contrast, learning from human feedback is more advantageous when aligning AI systems with human values, preferences, and complex real-world tasks that require understanding of subtle or subjective context. These methods can be combined to train LLMs that benefit from both human insights and the scalability of AI feedback.

#### 10.4.4 Step-by-step Alignment

So far, our discussion of alignment has primarily focused on the use of reward models for evaluating entire input-output sequence pairs. These methods can be easily adapted to scenarios where the correctness of an output can be examined by checking whether the desired result is included. For example, in the task of calculating a mathematical expression, a reward model can provide positive feedback if the answer is correct, and negative feedback if the answer is wrong. However, in many problems that require complex reasoning, simply examining the correctness of the final result is insufficient for learning. Imagine a student who is only given the final answer to a challenging math problem. Knowing whether the final answer is right or wrong does not help the student figure out where they went wrong and how to calculate the correct answer. A better approach would be to guide the student with a step-by-step breakdown of the problem-solving process and encourage understanding of the underlying concepts and logic behind these steps.

In Chapter 9, we studied CoT methods to prompt LLMs to explicitly write out intermediate steps or the reasoning process needed to reach a conclusion or solve a problem. We saw that breaking down a problem into smaller parts could make it easier to understand the solution path and increase the accuracy of the output. These methods can be naturally extended to the alignment of LLMs, that is, we supervise the model during the intermediate steps of reasoning. Consider a reasoning task where an LLM produces a sequence of reasoning steps  $\mathbf{y} = \{\bar{y}_1, \dots, \bar{y}_{n_s}\}$  for the given input. The result of the reasoning is assumed to be included in the last step  $\bar{y}_{n_s}$ , and can be easily verified. For this reasoning problem, Uesato et al. [2022] categorize LLM fine-tuning approaches into two classes:

- **Outcome-based Approaches.** Supervision occurs only when the end result is verified. This is a standard method for learning from human feedback we have discussed in this chapter. For example, the LLM is optimized to maximize some form of the reward  $r(\mathbf{x}, \mathbf{y})$ .
- **Process-based Approaches.** Supervision is involved in all intermediate steps in addition to the last step. To do this, we need to develop a model to give a supervision signal at each step, and develop loss functions that can make use of such supervision signals.

**Question** (from The International Junior Math Olympiad):

Jessica has a lot of apps on her phone. The number of apps she has is divisible by 5. If she downloaded 10 more apps, she would still have less than 50 apps. What is the maximum original number of apps Jessica could have?

**Response 1:**

$\bar{y}_1$ :  $x$  must be a multiple of 5, i.e.,  
 $x \equiv 0 \pmod{5}$ .

$\bar{y}_2$ : After downloading 10 apps, the total total number of apps should still be less than 50:  $x + 10 < 50$ . Solving this inequality:  $x < 40$ .

$\bar{y}_3$ : Since  $x$  must be a multiple of 5 and less than 40, the largest value of  $x$  is 35.

$\bar{y}_4$ : Verify the result. If  $x = 35$ , then after downloading 10 apps, Jessica would have:  $35 + 10 = 45$  which satisfy  $45 < 50$ .

$\bar{y}_5$ : The maximum original number of apps is 35.

↑  
correct

**Response 2:**

$\bar{y}_1$ : The number of apps is divisible by 5, so  $x$  must be a multiple of 5.

$\bar{y}_2$ : If she downloads 10 more apps, her new total will be  $x + 10$ , and this must be less than 50. Therefore,  $x + 10 < 50$ . Simplifying this gives  $x < 40$ .

$\bar{y}_3$ :  $x$  must be less than or equal to the largest multiple of 5, which is 40. problematic

$\bar{y}_4$ : But  $x + 10$  should not be more than or equal to 50. So we need to subtract 5 from 40. problematic

$\bar{y}_5$ : Therefore, the final result is 35.

↑  
correct

Figure 10.11: Two LLM responses to a math problem. In response 1, both the final result and all the reasoning steps are correct. In response 2, the final result is correct, but there are mistakes in the reasoning process (highlighted in red). For outcome-based approaches, both responses are considered correct. For process-based approaches, the mistakes in response 2 can be considered in reward modeling.

Figure 10.11 shows two LLM outputs for an example math problem. Although the LLM gives the correct final answer in both cases, it makes mistakes during the problem-solving process in the second output. Outcome-based approaches overlook these mistakes and give positive feedback for the entire solution. By contrast, process-based approaches can take these mistakes into account and provide additional guidance on the detailed reasoning steps.

An important issue for process-based approaches is that we need to get step-level feedback during a (potentially) long reasoning path. We can collect or generate reasoning paths corresponding to problems from existing datasets. Human experts then annotate each step in these paths for correctness. These annotations can be used to directly train LLMs or as rewards in reward modeling. However, in practice, richer annotations are often introduced [Lightman et al., 2024]. In addition to the *correct* and *incorrect* labels, a step can also be labeled as

*neutral* to indicate that while the step may be technically correct, it might still be problematic within the overall reasoning process. Furthermore, to improve the efficiency of data annotation, techniques such as active learning can be employed. Identifying obvious errors usually does not significantly contribute to learning from reasoning mistakes. Instead, annotating steps that the model confidently considers correct but are actually problematic is often more effective.

Given a set of step-level annotated reasoning paths and corresponding inputs, we can train a reward model to provide feedback for supervising policy learning. The reward model can be treated as a classification model, and so its architecture can be a Transformer decoder with a Softmax layer stacked on top. At step  $k$ , the reward model takes both the problem description (denoted by  $\mathbf{x}$ ) and the reasoning steps generated so far (denoted by  $\bar{\mathbf{y}}_{\leq k}$ ) as input and outputs a probability distribution over the label set  $\{correct, incorrect\}$  or  $\{correct, incorrect, neutral\}$ . Then the learned reward model is used to evaluate reasoning paths by assessing the correctness of each step. A simple method to model correctness is to count the number of steps that are classified as *correct*, given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \delta(correct, C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})) \quad (10.83)$$

where  $C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$  denotes the label with the maximum probability. We can also use log-probabilities of classification to define the reward of the entire path

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \log \Pr(correct | \mathbf{x}, \bar{\mathbf{y}}_{\leq k}) \quad (10.84)$$

where  $\Pr(correct | \mathbf{x}, \bar{\mathbf{y}}_{\leq k})$  denotes the probability of the *correct* label generated by the reward model. The reward score  $r(\mathbf{x}, \mathbf{y})$  can then be used to train the policy in RLHF as usual.

While we restrict our discussion to math problems, the approaches described here are general and can be applied to a wide variety of tasks that involve multi-step reasoning and decision-making. Moreover, we can consider various aspects when assessing the quality of a step, rather than just its correctness. For example, in dialogue systems, responses must not only be accurate but also contextually appropriate across multiple turns of conversation. If a model provides a correct response but fails to maintain coherence in the context of the ongoing dialogue, step-level feedback could help the model identify and correct such discrepancies. Also note that the process-based approaches are related to the fine-grained reward modeling approaches discussed in Section 10.4.1. All these approaches essentially aim to provide more detailed supervision to LLMs by breaking their outputs into smaller, more manageable steps. However, process-based feedback focuses more on evaluating the correctness of a step based on its preceding steps, while the approaches in Section 10.4.1 emphasize evaluating each step independently.

The idea of aligning LLMs step by step has great application potential, especially considering the recent shift towards more complex reasoning tasks in the use of LLMs. For example, both the GPT-o1 and GPT-o3 models are designed with more advanced reasoning techniques (such as long internal CoT) to solve challenging problems like scientific and mathematical

reasoning [OpenAI, 2024]. These tasks often rely on long and complex reasoning paths, and therefore, it seems essential to introduce detailed supervision signals in the reasoning process. Moreover, from a practical perspective, effective supervision on long reasoning paths not only improves reasoning performance, but it also helps the model eliminate redundant or unnecessary reasoning steps, thereby reducing reasoning complexity and improving efficiency.

### 10.4.5 Inference-time Alignment

In this section we explored a variety of methods to align models with human preferences and annotations. However, one of the significant limitations of many such methods is that LLMs must be fine-tuned. For RLHF and its variants, training LLMs with reward models can be computationally expensive and unstable, leading to increased complexity and costs when applying these approaches. In this case, we can consider aligning models at inference time, thus avoiding the additional complexity and effort involved.

One simple way to achieve inference-time alignment is to use the reward model to select the best one from  $N$  alternative outputs generated by the LLM, a method known as **Best-of- $N$  sampling (BoN sampling)**. We can consider BoN sampling as a form of reranking. In fact, reranking methods have been widely used in NLP tasks, such as machine translation, for a long time. They are typically applied in situations where training complex models is costly. In such cases, directly reranking the outputs allows for the incorporation of these complex models at a very low cost<sup>8</sup>.

In the BoN sampling process, the LLM takes the input sequence  $\mathbf{x}$  and generates  $N$  different output sequences  $\{\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_N\}$ :

$$\{\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_N\} = \underset{\mathbf{y}}{\operatorname{argTopN}}[\operatorname{Pr}(\mathbf{y}|\mathbf{x})] \quad (10.85)$$

where the  $\operatorname{argTopN}$  operation returns the top- $N$  outputs that maximize the function  $\operatorname{Pr}(\mathbf{y}|\mathbf{x})$ . These outputs can be generated in a variety of ways, depending on the search algorithm used by the model (e.g., sampling or beam search). Once the  $N$ -best output candidates are generated, the reward model is used to evaluate and select the best one:

$$\hat{\mathbf{y}}_{\text{best}} = \max\{r(\mathbf{x}, \hat{\mathbf{y}}_1), \dots, r(\mathbf{x}, \hat{\mathbf{y}}_N)\} \quad (10.86)$$

It is worth noting that the result of BoN sampling is also influenced by the diversity of the  $N$ -best list. This is a common issue with most reranking methods. Typically, we wish the  $N$ -best output candidates to have relatively high quality but be sufficiently different from each other. In many text generation systems, the  $N$ -best outputs are very similar, often differing by

---

<sup>8</sup>Reranking methods can also help us explore what are known as model errors and search errors, although these issues are not often discussed in the context of LLMs. For example, suppose we have an old model and a new, more powerful model. We can use the new model to select the best output from the  $N$ -best list of the old model as the oracle output. The performance difference between the oracle output and the top-1 output of the original  $N$ -best list reflects the performance gain brought by the new model. If the performance gain is significant, we can say that the old model has more model errors. If the gain is small, it may indicate that the issue lies in search errors, as the best candidates were not found.

just one or two words. The diversity issue is even more challenging in LLMs, as the  $N$ -best outputs generated by an LLM can be different in their wordings, yet their semantic meanings are often quite similar. In practice, one can adjust the model hyperparameters and/or adopt different LLMs to generate more diverse output candidates for reranking. Nevertheless, as with many practical systems, we need to make a trade-off between selecting high-quality candidates and ensuring sufficient variation in the generated outputs.

BoN sampling can be used for training LLMs as well. A closely related method is **rejection sampling**. In this method, we first select the “best” outputs from the  $N$ -best lists via the reward model, and then take these selected outputs to fine-tune the LLM. In this way, we can introduce human preferences into the training of LLMs via a much simpler approach compared to RLHF. Many LLMs have adopted rejection sampling for fine-tuning [Nakano et al., 2021; Touvron et al., 2023b].

## 10.5 Summary

In this chapter, we have explored a range of techniques for aligning LLMs. In particular, we have discussed fine-tuning methods that enable LLMs to follow instructions and align them with human preferences. One of the benefits of fine-tuning LLMs is computation efficiency. Unlike pre-training based on large-scale neural network optimization, fine-tuning is a post-training step and so is less computationally expensive. Moreover, it is better suited to address problems that are not easily solved in pre-training, such as human value alignment. The widespread attention to the alignment issue has also led to a surge of research papers on this topic, which has posed challenges in writing this chapter, as it is difficult to cover all the latest techniques. However, we have tried to provide a relatively detailed introduction to the fundamental approaches to alignment, such as instruction fine-tuning and RLHF.

While we have focused on LLM alignment techniques in this chapter, the term *AI alignment* is a wide-ranging concept. It generally refers to the process of ensuring that the behavior of an AI system aligns with human values, goals, and expectations. The idea of AI alignment can be traced back to the early days of AI. A widely cited description of AI alignment comes from an article by the mathematician and computer scientist Norbert Wiener [Wiener, 1960]. The quote is as follows

*If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere ... we had better be quite sure that the purpose put into the machine is the purpose which we really desire.*

At that time, AI alignment was a distant concern for researchers. But today, it greatly influences the design of various AI systems. For example, in robotics, alignment is critical to ensuring that autonomous robots safely interact with humans and their environments. In autonomous driving, cars must not only follow traffic laws but also make complex, real-time decisions that prioritize human safety, avoid accidents, and navigate ethical dilemmas.

In current AI research, alignment is usually achieved by developing a surrogate objective that is analogous to the real goal and steering the AI system towards this objective. However,

designing the objective of AI alignment is very difficult. One reason is that human values are diverse and often context-dependent, making it difficult to distill them into a single, universally applicable objective function. Also, the complexity of real-world environments, where values and goals often conflict or evolve over time, further complicates alignment efforts. Even if we could define an appropriate objective, AI systems may find unintended ways to achieve it, leading to “misaligned” outcomes that still technically satisfy the objective but in a harmful or counterproductive way.

These challenges have motivated and are motivating AI research towards more aligned systems, either through developing new mechanisms for perceiving the world or more efficient and generalizable methods to adapt these systems to given tasks. More importantly, as AI systems become more powerful and intelligent, especially given that recent advances in LLMs have shown remarkable capabilities in dealing with many challenging problems, the need for AI alignment has become more urgent. Researchers have started to be concerned with AI safety and warn the community that they need to develop and release AI systems with great caution to prevent these systems from being misaligned [[Russell, 2019](#); [Bengio et al., 2024](#)].



# Chapter 11

## Inference

Once we have pre-trained and fine-tuned an LLM, we can apply it to make predictions on new data. This process is called inference, in which the LLM computes the probabilities of different possible outputs given an input, and selects the output that maximizes the probability. The inference problem is generally expressed in the following form:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}} \operatorname{Pr}(\mathbf{y}|\mathbf{x}) \quad (11.1)$$

where  $\mathbf{x}$  is the input sequence,  $\mathbf{y}$  is a possible output sequence, and  $\hat{\mathbf{y}}$  is the best output sequence.

This is perhaps one of the most widely adopted formulas in NLP, and dates back to the early days of speech recognition and machine translation systems based on probabilistic models. Although for some applications, such as predicting a token using a very small language model, solving this optimization problem seems trivial, for most situations the computational challenges arise from both calculating  $\operatorname{Pr}(\mathbf{y}|\mathbf{x})$  and performing the  $\operatorname{argmax}$  operation. The problems we therefore wish to address in this chapter involve: 1) computing the prediction probability efficiently given a trained LLM, and 2) devising an efficient (suboptimal) search for  $\hat{\mathbf{y}}$ .

At a high level, these are fundamental issues in artificial intelligence, which have been extensively studied. So many well-established techniques can be directly applied, for example, one can use greedy search algorithms to implement an efficient inference system. On the other hand, model-specific optimizations, such as efficient attention models for Transformers, can be considered to further improve efficiency. But, in many practical applications, we still need to make a trade-off between accuracy and efficiency, by carefully combining various techniques.

The importance of the inference problem in LLMs also lies in the fact that many application scenarios require processing extremely long sequences. Recent studies have found that injecting additional prompts and contextual information, such as long chain-of-thought prompts, during inference can significantly improve the performance of LLMs. This provides a new approach to scaling LLMs: better results can be achieved by increasing the compute at inference time. For instance, through inference-time scaling, [OpenAI \[2024\]](#)'s o1 and [Deepseek \[2025\]](#)'s R1

systems have demonstrated impressive performance on complex reasoning and programming tasks. This, in turn, has encouraged the NLP field to focus more on the issue of efficient inference.

In this chapter, we will introduce basic concepts and algorithms of LLM inference, including prefilling-decoding frameworks, search (decoding) algorithms, and evaluation metrics of inference performance. We will then present methods for improving the efficiency of LLM inference, covering a range of techniques for speeding up the system and compressing the model. Finally, we will discuss inference-time scaling, which is considered an important application of inference optimization.

## 11.1 Prefilling and Decoding

In this section, we present the prefilling-decoding framework, which is the most commonly used for interpreting and implementing LLM inference processes. We first introduce the notation and background knowledge, and then describe the details of the framework, such as the decoding algorithms for LLM inference.

### 11.1.1 Preliminaries

Although we have described LLMs many times in this book, we begin by briefly defining the notation to facilitate the subsequent discussion, and to make this chapter self-contained.

- x**: The input token sequence. It is conceptually equivalent to a “prompt”, which includes instructions, user inputs, and any additional context intended as input to the LLM. **x** comprises  $m + 1$  tokens, denoted by  $x_0 \dots x_m$ , where  $x_0$  is the start symbol  $\langle \text{SOS} \rangle$ .
- y**: The output token sequence, also called the response to the input. **y** comprises  $n$  tokens, denoted by  $y_1 \dots y_n$ .
- $y_{<i}$** : The output tokens that precede position  $i$ , that is,  $\mathbf{y}_{<i} = y_1 \dots y_{i-1}$ .
- $\Pr(\mathbf{y}|\mathbf{x})$** : The probability of generating **y** given **x** using the LLM. If the LLM is parameterized by  $\theta$ , we can write it as  $\Pr_{\theta}(\mathbf{y}|\mathbf{x})$ .
- $[\mathbf{x}, \mathbf{y}]$** : The concatenated token sequence of **x** and **y**. That is,  $[\mathbf{x}, \mathbf{y}] = x_0 \dots x_m y_1 \dots y_n$ . Occasionally, we use the notation  $\text{seq}_{\mathbf{x}, \mathbf{y}}$  to represent  $[\mathbf{x}, \mathbf{y}]$ .
- $\Pr([\mathbf{x}, \mathbf{y}])$** : The probability of generating the token sequence  $[\mathbf{x}, \mathbf{y}]$  using the LLM.

As described in Eq. (11.1), the goal of LLM inference is to maximize  $\Pr(\mathbf{y}|\mathbf{x})$ . Modeling this conditional probability is common in NLP. At first glance, it seems to be a sequence-to-sequence problem, where we transform a sequence into another using encoding-decoding models. However, we are not discussing sequence-to-sequence problems or encoding-decoding architectures. Instead, as discussed in earlier chapters, this modeling problem can be addressed

by using decoder-only models. To do this, we can interpret the log-scale probability  $\log \Pr(\mathbf{y}|\mathbf{x})$  as the difference between  $\log \Pr([\mathbf{x}, \mathbf{y}])$  and  $\log \Pr(\mathbf{x})$

$$\log \Pr(\mathbf{y}|\mathbf{x}) = \log \Pr([\mathbf{x}, \mathbf{y}]) - \log \Pr(\mathbf{x}) \quad (11.2)$$

where  $\log \Pr([\mathbf{x}, \mathbf{y}])$  and  $\log \Pr(\mathbf{x})$  can be obtained by running the LLM on the sequences  $[\mathbf{x}, \mathbf{y}]$  and  $\mathbf{x}$ , respectively. For example, we can calculate the probability of generating  $\mathbf{x}$  using the chain rule

$$\begin{aligned} \log \Pr(\mathbf{x}) &= \log \Pr(x_0 \dots x_m) \\ &= \log [\Pr(x_0) \Pr(x_1|x_0) \cdots \Pr(x_m|x_0 \dots x_{m-1})] \\ &= \underbrace{\log \Pr(x_0)}_{=0} + \sum_{j=1}^m \log \Pr(x_j|\mathbf{x}_{<j}) \\ &= \sum_{j=1}^m \log \Pr(x_j|\mathbf{x}_{<j}) \end{aligned} \quad (11.3)$$

In other words, we calculate the token prediction log-probability at each position of  $\mathbf{x}$ , and sum all these log-probabilities.

In common implementations of LLMs, however, we do not need to compute the log-probability of the input sequence, but use the LLM to directly compute the log-probability of the output sequence in the following form

$$\log \Pr(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \quad (11.4)$$

where  $[\mathbf{x}, \mathbf{y}_{<i}]$  represents the context for predicting  $y_i$ . We use  $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$  to denote  $\Pr(y_i|[\mathbf{x}, \mathbf{y}_{<i}])$ , following the commonly used notation in the literature.

Now, we have two sub-problems in addressing the inference issue described in Eq. (11.1):

- **Model Computation:** we model  $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$  and compute it in an efficient manner.
- **Search:** we find the optimal (or sub-optimal) output sequence in terms of  $\log \Pr(\mathbf{y}|\mathbf{x})$ .

The second sub-problem is a classic issue in NLP. We will show in Section 11.1.3 that there are several well-studied algorithms that can be applied to efficiently search the space of possible output sequences. The first sub-problem requires a language model to produce a distribution over a vocabulary  $V$  given a sequence of context tokens. We can do this by training a Transformer decoder, which outputs the distribution

$$\Pr(\cdot|\mathbf{x}, \mathbf{y}_{<i}) = \text{Softmax}(\mathbf{H}\mathbf{W}^o)_{m+i} \quad (11.5)$$

$$\mathbf{H} = \text{Dec}([\mathbf{x}, \mathbf{y}_{<i}]) \quad (11.6)$$

Here  $\text{Dec}(\cdot)$  produces a sequence of representations, each corresponding to a position of the input sequence. So, if we input  $[\mathbf{x}, \mathbf{y}_{<i}]$  to the LLM,  $\mathbf{H}$  is an  $i' \times d$  matrix, where  $d$  is the

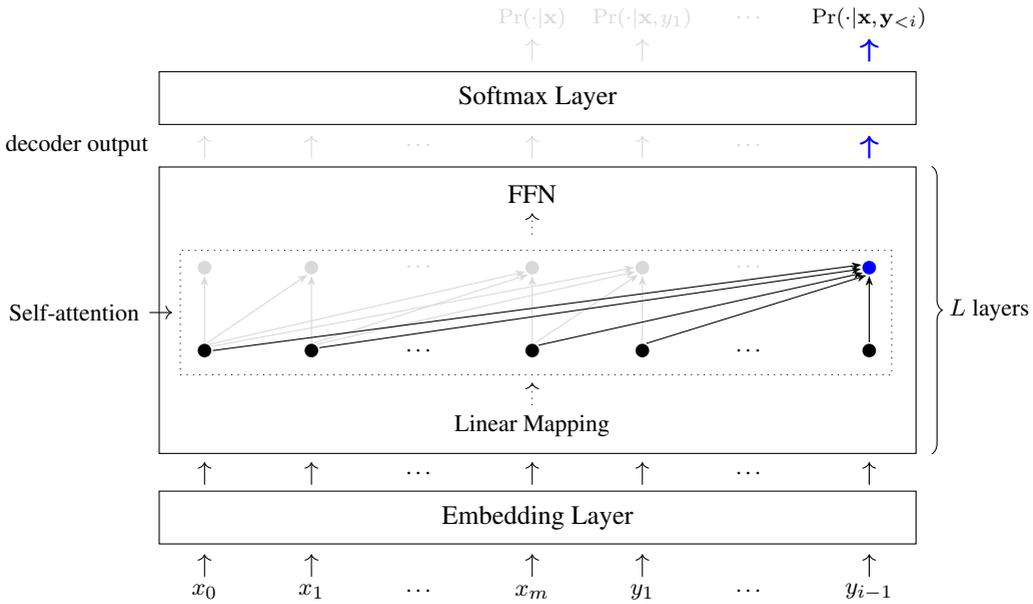


Figure 11.1: The decoder-only architecture for LLMs. The decoder consists of an embedding layer and a stack of Transformer layers. In each Transformer layer, the input passes through a linear mapping, a self-attention network, and an FFN. The output of the decoder is a sequence of representations that are taken as input to a Softmax layer, which generates a distribution of tokens for each position.

dimensionality of each representation, and  $i' = m + i$  is the number of context tokens. We can then use a Softmax layer to transform these representations into distributions of tokens.  $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$  is the linear mapping matrix of the Softmax layer, and  $\mathbf{H}\mathbf{W}^o$  transforms the  $d$ -dimensional representations in  $\mathbf{H}$  into the  $|V|$ -dimensional representations. The use of the subscript  $m + i$  indicates that the Softmax function is performed only on the representation at position  $m + i$ . See Figure 11.1 for an illustration of this architecture.

$\text{Dec}(\cdot)$  is a Transformer decoding network that consists of an embedding network and a number of stacked self-attention and FFN networks. We will not discuss Transformers in detail here, as readers can easily learn about these models from the literature. However, it is worth pointing out that the difficulty of inference is in part from the use of the self-attention mechanism in Transformers. Recall that a general form of single-head self-attention is given by

$$\text{Att}_{\text{qkv}}(\mathbf{q}_{i'}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{q}_{i'} \mathbf{K}^T}{\sqrt{d}}\right) \mathbf{V} \quad (11.7)$$

where  $\mathbf{q}_{i'} \in \mathbb{R}^d$  is the query at the position  $i'$  (i.e., position of  $y_i$ ), and  $\mathbf{K}$  and  $\mathbf{V} \in \mathbb{R}^{i' \times d}$  are the keys and values up to  $i'$ , respectively.

At each step during inference, we call the self-attention function  $\text{Att}_{\text{qkv}}(\cdot)$ , followed by

an FFN, to generate a  $d$ -dimensional representation that integrates information from both the current token and its left context. This process is repeated through  $L$  layers of self-attention and FFN, forming a stack of Transformer layers. The output of the  $L$ -th layer in this stack is the final representation.

Each time, the model attends position  $i'$  to all previous positions, which results in  $2i'$  vector products ( $i'$  times for  $\mathbf{q}_{i'}\mathbf{K}^T$  and  $i'$  times for the product of  $\text{Softmax}(\frac{\mathbf{q}_{i'}\mathbf{K}^T}{\sqrt{d}})$  and  $\mathbf{V}$ ). Hence, generating a sequence of length  $len$  has a time complexity of  $O(L \times len^2)$  for the self-attention network. Clearly, the inference of this model is slow for long sequences due to its quadratic time complexity with respect to sequence length. Therefore, many improvements to Transformers and alternative models have focused on efficient methods that are faster than this quadratic time complexity, such as sparse attention mechanisms and linear-time models. A detailed discussion of efficient Transformers can be found in the previous chapters, and this section will focus on the standard Transformer architecture.

Note that in self-attention, the queries, keys, and values of a layer are linear mappings from the same input (i.e., the output of the previous layer). Once a new key-value pair is generated, it is repeatedly used in subsequent inference steps. Rather than regenerating these key-value pairs during inference, a more desirable way is to store them in a structure, called the **key-value cache**, or the **KV cache**. Thus,  $(\mathbf{K}, \mathbf{V})$  can straightforwardly be considered a KV cache. This cache is updated as follows

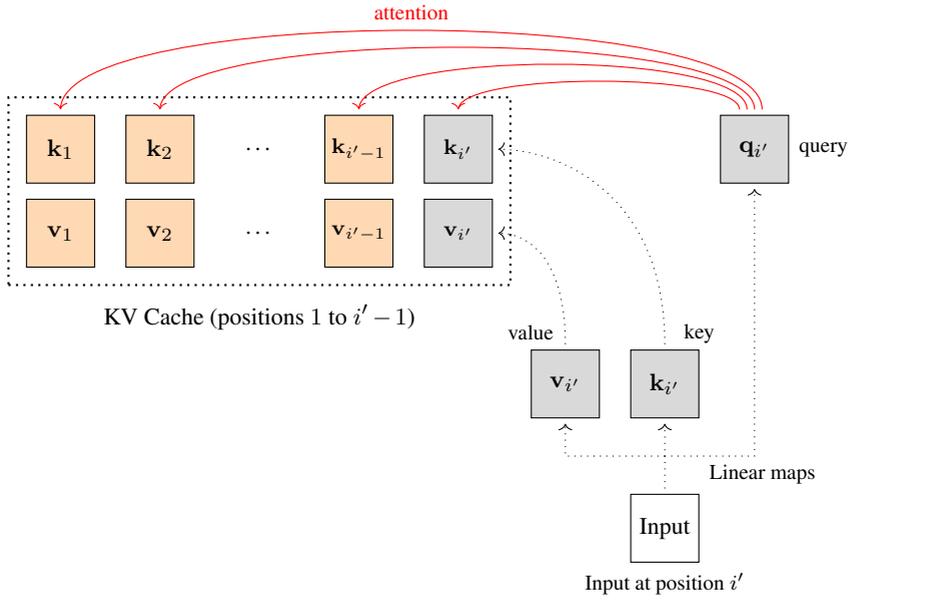
$$\mathbf{K} = \text{Append}(\mathbf{K}, \mathbf{k}_{i'}) \quad (11.8)$$

$$\mathbf{V} = \text{Append}(\mathbf{V}, \mathbf{v}_{i'}) \quad (11.9)$$

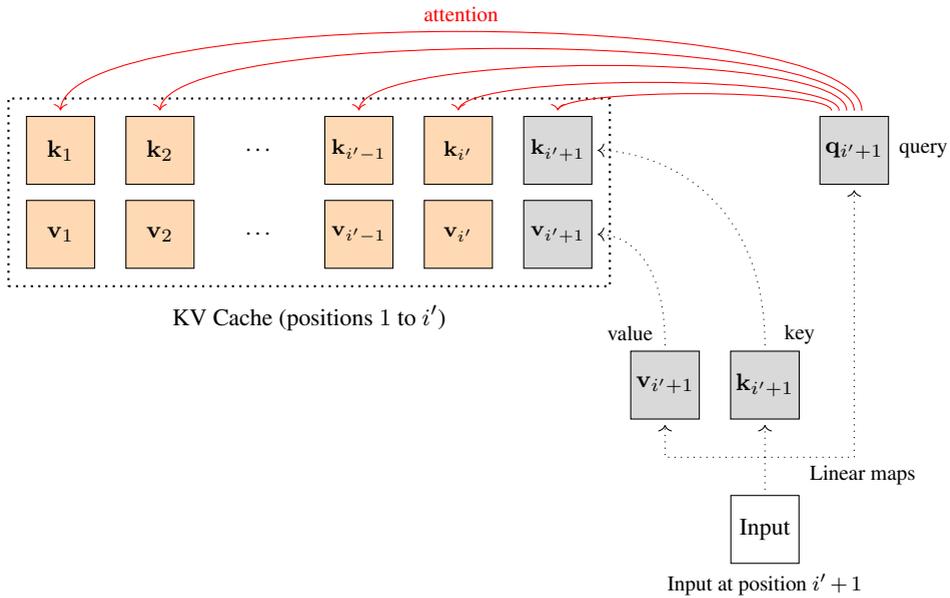
where  $(\mathbf{k}_{i'}, \mathbf{v}_{i'})$  is the newly generated key-value pair at position  $i'$ , and  $\text{Append}(\mathbf{a}, \mathbf{b})$  denotes a function that appends a row vector  $\mathbf{b}$  to a matrix  $\mathbf{a}$ . Figure 11.2 shows how a Transformer decoder works with a KV cache.

Finally, the process of computing  $\log \Pr(\mathbf{y}|\mathbf{x})$  is summarized as follows:

1. We concatenate  $\mathbf{x}$  and  $\mathbf{y}$  into a sequence  $[\mathbf{x}, \mathbf{y}]$ . For each position  $i'$  of this sequence, we perform the following steps.
  - (a) We compute the embedding of the token at position  $i'$ , and feed the resulting embedding as an initial representation into the stack of Transformer layers.
  - (b) In each Transformer layer, we pass the input representation through the self-attention network first and then through an FFN. In the self-attention network, the input representation is transformed into  $\mathbf{q}_{i'}$ ,  $\mathbf{k}_{i'}$ , and  $\mathbf{v}_{i'}$ . Then, we update the KV cache  $(\mathbf{K}, \mathbf{V})$  using  $\mathbf{k}_{i'}$  and  $\mathbf{v}_{i'}$  (see Eqs. (11.8-11.9)). Then, we compute the output of the attention model by attending  $\mathbf{q}_{i'}$  to  $(\mathbf{K}, \mathbf{V})$  (see Eq. (11.7)).
  - (c) If  $i' > m$  (i.e.,  $i = i' - m \geq 0$ ), we take the output of the Transformer stack and compute the token prediction probability  $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$  via the Softmax layer (see Eq. (11.5)).
2. When reaching the end of the sequence, we obtain  $\log \Pr(\mathbf{y}|\mathbf{x})$  by summing  $\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$  over  $i \in [1, n]$  (see Eq. (11.4)).



(a) Updating the KV Cache at Position  $i'$



(b) Updating the KV Cache at Position  $i' + 1$

Figure 11.2: Illustration of the KV cache. We update the KV cache at a position, perform the attention operation, and then move to the next position to repeat the process.

### 11.1.2 A Two-phase Framework

As we have seen, language modeling is a standard autoregressive process, where each token is generated one at a time, conditioned on the previous tokens. For Transformers, this requires the model to maintain a KV cache that stores past representations, and attend the newly generated representation to this KV cache. If we think of the model  $\Pr(\mathbf{y}|\mathbf{x})$  from the perspective of computing the KV cache, it is natural to divide inference into two phases:

- **Prefilling.** The prefilling phase computes the KV cache for the input sequence  $\mathbf{x}$ . It is called prefilling because the model prepares and stores the key-value pairs for each token in the input before the actual inference begins. The process of prefilling in an LLM can be expressed as

$$\text{cache} = \text{Dec}_{\text{kv}}(\mathbf{x}) \quad (11.10)$$

where  $\text{Dec}_{\text{kv}}(\cdot)$  is the decoding network (i.e., the same as  $\text{Dec}(\cdot)$ ), but it returns the KV cache in self-attention instead of the output representations.  $\text{cache}$  is a list, given by

$$\text{cache} = \{\text{cache}^1, \dots, \text{cache}^L\} \quad (11.11)$$

where  $\text{cache}^l$  represents the key-value pairs for the  $l$ -th layer.

- **Decoding.** The decoding phase continues generating tokens based on the KV cache, as illustrated in Figure 11.2. When a new token is input into the decoder, we update the KV cache in each layer by adding the new key-value pair. The updated cache is then used for self-attention computation. The token generation stops when some stopping criterion is met, such as when the generated token is the end symbol. The goal of decoding is to find the best predicted sequence, which is given by

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{arg max}} \Pr(\mathbf{y}|\text{cache}) \quad (11.12)$$

Here we use  $\Pr(\mathbf{y}|\text{cache})$  instead of  $\Pr(\mathbf{y}|\mathbf{x})$  to emphasize that the decoding process actually relies on the KV cache rather than  $\mathbf{x}$ .

The prefilling and decoding processes are illustrated in Figure 11.3. Note that both these processes are autoregressive. However, as shown in Table 11.1, they differ in several aspects, which lead to very different implementations in practice.

In essence, while the underlying model of prefilling is based on token prediction, it can be considered an encoding process. This is because our goal is not to generate tokens, but to build a context representation (i.e., the KV cache) for the subsequent steps in the decoding phase. In this sense, it is similar to BERT, where we encode the input sequence into a sequence of contextualized token representations. On the other hand, unlike BERT which generates bidirectional sequence representations, prefilling is based on standard language modeling tasks, and is thus unidirectional. Note that, since the entire sequence  $\mathbf{x}$  is input into the model all at once, all queries can be packed together and the self-attention operation is performed on  $\mathbf{x}$

	Prefilling	Decoding
Goal	Set up initial context $\mathbf{x}$ .	Continue generating tokens $\mathbf{y}$ after the initial input.
All-at-once Visibility	Tokens in $\mathbf{x}$ are presented all at once.	Tokens in $\mathbf{y}$ are presented sequentially, that is, predicting a token requires waiting for the previous tokens to be predicted first.
Context Use	Build the context or encoded representation of the input.	Use the cached key-value pairs (from prefilling) to generate further tokens.
Resource Limitation	Compute-bound	Memory-bound
Computational Cost	High	Very High

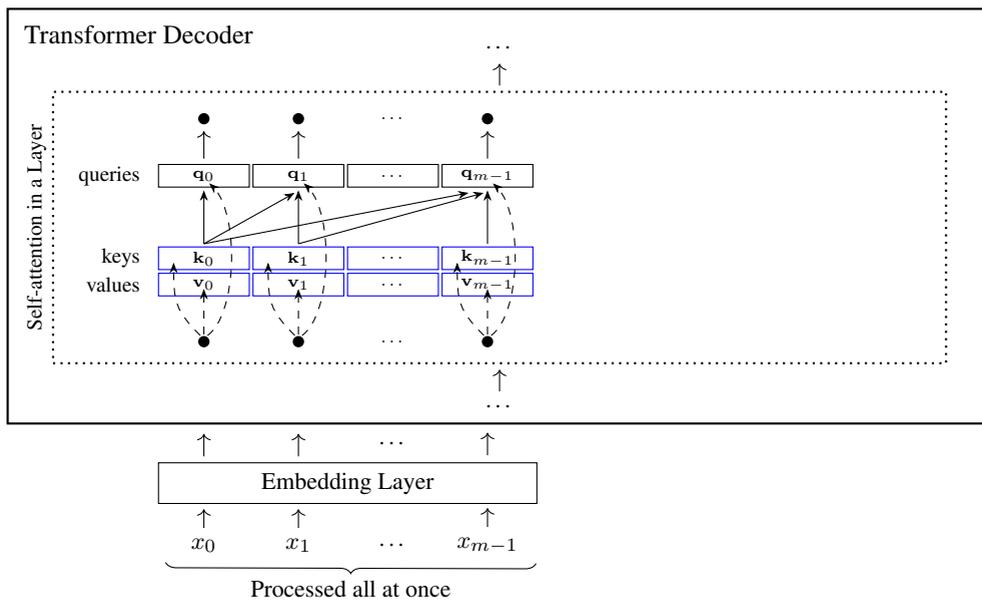
Table 11.1: Prefilling vs Decoding.

in parallel. Let  $\mathbf{Q}$  be the queries that are packed into one matrix. The self-attention model in prefilling can be defined as

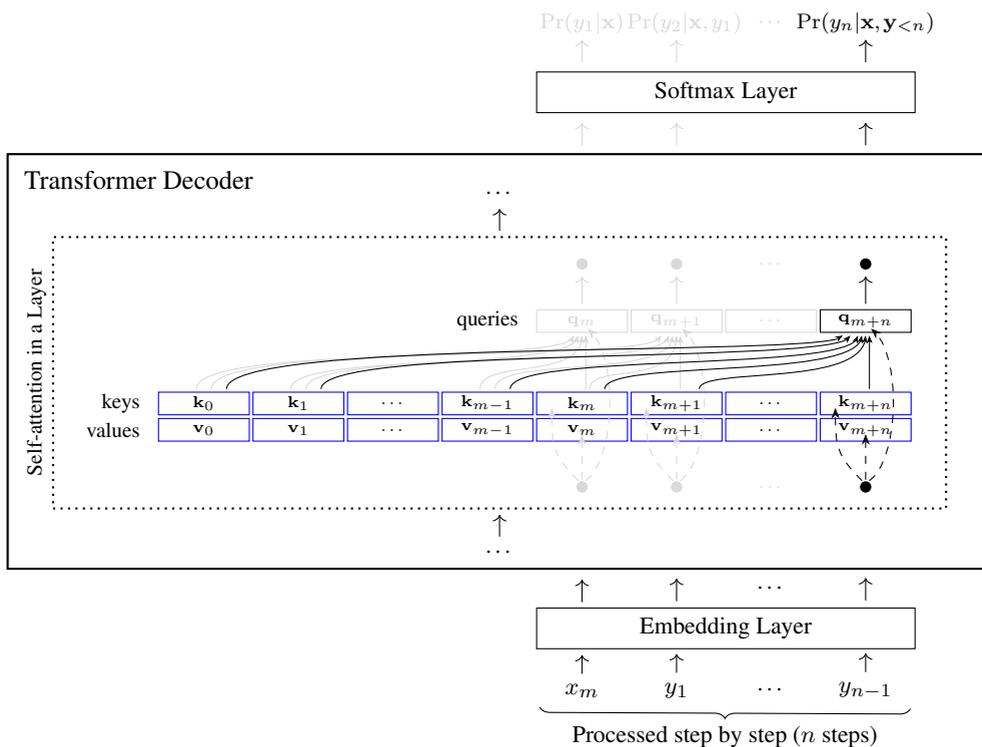
$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \text{Mask}\right)\mathbf{V} \quad (11.13)$$

where  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{d \times (m+1)}$ .  $\text{Mask} \in \mathbb{R}^{(m+1) \times (m+1)}$  is a mask that ensures that each token only attends to itself and the tokens that precede it in the sequence. It is represented by setting the values in the mask corresponding to future tokens to a large negative number, for example, for the query  $\mathbf{q}_i$  and the key  $\mathbf{k}_j$ , we set the value of the entry  $(i, j)$  to  $-\infty$  if  $i < j$ . One advantage of processing the sequence with a single self-attention computation is that we can make better use of the parallel computing capabilities of modern GPUs, and so speed up prefilling. In general, the prefilling process is considered compute-bound. This is because merging multiple computational operations into one operation reduces the number of data transfers and the performance bottleneck usually comes from the computational capacity rather than memory bandwidth.

Decoding is a standard left-to-right text generation process. The token sequence is generated autoregressively by predicting one token at a time based on the KV cache. Each time a new token is generated, we need to attend it to previous tokens, following Eq. (11.7). Therefore, the decoding process is memory-bound due to its frequent access to the KV cache. The cost of decoding grows significantly as more tokens are generated. In most cases, decoding is computationally more expensive than prefilling. Note that this is not just because, in decoding, the LLM generates tokens one by one and repeatedly updates the KV cache. As we will see in the following subsection, we may need to explore multiple different token sequences during decoding, which makes the problem more complex and increases its cost further.



(a) Prefilling



(b) Decoding (at the  $n$ -th step)

Figure 11.3: Illustration of the prefilling and decoding processes. In prefilling, the entire input sequence is processed together and the KV cache is filled. In decoding, the LLM generates the output sequence step by step based on the prefilled KV cache.

### 11.1.3 Decoding Algorithms

So far our discussion of LLM inference has primarily focused on the model computation problem, that is, how to compute  $\Pr(\mathbf{y}|\mathbf{x})$ . Now we turn to the discussion of the search problem. The problem can be stated as: given an LLM  $\Pr(\mathbf{y}|\mathbf{x})$ , how do we efficiently search for the best output sequence  $\hat{\mathbf{y}}$  given the input sequence  $\mathbf{x}$  (or the generated KV cache)? Naively, we can consider all of the output sequences, compute the prediction probability for each, and then select the output sequence having the highest probability. This method can guarantee the globally optimal solution, but direct exhaustive search is impractical for LLMs as the number of possible output sequences grows exponentially with the length of  $\mathbf{y}$ .

In practice, various heuristic search algorithms, such as greedy search and sampling-based search, are commonly employed to approximate the solution. Each of these methods offers trade-offs between search quality and computational efficiency. The search problem, therefore, becomes a balancing act between exploration and exploitation, where the goal is to find an efficient strategy that produces high-quality outputs without exploring the entire space.

Before giving a more detailed discussion of these methods, let us first informally define what a search space is and how it is represented. In LLM inference, we define a hypothesis as a tuple of input and output sequences. Since  $\mathbf{x}$  is fixed during inference, we can simply consider each hypothesis as an output sequence. The search space, denoted by  $\mathcal{Y}$ , is then the set of all possible hypotheses (i.e., output sequences) that the model can generate. The search problem for LLM inference can be re-expressed as

$$\hat{\mathbf{y}} = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{arg\,max}} \Pr(\mathbf{y}|\mathbf{x}) \quad (11.14)$$

In NLP,  $\mathcal{Y}$  is commonly represented in a tree data structure to facilitate search. Figure 11.4 shows an example of the search tree resulting from a small vocabulary. In this example, a node represents a prefix subsequence that can be shared by many sequences. The search starts with the root of the tree, which can be regarded as the beginning of all sequences that can be generated<sup>1</sup>. Each child node extends the prefix of its parent node by adding one token from the vocabulary to the sequence, along with the probability of predicting the token given the prefix. This process continues as each node further branches out into additional child nodes, each representing a new possible extension of the sequence with another token. The search tree thus grows deeper and wider, representing an ever-increasing number of potential sequences as more tokens are appended. This structure allows us to efficiently traverse through possible sequences, evaluating each in terms of the log-probability accumulated over the path from the root to that node. For example, in Figure 11.4, the path from the root to the node 17 corresponds to the output sequence “*Cats are playful.*”. The prediction log-probability  $\log \Pr(\mathbf{y}|\mathbf{x})$  is the sum of the log-probabilities of all the nodes on this path.

In general, the search tree is organized as levels, where each level consists of all nodes that are the same distance from the root node. Thus, a breadth-first search over the tree essentially performs left-to-right generation of tokens. Nodes in the same level correspond to sequences

<sup>1</sup>Here, since the predictions in LLMs are based on  $\mathbf{x}$ , we can think of the root as a representation of  $\mathbf{x}$ .

Path: node 0 → node 3 → node 9 → node 11 → node 17

Output: cats are playful.

Probability:

node 0 → 0

node 3 →  $\log \Pr(\text{"cats"}|\mathbf{x})$

node 9 →  $\log \Pr(\text{"are"}|\mathbf{x}, \text{"cats"})$

node 11 →  $\log \Pr(\text{"playful"}|\mathbf{x}, \text{"cats are"})$

node 17 →  $\log \Pr(\text{"."}|\mathbf{x}, \text{"cats are playful"})$

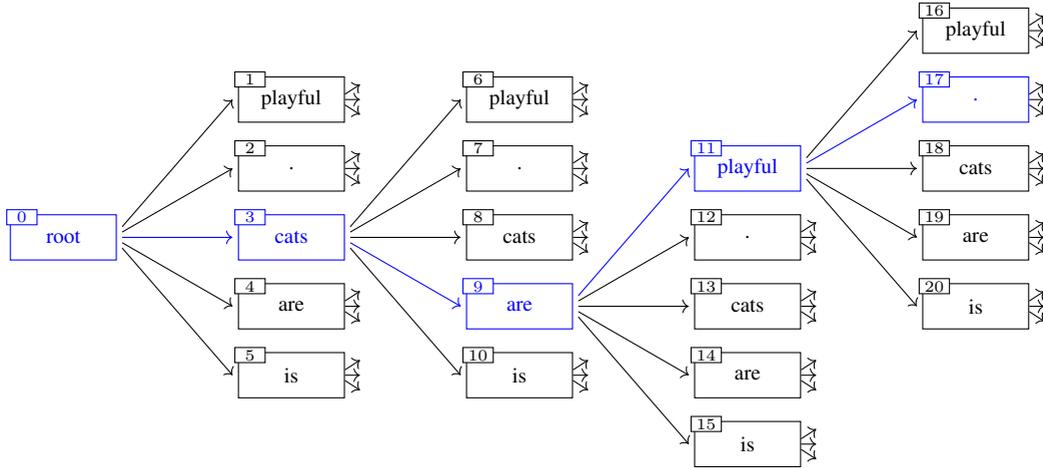


Figure 11.4: A search tree for decoding. At each node, we expand the tree by considering all possible tokens, each leading to a new node representing a potential continuation of the text. Here we highlight a path through nodes 0, 3, 9, 11, and 17. The path represents the output sequence “cats are playful.”, whose log-probability can be computed by accumulating the log-probabilities of these nodes.

of the same length. As the search progresses, new tokens are appended to these sequences, expanding them incrementally.

Let  $Y_i$  be the set of the sequences that the LLM generates at step  $i$ .  $Y_i$  can be obtained by expanding each sequence in  $Y_{i-1}$  with all possible next tokens in the vocabulary  $V$ , given in the following recursive form

$$Y_i = Y_{i-1} \times V \tag{11.15}$$

where  $Y_{i-1} \times V$  denotes the Cartesian product of  $Y_{i-1}$  and  $V$  (i.e., each sequence in  $Y_{i-1}$  is concatenated with each token in  $V$ ). Note that if a sequence in  $Y_{i-1}$  is complete (e.g., ending with the  $\langle \text{EOS} \rangle$  token), it will not be expanded any further. Let  $\Psi(Y_i)$  be the set of all complete sequences in  $Y_i$ . Then, the search space can be expressed as

$$\mathcal{Y} = \Psi(Y_1) \cup \Psi(Y_2) \cup \dots \cup \Psi(Y_{n_{\max}}) \tag{11.16}$$

where  $n_{\max}$  is the maximum length of a sequence.

Most decoding algorithms follow this level-by-level search process. However,  $\mathcal{Y}$  consists

of an exponentially large number of sequences, and a direct search in such a vast space is computationally infeasible. Therefore, practical decoding algorithms often rely on strategies to prune the search space and avoid exploring low-quality sequences. For example, at each decoding step,  $Y_i$  can be obtained in the following way

$$Y_i = \text{Prune}(Y_{i-1} \times V) \quad (11.17)$$

where  $\text{Prune}(\cdot)$  is a function that selectively removes sequences less likely to result in high-quality outcomes. In general, we expect that  $|Y_i| \ll |Y_{i-1}| \cdot |V|$ . Thus we can drastically reduce the number of sequences under consideration at each step, ensuring that the computational load does not grow exponentially with the sequence length.

Next, we will introduce these decoding algorithms. Some of them have already been discussed in sequence-to-sequence models (see Chapter 5), while others are more commonly used in LLMs.

## 1. Greedy Decoding

Greedy search (or greedy decoding) is one of the most widely used decoding methods in NLP, particularly in text generation tasks like machine translation. The idea behind greedy search is straightforward: at each step in generation, it selects the next token that has the highest prediction probability. For each sequence  $\mathbf{y} = y_1 \dots y_i \in Y_{i-1} \times V$ , we can evaluate it using  $\log \Pr(\mathbf{y}|\mathbf{x})$ . This log-probability can be easily computed by noting that

$$\begin{aligned} \log \Pr(\mathbf{y}|\mathbf{x}) &= \log \Pr(y_1 \dots y_i|\mathbf{x}) \\ &= \underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{accumulated up to the parent node}} + \underbrace{\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}_{\text{newly computed for the current node}} \end{aligned} \quad (11.18)$$

Here the first term is the sum of the log-probabilities of the path from the root to the parent node, which has been computed in the previous decoding steps. At step  $i$ , we only need to compute the second term which is the standard token prediction log-probability produced by the LLM.

The “best” token at step  $i$  is then chosen as

$$\begin{aligned} y_i^{\text{top1}} &= \arg \max_{y_i \in V} \log \Pr(y_1 \dots y_i|\mathbf{x}) \\ &= \arg \max_{y_i \in V} \left[ \underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{fixed wrt. } y_i} + \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \right] \\ &= \arg \max_{y_i \in V} \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \end{aligned} \quad (11.19)$$

Thus, the “best” sequence generated up to step  $i$  is given by

$$\mathbf{y}^{\text{top1}} = y_1 \dots y_{i-1} y_i^{\text{top1}} \quad (11.20)$$

Finally,  $Y_i$  contains only this sequence

$$Y_i = \{\mathbf{y}^{\text{top1}}\} \quad (11.21)$$

The greedy choice in one decoding step is illustrated in Figure 11.5 (a). Greedy search offers computational efficiency and simplicity in implementation for LLM inference. Its primary disadvantage, however, lies in its suboptimal nature — high-quality sequences are likely pruned at early stages of decoding. Therefore, greedy search is appealing for tasks that demand speed and simplicity. For tasks that require better search results, alternative strategies such as beam search, which explores multiple potential paths simultaneously, are preferable.

## 2. Beam Decoding

Beam search (or beam decoding) is a natural extension of greedy search. Instead of selecting the single most probable token at each step, beam search maintains a fixed number of the best candidates at each step, known as the “beam width”. See Figure 11.5 (b) for an illustration of beam search.

Let  $K$  be the beam width. Given a parent node, which corresponds to the prefix  $y_1 \dots y_{i-1}$ , we can select the top- $K$  next tokens by

$$\{y_i^{\text{top1}}, \dots, y_i^{\text{topK}}\} = \underset{y_i \in V}{\text{argTopK}} \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \quad (11.22)$$

where  $\text{argTopK}$  is a function that ranks the prediction probabilities of all possible next tokens and selects the top  $K$  candidates. Given these tokens, the top- $K$  sequences for step  $i$  are given by

$$\mathbf{y}^{\text{top1}} = y_1 \dots y_{i-1} y_i^{\text{top1}} \quad (11.23)$$

$$\vdots$$

$$\mathbf{y}^{\text{topK}} = y_1 \dots y_{i-1} y_i^{\text{topK}} \quad (11.24)$$

Then, we can define  $Y_i$  as

$$Y_i = \{\mathbf{y}^{\text{top1}}, \dots, \mathbf{y}^{\text{topK}}\} \quad (11.25)$$

We can adjust the beam width  $K$  to balance search efficiency and accuracy. But a very large beam width might not be helpful. In many practical applications, selecting a relatively small number for  $K$ , such as  $K = 2$  or  $K = 4$ , is often sufficient to achieve satisfactory performance in LLM inference.

## 3. Sampling-based Decoding

Both greedy and beam search generate deterministic outputs, that is, given an LLM, the output of the model will always be the same every time it processes the same input. The deterministic nature of greedy and beam search ensures predictability and reliability in applications where

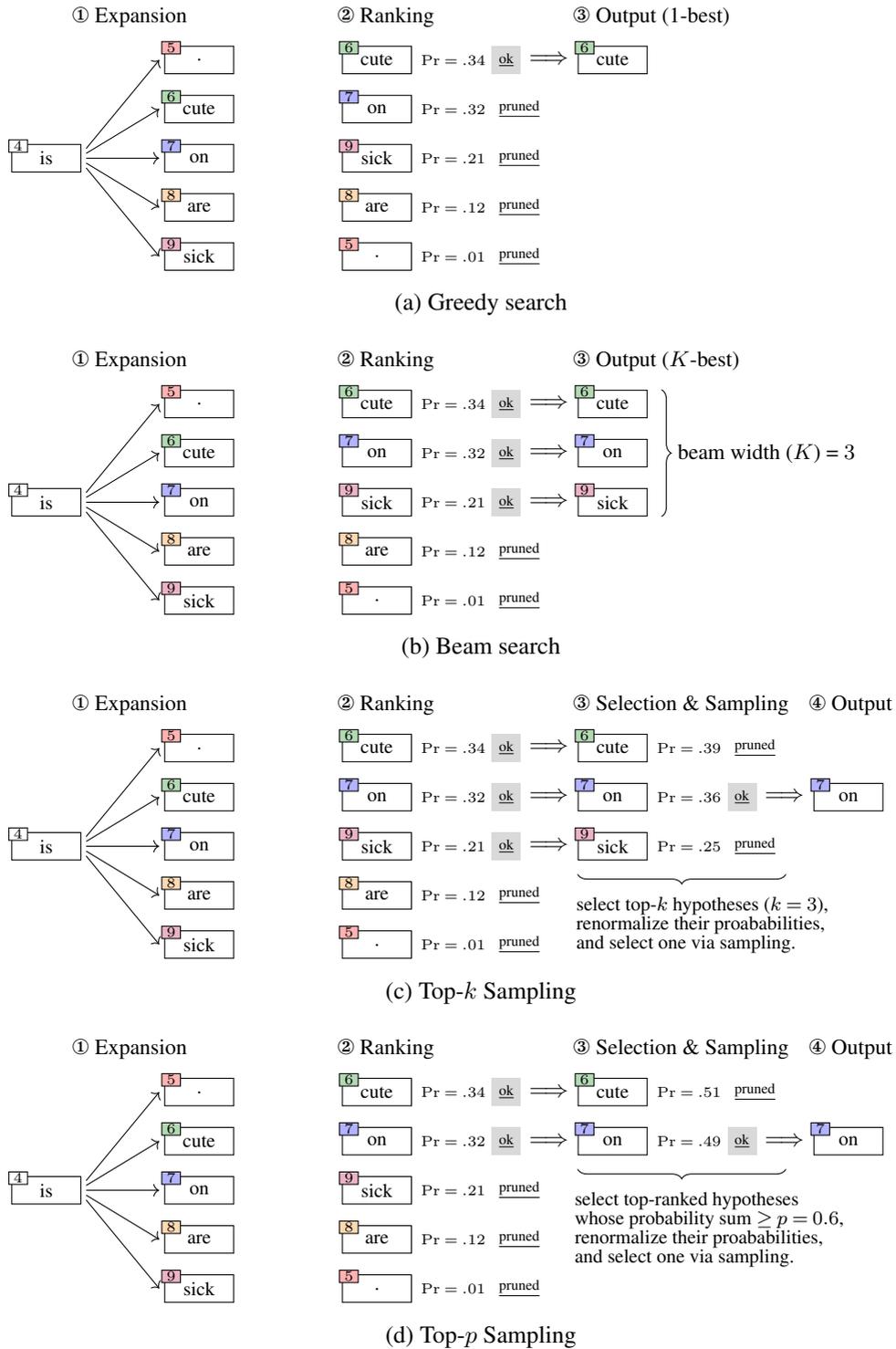


Figure 11.5: Illustrations of greedy decoding, beam decoding, top- $k$  decoding and top- $p$  decoding methods (in one decoding step).

consistent outcomes are critical, such as in formal document generation, where varying outputs could cause confusion or errors. On the other hand, one disadvantage of these methods is the lack of diversity and flexibility. For example, in creative tasks like story generation or conversational agents, generic or repetitive outputs generated by deterministic systems are often less engaging.

To add variation into LLM outputs, we can use sampling-based decoding methods. There are two commonly used methods.

- **Top- $k$  Sampling.** This method selects the next token from the top- $k$  most likely candidates at each step of the generation process [Fan et al., 2018]. Let  $\bar{V}_i$  be the selection pool for top- $k$  sampling. We can define it as

$$\bar{V}_i = \{y_i^{\text{top}1}, \dots, y_i^{\text{top}k}\} \quad (11.26)$$

where  $\{y_i^{\text{top}1}, \dots, y_i^{\text{top}k}\}$  are the top- $k$  tokens selected based on their prediction probabilities (see Eq. (11.22)). Once the selection pool is determined, we recompute the prediction probability distribution over  $\bar{V}_i$ . One of the simplest ways to do this is to renormalize their probabilities:

$$\bar{\Pr}(y_i | \mathbf{x}, \mathbf{y}_{<i}) = \frac{\Pr(y_i | \mathbf{x}, \mathbf{y}_{<i})}{\sum_{y_j \in \bar{V}_i} \Pr(y_j | \mathbf{x}, \mathbf{y}_{<i})} \quad (11.27)$$

Alternatively, we can calculate the distribution by using the Softmax function:

$$\bar{\Pr}(y_i | \mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i})}{\sum_{y_j \in \bar{V}_i} \exp(u_{y_j})} \quad (11.28)$$

where  $u_{y_i}$  is the logit for token  $y_i$ . Then, we sample a token  $\bar{y}_i$  from this distribution:

$$\bar{y}_i \sim \bar{\Pr}(y_i | \mathbf{x}, \mathbf{y}_{<i}) \quad (11.29)$$

The corresponding sequence is  $\bar{\mathbf{y}} = y_1 \dots y_{i-1} \bar{y}_i$ , and  $Y_i$  is given by

$$Y_i = \{\bar{\mathbf{y}}\} \quad (11.30)$$

- **Top- $p$  Sampling.** This sampling method, also known as **nucleus sampling**, follows a procedure similar to that of top- $k$  sampling. Instead of drawing from a fixed size candidate pool, it selects the next token from the smallest set of tokens that together have a cumulative probability higher than a predefined threshold  $p$  [Holtzman et al., 2020b]. In this way we prevent the prediction from choosing from low-probability tokens in the long tail that could lead to incoherent or nonsensical outputs. To obtain the candidate pool in the top- $p$  sampling method, we can sort all tokens by their predicted probabilities. Then, starting with the token with the highest probability, we continue to add tokens to the candidate pool until the cumulative probability of the tokens in the pool reaches or exceeds  $p$  (we denote the size of the candidate pool at this point as  $k_p$ ). The candidate

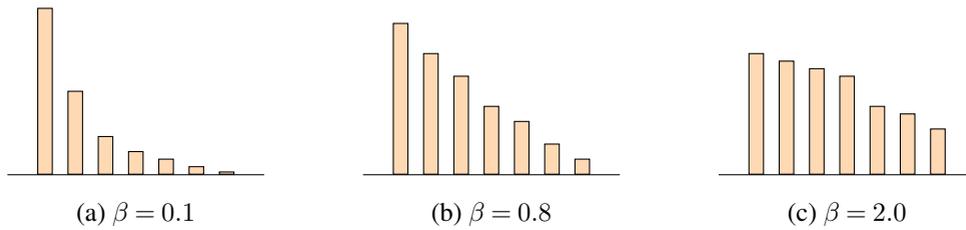


Figure 11.6: Histogram estimates of the distributions generated by the Softmax function with different values of the temperature parameter  $\beta$ .

pool can then be expressed as

$$\bar{V}_i = \{y_i^{\text{top}1}, \dots, y_i^{\text{top}k_p}\} \quad (11.31)$$

The subsequent steps, such as the renormalization of the distribution and sampling, are the same as in the top- $k$  sampling method (see Eqs.(11.27-11.30)).

See Figure 11.5 (c-d) for illustrations of the top- $k$  and top- $p$  sampling methods. By limiting the choices to a smaller set of high-probability tokens, these methods strike a balance between randomness and coherence. They allow for more diverse outputs while still maintaining a reasonable level of relevance and fluency. However, the value of  $k$  or  $p$  must be carefully chosen: if  $k$  or  $p$  is too small, the output may still be overly deterministic (more like greedy decoding), and if  $k$  or  $p$  is too large, the LLM might produce degenerate outputs.

In order to further control the randomness of the token selection process, the renormalized distribution  $\bar{\text{Pr}}(\cdot)$  is typically obtained by using the Softmax function with the temperature parameter, given by

$$\bar{\text{Pr}}(y_i | \mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i}/\beta)}{\sum_{y_j \in \bar{V}_i} \exp(u_{y_j}/\beta)} \quad (11.32)$$

Here  $\beta$  is a temperature parameter  $\beta$  that controls the sharpness of the probability distribution derived from logits. In Figure 11.6, we show simple examples involving distributions generated by the above function with different temperatures. When the temperature is set to a higher value, the resulting probability distribution becomes more uniform, as the differences between the logits are diminished. This means that each token in the candidate pool has a more equal chance of being selected, leading to greater diversity in the generated output. By contrast, when the temperature is set to a lower value, the distribution becomes sharper, making the high-probability tokens even more likely to be chosen, which often results in more deterministic outputs. For example, if we set  $p$  to 1 and  $\beta$  to a very small number (approaching zero), the top- $p$  sampling method will become equivalent to the greedy search method.

#### 4. Decoding with Penalty Terms

One common improvement to decoding methods in text generation is to modify the search objective. For example, one can replace maximum a posteriori (MAP) decoding with minimum

Bayes risk (MBR) decoding [Kumar and Byrne, 2004b], where the focus shifts from selecting the single most probable output to choosing an output that minimizes the expected risk over a distribution of possible outputs. More details on MBR decoding can be found in Chapter 5. Here we explore methods that incorporate penalty terms into decoding. These methods offer a simple but effective way to make decoding more controllable.

Recall from Eq. (11.14) that the goal of decoding is to maximize the likelihood of the output sequence. With penalty terms, the objective is extended to include additional factors that penalize or reward certain behaviors in the generated text. A general form of the new objective is given by

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \mathcal{Y}} [\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})] \quad (11.33)$$

where  $\text{Penalty}(\mathbf{x}, \mathbf{y})$  is a function that quantifies the degree to which the generated sequence  $\mathbf{y}$  violates certain constraints or exhibits undesirable behaviors given the input  $\mathbf{x}$ . The design of  $\text{Penalty}(\cdot)$  is very flexible, thus allowing us to incorporate a wide range of constraints or prior knowledge into it. Below, we present some common types of penalty functions.

- **Repetition Penalty.** A repetition penalty discourages the model from generating repetitive or redundant text. The penalty function might measure the frequency of repeated tokens or phrases in the generated sequence and impose a penalty proportional to their occurrence.
- **Length Penalty.** A length penalty ensures that the generated sequence adheres to a desired length. For example, in text summarization tasks, the penalty function could penalize outputs that are too short or too long.
- **Diversity Penalty.** A diversity penalty promotes variation in the generated text. For example, in beam search, we can measure the similarity between generated hypotheses, and encourage the model to explore different hypotheses.
- **Constraint-based Penalty.** A constraint-based penalty enforces specific constraints related to the content or style of the generated text. For example, in machine translation, the penalty function could penalize outputs that deviate from a desired tone or terminology.

In general, we can consider  $\text{Penalty}(\mathbf{x}, \mathbf{y})$  as a function that defines the cost of generating the surface form of the output sequence  $\mathbf{y}$  given the input sequence  $\mathbf{x}$ . Alternatively, this function can be defined to assess the hidden states of an LLM when generating  $\mathbf{y}$ . For example, Su et al. [2022] develop a penalty term that calculates the maximum distance between the representation of the predicted token and the representations of the previously generated tokens. Therefore, the search objective will penalize degenerated outputs, such as texts with many repetitions.

The method described in Eq. (11.33) is general and can be easily adapted to different search algorithms. For example, in greedy search, we can keep the single sequence that maximizes  $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$  at each decoding step; in sampling-based search, we can rank and

select the top-ranked sequences based on  $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$  to form the candidate pool.

## 5. Speculative Decoding

**Speculative decoding** stems from the concept of **speculative execution**, where a system makes educated guesses about future actions and performs them in advance. If the guess is correct, the results are immediately available, which speeds up processing. In the case of LLM inference, suppose we have two models. One is a smaller, faster model (called draft model), and the other is the full, more accurate model (called verification model). These two models represent two baselines in LLM inference: the draft model is efficient but not very accurate; the verification model is usually the one we want to run, but it is very slow. Given a prefix, we first use the draft model to speculatively predict a sequence of likely future tokens. This is a standard autoregressive decoding process, but it is still fast in practice due to the high efficiency of the draft model. Then, the verification model evaluates the speculated tokens in parallel. It checks whether the predicted tokens are correct or need to be adjusted. Note that, since we can deal with these tokens all at once, the verification can be done in a single step for all the tokens simultaneously, rather than in a token-by-token manner. If the speculated tokens are correct, they are accepted, and the process continues with the next set of tokens. If they are incorrect, the incorrect speculations are discarded, and the verification model is used to generate the correct tokens.

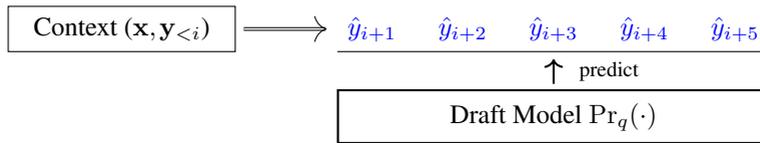
To be more specific, let us see the speculative decoding method presented in [Leviathan et al. \[2023\]](#)'s work. In this method, the draft model is a small language model, denoted by  $\Pr_q(y_i|\mathbf{x}, \mathbf{y}_{<i})$ , while the verification model is a normal LLM, denoted by  $\Pr_p(y_i|\mathbf{x}, \mathbf{y}_{<i})$ . The goal is that, given a prefix, we use the draft model to autoregressively predict up to  $\tau$  tokens. The verification model is then employed to generate the last token at the point where errors begin to occur in the speculative predictions. Figure 11.7 illustrates one step in this decoding process.

The speculative decoding algorithm can be summarized as follows.

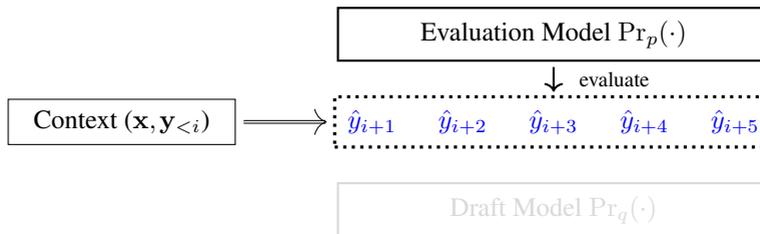
- Given the prefix  $[\mathbf{x}, \mathbf{y}_{\leq i}]$ , we use the draft model to predict the next  $\tau$  consecutive tokens, denoted by  $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+\tau}\}$ . This is a token-by-token generation process, given by

$$\hat{y}_{i+t} = \arg \max_{y_{i+t}} \Pr_q(y_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1}) \quad (11.34)$$

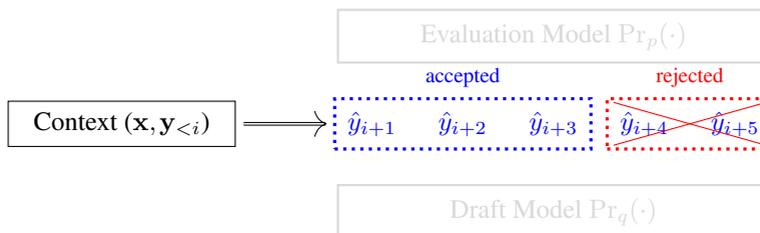
- We evaluate  $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+\tau}\}$  using the verification model, that is, we compute  $\{\Pr_p(\hat{y}_{i+1}|\mathbf{x}, \mathbf{y}_{\leq i}), \dots, \Pr_p(\hat{y}_{i+\tau}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+\tau-1})\}$ . Note that we can compute these probabilities in parallel, and so this verification step is efficient.
- We determine the maximum number of accepted speculated tokens. In order to keep the notation uncluttered, we denote  $\Pr_q(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1})$  and  $\Pr_p(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1})$  simply by  $q(\hat{y}_{i+t})$  and  $p(\hat{y}_{i+t})$ , respectively. We then define that, if  $q(\hat{y}_{i+t}) \leq p(\hat{y}_{i+t})$ , then we accept this speculation. By contrast, if  $q(\hat{y}_{i+t}) > p(\hat{y}_{i+t})$ , we reject this speculation with probability  $1 - \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}$ . Starting from  $\hat{y}_{i+1}$ , the maximum number of accepted



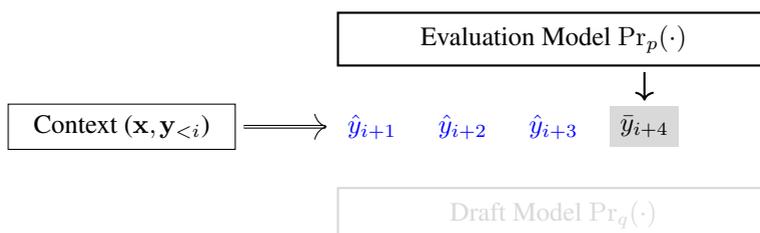
(a) Predict the next  $\tau$  tokens given the context using the draft model ( $\tau = 5$ )



(b) Evaluate the predicted tokens using the evaluation model



(c) Determine the number of accepted tokens



(d) Predict a new token following the accepted tokens using the evaluation model

Figure 11.7: Illustration of one step of speculative decoding. The goal is to predict as many next tokens as possible using the draft model. There are four sub-steps. Given the context, we first use the draft model to predict the next  $\tau$  tokens. Then, we evaluate these predictions in parallel using the evaluation model. Next, we determine the maximum number of predicted tokens that can be accepted. Finally, we use the evaluation model to predict a new token following these accepted tokens.

consecutive speculated tokens is defined as

$$n_a = \min \left\{ t-1 \mid 1 \leq t \leq \tau, r_t > \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})} \right\} \quad (11.35)$$

where  $r_t$  is a variable drawn from the uniform distribution  $U(0, 1)$ .

- Given  $n_a$ , we keep the speculated tokens  $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}\}$ . We then use the verification model to make a new prediction at  $i + n_a + 1$

$$\bar{y}_{i+n_a+1} = \arg \max_{y_{i+n_s+1}} \Pr_p(y_{i+n_s+1} \mid \mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+n_s}) \quad (11.36)$$

- Above, we have described one step of speculative decoding. The result sequence (including both the context and predicted tokens) is illustrated as follows

$$\underbrace{[\mathbf{x}, \mathbf{y}_{<i}]}_{\text{Context}} \quad \underbrace{\hat{y}_{i+1} \dots \hat{y}_{i+n_a}}_{\substack{n_a \text{ tokens} \\ \text{predicted using} \\ \text{the draft model}}} \quad \underbrace{\bar{y}_{i+n_a+1}}_{\substack{\text{One token} \\ \text{predicted using} \\ \text{the verification model}}}$$

Once we have finished this step, we add the predicted tokens  $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}, \bar{y}_{i+n_a+1}\}$  to the context, and repeat the above process.

In practice, we usually wish to use a smaller draft model so that predicting  $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}\}$  would be computationally cheaper. But a very small draft model is less accurate and can result in smaller  $n_a$ . We therefore need to carefully select the draft model to make the trade-off between the computational efficiency and accuracy.

## 6. Stopping Criteria

Stopping criteria are a critical component of LLM inference. They typically involve rules or conditions that specify when the model should stop generating text during decoding. Most LLMs are trained to generate an end-of-sequence token (e.g.,  $\langle \text{EOS} \rangle$  or  $\langle /s \rangle$ ) to signal the end of the generated text. So one of the simplest strategies is that the generation process stops when this token is produced. For beam search, which explores multiple hypotheses simultaneously, the process can continue until a given number of complete sequences have been generated.

In practical applications, it will generally be undesirable to generate very long sequences, and so we need to reduce the decoding cost and unnecessary verbosity. One commonly-used stopping criterion is the maximum length of the output. The model stops generating text once it has produced a predetermined number of tokens. Alternatively, we can stop the decoding based on the real cost, such as the computational resources or time constraints. For example, in real-time applications like chatbots, decoding may need to stop after a certain time limit to ensure responsiveness.

Another approach is to design stopping criteria based on the behavior of LLMs. For example, decoding can be stopped if the probability of predicting the next token falls below a certain threshold. In addition to probability-based stopping, a repetition detection module can

be implemented to trigger the model to stop if it begins repeating tokens or phrases beyond a predefined limit. This helps prevent redundant or incoherent outputs.

#### 11.1.4 Evaluation Metrics for LLM Inference

Evaluating the performance of LLMs during inference involves a variety of metrics to assess how well these models meet desired standards, such as accuracy, robustness, usability, and efficiency. As with most NLP systems, we can evaluate LLMs using accuracy-based metrics, such as perplexity and F1 score. We can also examine their robustness by testing how well they handle ambiguous or challenging inputs, including adversarial, perturbed, or out-of-distribution data. Additionally, usability can be assessed by measuring how well the generated outputs align with user expectations in terms of fluency, coherence, relevance, and diversity. Human evaluators can rate the naturalness of the text or assess whether the responses are contextually appropriate and logically consistent. Ethical and fairness metrics can also be included to ensure LLMs avoid perpetuating biases or generating harmful content.

All of the evaluation metrics mentioned above essentially focus on assessing the quality of the outputs. Given the high cost of deploying and applying LLMs, efficiency metrics are also very important for practitioners. Below are some commonly used efficiency metrics [[Nvidia, 2025](#)]:

- **Request Latency.** This metric measures the total time taken from when a request is sent to the LLM until the complete response is received. This includes the time taken for data transmission, processing by the model, and the return of the output to the user.
- **Throughput.** It refers to the number of tokens or requests the model can process per second.
- **Time to First Token (TTFT).** This metric measures the time it takes from the beginning of a request being sent to the generation of the first token of the response. If data transmission does not consume too much time, then TTFT is mainly the time for prefilling and predicting the first token.
- **Inter-token Latency (ITL).** This metric refers to the time taken to generate each subsequent token after the first one. It reflects the efficiency of the decoding process.
- **Tokens Per Second (TPS).** This metric quantifies the number of tokens that the model can generate per second.
- **Resource Utilization.** This involves measuring the computational resource usage (e.g., CPU and GPU utilization) and memory consumption of the model during inference.

In addition to these metrics, energy efficiency and cost efficiency are practical considerations for deploying LLMs at scale. Energy efficiency measures the amount of electrical power consumed by the model during inference. Cost efficiency, on the other hand, evaluates the total expenses related to deploying and maintaining the model.

In general, choosing the right evaluation metrics depends on the specific task and application. While quality-focused metrics are essential for assessing LLMs, efficiency metrics are equally crucial for their effective deployment in real-world applications. A comprehensive

evaluation framework should include both sets of metrics to accurately estimate an LLM’s performance and practicality.

## 11.2 Efficient Inference Techniques

In practical applications, we often wish a system to be as efficient as possible. For LLM inference, this typically involves two types of improvements: reducing memory requirements and accelerating the system. For example, we can modify the Transformer architecture to avoid memory explosion when processing very long input sequences. Another example is that we can compress input sequences to reduce computational overhead while preserving their semantic information. In addition, techniques like quantization and pruning can be employed to further optimize memory usage and inference speed.

Efficient inference is a wide-ranging topic that overlaps with several sub-fields of LLMs, such as architecture design and model compression. Most of these topics have been covered in previous chapters. For example, in Chapter 6, we discussed efficient Transformer architectures; in Chapter 8, we discussed long-context LLMs; and in Chapter 9, we discussed prompt compression methods for reducing prompt length. In this section, we focus on techniques that are commonly used in LLM deployment and serving.

### 11.2.1 More Caching

In real-world applications, it is common practice to store frequent requests and their corresponding responses in a cache. When a new request hits the cache, the system can retrieve the response directly from the cache instead of recomputing the result. One straightforward implementation is a key-value datastore (e.g., a hash table) that maps input sequences to their LLM-generated output sequences. In the simplest case, we can collect frequent queries, generate their responses using the LLM, and store these query-response pairs in the datastore. This creates a basic sequence-level caching mechanism that allows the system to bypass LLM computation when the input sequence exactly matches a cached query.

A straightforward extension of the caching mechanism is to cache prefixes and their corresponding hidden states. Given an input sequence  $\mathbf{x}$  in a dataset  $\mathcal{D}$ , we can process it as in the standard prefilling phase. Thus, we obtain a sequence of prefixes and their corresponding KV cache states:

$$\begin{aligned} x_0 (\mathbf{x}_{<1}) &\Rightarrow \text{cache}_{<1} \\ x_0x_1 (\mathbf{x}_{<2}) &\Rightarrow \text{cache}_{<2} \\ &\dots \\ x_0x_1\dots x_{m-1} (\mathbf{x}_{<m}) &\Rightarrow \text{cache}_{<m} \end{aligned}$$

where  $\text{cache}_{<i}$  denotes the KV cache for the prefix  $\mathbf{x}_{<i}$  (see also Eq. (11.10)). All these mappings can be stored in the prefix cache for efficient reuse.

When processing a new sequence that shares a common prefix with a previously seen sequence in  $\mathcal{D}$ , we can load the corresponding cached hidden states instead of recomputing

them. Specifically, if a new input  $\mathbf{x}'$  has  $\mathbf{x}_{<k}$  (i.e.,  $\mathbf{x}'_{<k} = \mathbf{x}_{<k}$  for some  $k \leq m$ ), we can initialize the KV cache with  $\text{cache}_{<k}$  and only compute the hidden states for the remaining tokens  $\mathbf{x}'_{\geq k}$ .

As usual, we can maintain a key-value datastore that maps frequently encountered prefixes to their precomputed KV caches. The lookup can be performed using a hash of the prefix tokens, allowing constant-time access to the cached states. Care must be taken to manage memory usage, as storing all possible prefixes may be infeasible for large datasets. Practical systems often employ least recently used (LRU) caching methods or other strategies to balance between computational savings and memory constraints.

### 11.2.2 Batching

Batching in LLM inference refers to the process of processing multiple input sequences simultaneously as a group (called a batch) rather than one at a time. Because modern GPUs excel at parallel processing, batching allows them to compute multiple sequences in a single forward pass, keeping the hardware fully occupied. Therefore, when serving LLMs at scale, batching is important for improving computational efficiency and maximizing hardware utilization<sup>2</sup>.

To illustrate the idea of batching, Figure 11.8 (a-b) show simple examples with batch sizes of 1 and 4, respectively. When using a batch size of 1 (i.e., without batching), the GPU processes one input sequence at a time. Thus, the processing is sequential: the next sequence must wait for the current computation to finish. By contrast, when using a batch size of 4, the GPU can process four sequences simultaneously in a single forward pass. As the input sequences vary in length, we need to standardize their length using padding techniques. Here we use left padding, which adds dummy tokens to the beginnings of short sequences, so all the sequences in the batch would have the same length for prefilling. For decoding, tokens are generated simultaneously for all these sequences, and the generation process continues until the longest sequence reaches completion.

The above examples imply a trade-off between throughput and latency, which is a very important consideration in designing and implementing LLM inference systems. If we choose a smaller batch size, the latency would be lower, as fewer tokens need to be processed in a single run of inference. Imagine that we have only one sequence. The result becomes available immediately after generation completes, with no additional computational overhead. However, this low-latency advantage comes at the cost of underutilizing parallel computing resources, as the parallelism of GPUs remains largely idle during sequential processing. On the other hand, if we use a larger batch, we can make better use of the parallelism, as GPUs can be occupied by large-scale matrix computations. As a result, we can process more tokens in the same period of time and the throughput is improved. However, since the result is obtained only when the last token in the batch is predicted, the latency would be higher.

In practice, we usually prefer to use a slightly larger batch, but try to fill the batch with sequences of similar lengths to reduce the number of padding tokens and improve device utilization. For example, we can group the incoming user requests in a short period of time into

<sup>2</sup>See

<https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf> for a simple evaluation.

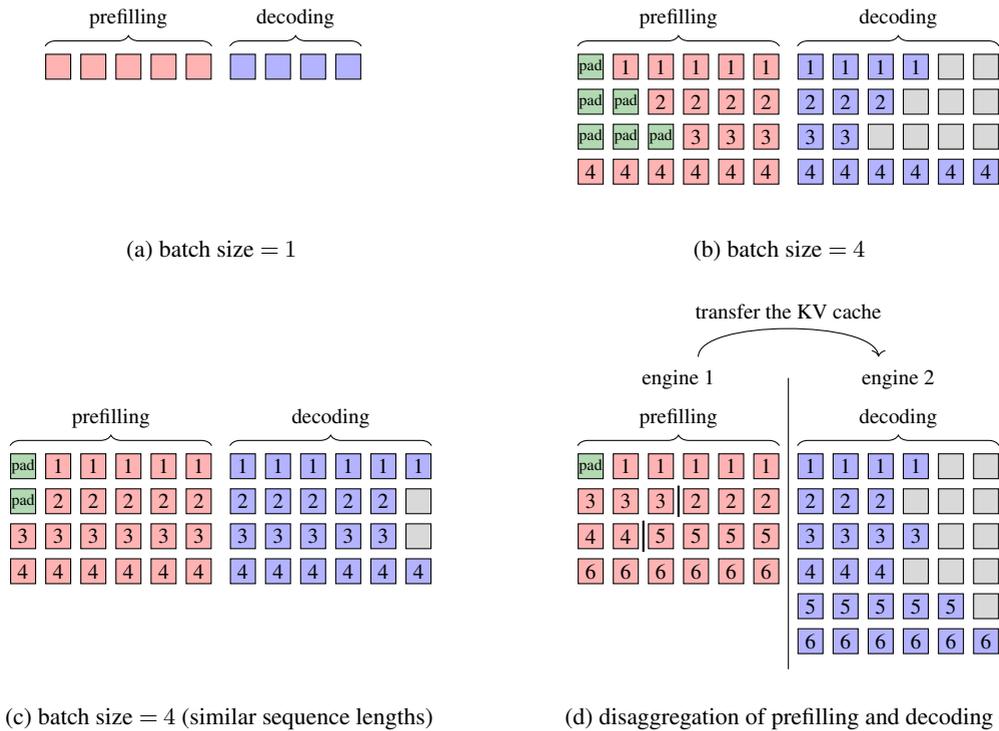


Figure 11.8: Illustrations of basic batching methods. We use a 2D layout to illustrate the batch, where each square represents a token. Red squares indicate tokens in the prefilling stage, blue squares represent tokens in the decoding stage, green squares denote padding tokens, and gray squares correspond to meaningless tokens. Subfigures (a) and (b) compare the cases where the batch size is 1 and 4, respectively. Subfigure (c) shows the strategy of grouping sequences with similar lengths into the same batch. Subfigure (d) illustrates the disaggregation of prefilling and decoding. In this approach, we can make better use of the parallelism of GPUs by concatenating multiple short sequences into a single long sequence for joint processing. This allows us to maximize the number of tokens processed in a batch while minimizing the number of padding tokens. However, as a trade-off, we need to copy the KV cache to the decoding engine and reorganize it after the prefilling phase, which introduces additional data transfer overhead.

buckets, each of which contains sequences with similar lengths. Then, we can fill the batch with sequences in the same bucket, so that we can minimize wasted computational resources, as illustrated in Figure 11.8 (c).

Another approach to implementing batching in LLMs is to disaggregate the prefilling and decoding processes [Wu et al., 2023a; Patel et al., 2024; Zhong et al., 2024]. For example, we can perform prefilling on one GPU, and perform decoding on another GPU. One advantage of disaggregation is that we can rearrange the input sequences in the batch to better fill it, because there is no interference between prefilling and decoding. For example, we can concatenate multiple short sequences into a longer one, thus ensuring that the lengths of sequences in the batch are as consistent as possible, as illustrated in Figure 11.8 (d). In this way, we can

maximize the throughput of the prefilling phase. However, as a trade-off, we need to transfer the KV cache to the devices performing decoding, which also incurs extra communication overhead. Typically, this method requires a high-bandwidth, low-latency network to achieve optimal performance.

In this section, we will discuss several improvements to the above basic batching strategies. Most of them are based on an aggregated architecture, that is, decoding and prefilling can be considered as different stages of a model executed on the same device.

## 1. Scheduling

A practical LLM inference system typically consists of two components:

- **Scheduler.** Its primary role is to efficiently queue and dispatch tasks (i.e., input sequences) to the inference engine based on the current system load and task priorities. This often involves a variety of batching strategies that group certain requests together to maximize processing efficiency in some way.
- **Inference Engine.** It is responsible for the actual execution of the LLMs, processing the queued requests as they come in. As discussed previously, this engine involves both prefilling and decoding processes.

This architecture is illustrated in Figure 11.9. Incorporating scheduling into batch processing provides a flexible way to optimize both the system’s throughput and latency, thereby achieving a better balance between them. For example, the batching methods shown in Figure 11.8 (a) and (b) can be considered one of the simplest scheduling strategies, called **request-level scheduling**. In this strategy, once a batch is filled and sent to the engine, the processing of the entire batch cannot be interrupted. The scheduler waits for this batch to be processed before handling the next batch [Timonin et al., 2022].

A more sophisticated scheduling strategy, called **iteration-based scheduling**, interacts with the inference engine at each token prediction step rather than at the sequence level. This approach allows dynamic batch adjustment during inference, as illustrated in Figure 11.10. Such fine-grained control lets the system prioritize critical tokens or sequences in real-time. For instance, if an urgent request arrives at some decoding step, the scheduler can add this request into the batch so that it can be processed as early as possible. In the following subsections, we will discuss batching methods based on iteration-based scheduling.

## 2. Continuous Batching

**Continuous batching** is an iteration-based scheduling method used in the Orca system [Yu et al., 2022]. In this method, an iteration refers to either the entire prefilling procedure or a single decoding step. For example, given an input sequence  $\mathbf{x} = x_0 \dots x_m$  and an output sequence  $\mathbf{y} = y_1 \dots y_n$ , there are  $n + 1$  iterations in total: one for prefilling, and  $n$  for generating the output tokens (one per token). During scheduling, the batch can be adjusted between iterations. For example, we can either add a new input sequence to the batch, or remove a complete sequence from the batch at some iteration, even if the batch processing is not yet

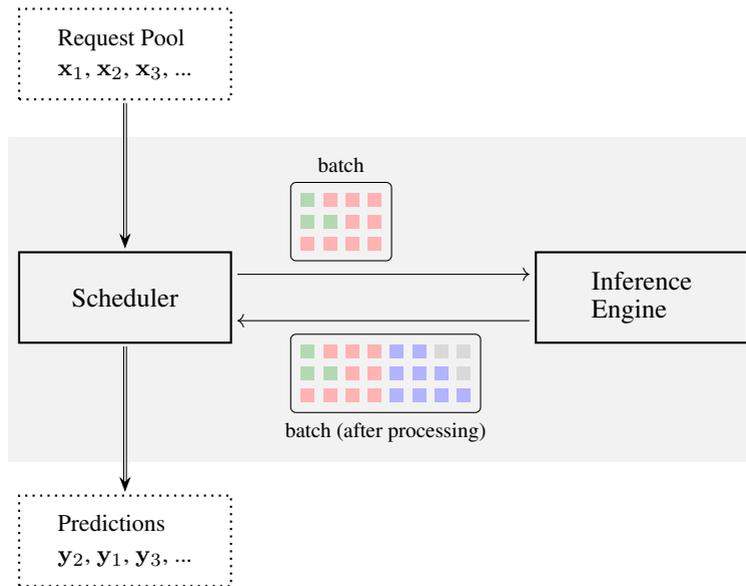


Figure 11.9: Illustration of the LLM inference architecture involving a scheduler and an inference engine. Each time, the scheduler selects a number of user requests to form a batch and sends it to the inference engine. The scheduler can interact with the inference engine and adjust the batch at certain points during inference, such as at the beginning of batch processing and at the start of each token prediction.

finished.

The general process of continuous batching includes the following steps:

- Initially, a batch is created with one or more input sequences, based on both the inference engine's processing capacity and the current user requests. The batch is then fed into the inference engine.
- The inference engine processes the batch iteration by iteration. After each iteration, the scheduler may adjust the batch in one of the following ways:
  - If a sequence in the batch completes generation (i.e., generates the end-of-sequence symbol), that sequence is removed from the batch.
  - If a new user request arrives and the inference engine has additional processing capacity, it is added to the batch.
  - If no sequences are added to or removed from the batch, the batch remains unchanged.
- The processing terminates only when all sequences have been completed and no new user requests arrive.

See Figure 11.11 for an example of continuous batching. In this example, we start with two user requests,  $x_1$  and  $x_2$ . These two sequences are packed into a batch and sent to the inference engine for processing. After the engine completes two iterations, a new user request,  $x_3$ , arrives. At this point, the scheduler adjusts the batch by adding  $x_3$  to it. The inference engine

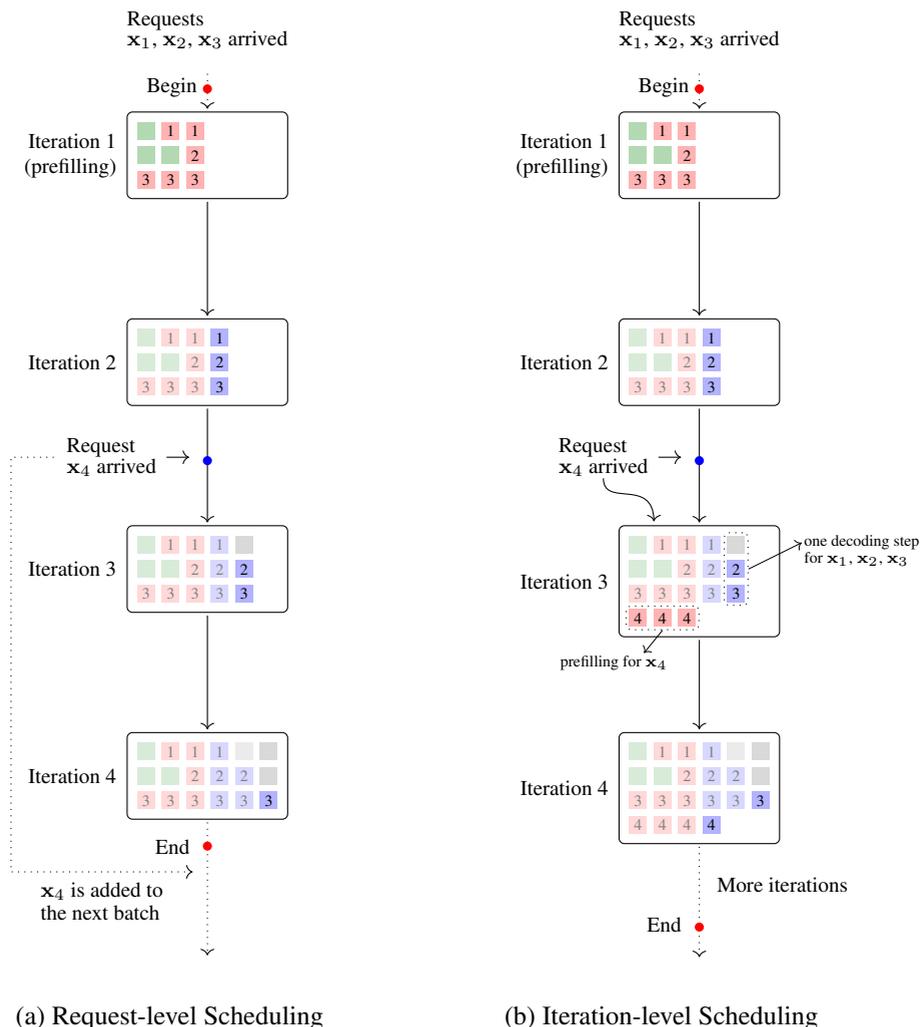


Figure 11.10: Illustrations of request-level scheduling and iteration-based scheduling. In request-level scheduling, once a batch is created and sent to the inference engine, we cannot adjust the batch. In other words, scheduling only occurs after the processing of a batch finishes. In iteration-level scheduling, we can perform scheduling during batch processing. For example, if a new request arrives at some point during inference, we can add it to the batch and continue processing.

then continues processing the updated batch. Note that the inference engine now processes different sequences in different ways:  $x_1$  and  $x_2$  proceed with the decoding process (i.e., predicting the next tokens), while  $x_3$  undergoes the prefilling process. After some time, the generation for  $x_2$  completes. As it happens, two more user requests,  $x_4$  and  $x_5$ , arrive. The scheduler removes the completed sequence  $x_2$  from the batch and, considering the current load of the inference engine, adds  $x_4$  to the batch. However,  $x_5$  must wait until another sequence in the batch finishes before it can be added.

The idea behind continuous batching is to keep the inference engine fully utilized by

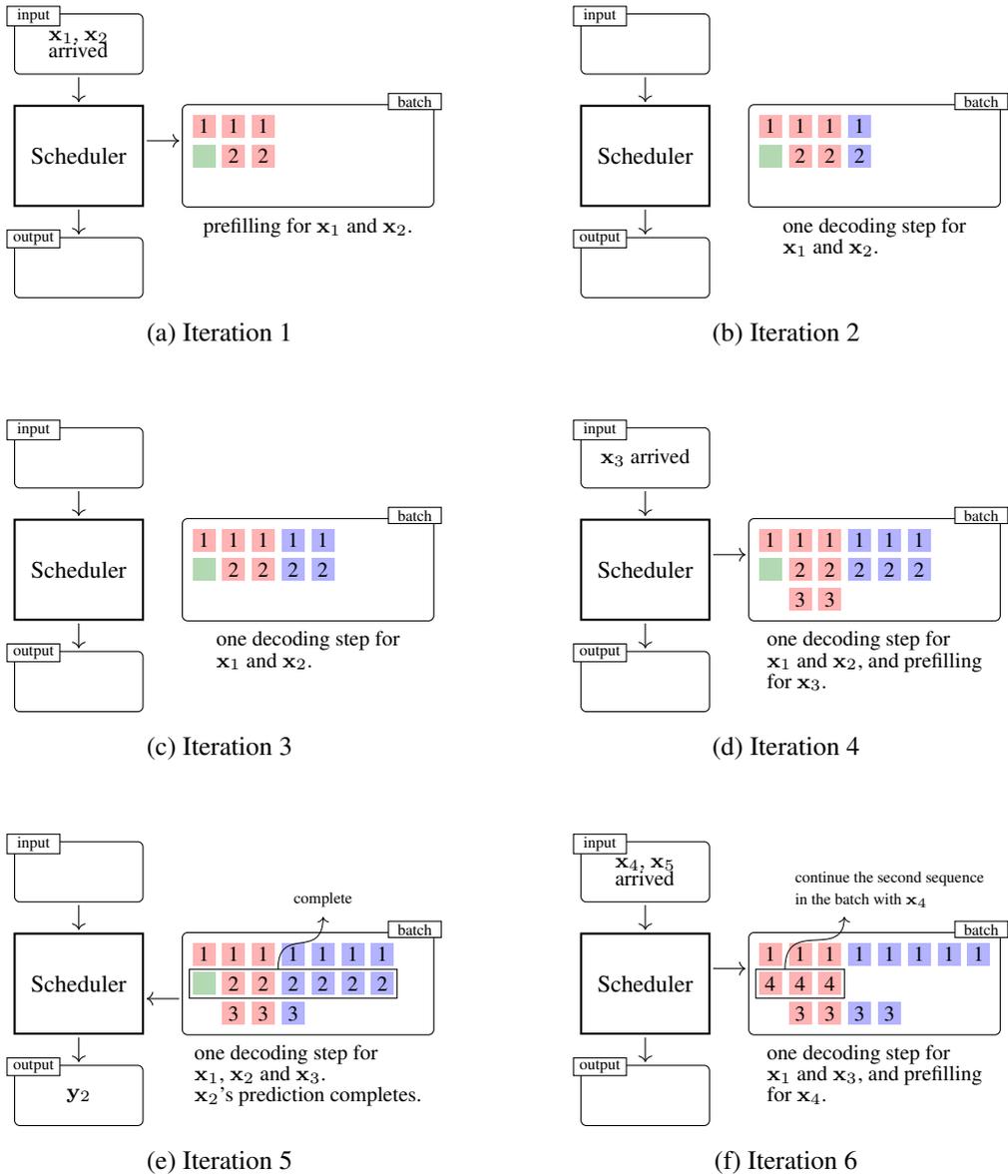


Figure 11.11: Illustration of batch adjustment in continuous batching. Instead of fixing a batch of input sequences and processing them to completion (as in request-level batching), continuous batching dynamically updates the batch during inference. The system continuously accepts and adds new requests (e.g.,  $x_3$  and  $x_4$ ) into the current batch as long as there is available compute capacity.

processing as many sequences as possible, thereby maximizing computational resource usage. A key difference between continuous batching and standard batching (see Figure 11.8) lies in the fact that, in continuous batching, prefilling and decoding can occur simultaneously across different sequences, whereas in standard batching, these two phases are performed sequentially for the entire batch. As discussed in Section 11.1.2, prefilling is considered a

compute-bound process, while decoding is considered a memory-bound process. The intuition behind overlapping prefilling and decoding is to reduce idle times for both computation and data transfer. Consider two mini-batches: one for prefilling and one for decoding. While the prefilling mini-batch keeps the GPUs occupied, the decoding mini-batch can perform memory transfers concurrently.

Another difference between continuous batching and standard batching is that continuous batching is prefilling-prioritized, while standard batching is decoding-prioritized [Agrawal et al., 2024]. In continuous batching, once the inference engine has spare computational resources, the scheduler will add new requests to the batch. In other words, these newly added requests will be processed for prefilling as early as possible. This approach improves system throughput, but at the cost of increased latency, as the newly added requests extend the processing time of earlier ones. In contrast, in standard batching, once the batch is created, we must wait for the last sequence in the batch to complete before processing new requests. This ensures relatively low latency, but results in lower device utilization and system throughput.

It is important to note that the cost of continuous batching is that we need to continuously reorganize the batches, which involves rearranging the data in memory. Each time a new request is added, the scheduler needs to reassess and optimize the current batch structure. This dynamic adjustment can incur additional memory and computational overhead, especially when the batches are frequently adjusted. Therefore, while this method can improve throughput, it may also lead to increased memory fragmentation and, in some cases, introduce additional latency.

### 3. PagedAttention

PagedAttention (or paged KV caching) is a technique used in the vLLM system [Kwon et al., 2023]. Inspired by operating system paging, it optimizes memory usage during LLM inference — particularly for the KV cache — by addressing fragmented memory allocation in dynamic batching scenarios with variable-length sequences. The idea behind PagedAttention is to break down large memory requirements for KV caching into more manageable "pages" or chunks of memory. In this way, we do not need to store the KV cache of the full sequence in a continuous memory. Instead, the KV cache is divided into fixed-size blocks (analogous to memory pages in an operating system), which can be non-contiguously allocated in physical memory. One advantage of PagedAttention is that it enables flexible memory management, supporting dynamic sequence growth without requiring expensive reallocation or copying of large contiguous memory regions. Note that PagedAttention is not specifically designed for batching. But it indeed helps improve memory efficiency in batched inference scenarios, where memory management is more demanding and complicated.

Consider a simple example of memory allocation in Figure 11.12 in which self-attention is performed for a batch consisting of two sequences. For each sequence, we need to attend the current token to the key-value pairs in the KV cache of this sequence, as required by self-attention. In the standard implementation of self-attention, the KV cache is stored in a contiguous block of memory, allowing us to efficiently access this continuous memory. However, in a paged KV caching system, the KV cache is divided into smaller, fixed-size



memory blocks which are not necessarily contiguous. These smaller KV cache blocks can be more effectively allocated to fragmented memory regions, thereby improving memory utilization. Another benefit of distributing chunks of the KV cache across different memory blocks is that it enables parallelization of the caching process. For example, if the input sequence is long and the memory bandwidth is sufficient, it would be beneficial to write and read the key and value vectors of different segments of the sequence in parallel across multiple memory blocks.

In general, storing contiguous data in non-contiguous regions can cause issues, for example, accessing fragmented data requires additional seek time, which reduces I/O efficiency. However, when handling large-scale data (e.g., performing multiplication on extremely large matrices), we typically do not process all the data at once but instead divide it into smaller blocks for block-level computation. From this perspective, it is also reasonable to partition the attention computation. If the paging strategy is well designed, the additional overhead in memory access can be minimal, while the improvement in memory utilization can be significant.

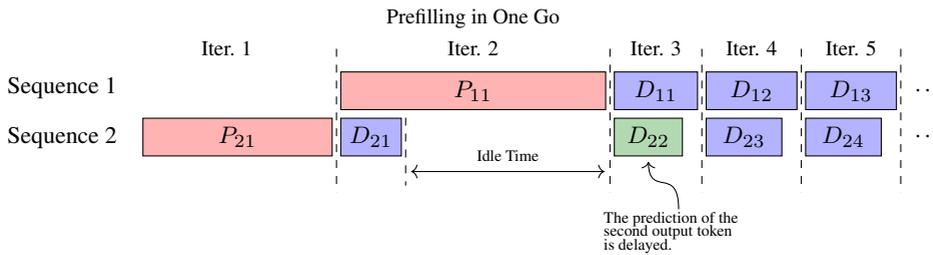
#### 4. Chunked Prefilling

We have seen that, in iteration-level scheduling, prefilling and decoding for different sequences can occur simultaneously. This can be seen as a prefilling-prioritized strategy which can maximize the throughput. However, one such iteration can take a long time if the input sequence is very long and the prefilling process dominates the computation. In this case, decoding for other sequences has to wait until the prefilling completes, leading to increased latency for generating output tokens. Therefore, while prefilling-prioritized strategies are effective for maximizing hardware utilization, they may introduce significant variability in token generation latency, particularly when the system is handling a mix of long and short input sequences.

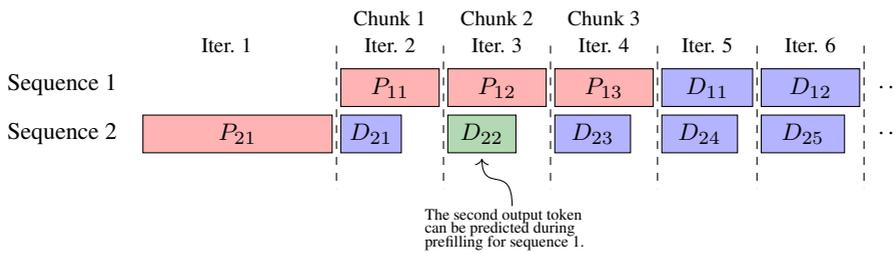
A simple way to reduce decoding latency is to make computations for different sequences in the batch comparable. One such method is to divide sequences into chunks and perform prefilling chunk by chunk. This approach, often referred to as chunked prefilling, processes smaller portions of each sequence at a time, allowing the system to better balance the computational load across sequences [Agrawal et al., 2023]. By choosing an appropriate chunk size, we can ensure that when prefilling and decoding overlap for two sequences, their processing within the same iteration tends to take a similar amount of time. As a result, decoding idle time is reduced and overall throughput is improved.

Figure 11.13 shows an illustration of chunked prefilling in a few iterations. In this example, the batch contains two sequences. The whole prefilling process of the first sequence is divided into three prefilling steps, giving rise to the chunks denoted  $P_{11}$ ,  $P_{12}$  and  $P_{13}$ . Each chunk corresponds to one iteration and can thus overlap with one decoding step. In this way, during the prefilling of the first sequence, we can perform three decoding steps, rather than only a single decoding step, as is the case in standard iteration-level scheduling. As a result, the idle time of the decoding process is reduced, and the output tokens can be generated earlier.

Chunked Prefilling improves decoding efficiency by overlapping prefilling and decoding,



(a) Simple Iteration-level Scheduling



(b) Chunked Prefilling

Figure 11.13: Comparison of simple iteration-based scheduling and chunked prefilling.  $P_{xy}$  denotes the  $y$ -th prefilling step for sequence  $x$ , and  $D_{xy}$  denotes the  $y$ -th decoding step for sequence  $x$ . In simple iteration-based scheduling (or prefilling-prioritized scheduling), since prefilling is treated as a single iteration,  $D_{22}$  has to wait for the completion of the prefilling of sequence 1. In chunked prefilling, the prefilling process can be divided into multiple steps. Thus,  $D_{22}$  can execute during prefilling for sequence 1 (i.e., during  $P_{12}$ ).

but at the cost of additional memory overhead and scheduling complexity. In standard prefilling, we process the whole input sequence once, building the KV cache in one go. By contrast, in chunked prefilling, each chunk needs a separate forward pass to compute its attention outputs and update the KV cache. As a result, we need to maintain the KV cache of early chunks while processing later chunks. This also compromises the parallelism of completing the prefilling for the entire sequence in a single pass. In practice, it is usually possible to balance throughput and latency by choosing an appropriate chunk size.

It is worth noting that the methods discussed in this subsection can broadly be categorized as priority-based scheduling methods. In these methods, we can give priority to certain requests, or to certain prefilling or decoding steps, so that system resources are allocated in a way that better aligns with specific performance goals. As presented above, for example, we may prioritize decoding over prefilling to minimize token generation latency, or prioritize prefilling over decoding to maximize overall throughput in batch-processing scenarios. Practitioners can design custom priority policies for specific needs and operational constraints in real-world applications, such as request deadlines and importance levels defined by users.

### 11.2.3 Parallelization

Parallelization is a widely used approach to scale up LLM inference, especially for large-scale deployments. In Chapter 7, we have discussed several common parallelization strategies to parallelize LLM pre-training, such as model parallelism, tensor parallelism, and pipeline parallelism. We have also discussed efficient architectures that are easy to deploy in distributed computing systems. For example, in MoE models, we assign different experts to different devices<sup>3</sup>. Only the active experts for a given input are executed, which significantly improves computational efficiency while maintaining model quality. Many of these methods can be directly applied to LLM inference with minimal modifications.

However, applying these parallelization techniques to inference poses new challenges compared to pre-training. These issues become especially pronounced in real-time or low-latency inference scenarios, where load imbalance across devices and communication overhead can significantly impact performance. For example, unlike pre-training, where batches can be prepared in advance, inference must handle variable-length sequences in real time. This makes it harder to maintain optimal device utilization and complicates scheduling across heterogeneous computational resources. A related issue is load balancing. When a large number of requests arrive in a short period of time, the system must efficiently distribute workloads across available devices. For example, real-world requests typically exhibit highly variable computational demands due to differences in task types and prompt lengths. Such variability renders simple static load balancing approaches ineffective, and so we need to use finer-grained strategies that can adapt to runtime conditions. The problem becomes even more complicated when we deploy the system on heterogeneous hardware and there are strict latency constraints.

In the development of LLMs, parallelization is closely related to LLM serving. Generally, building a high-quality LLM serving system is not a simple task — it typically requires the combination of multiple techniques, such as architectural design, workload distribution, and LLM-specific hardware/software optimizations. As such, LLM serving constitutes an exceptionally broad subject that often demands substantial engineering expertise. Here, we will not go into the details of LLM serving. For related concepts and techniques, readers may refer to relevant open-source systems (such as vLLM<sup>4</sup>, TensorRT-LLM<sup>5</sup> and TGI<sup>6</sup>) and papers [Pope et al., 2023; Li et al., 2024a].

### 11.2.4 Remarks

We have considered many methods for improving the efficiency of LLMs in this and previous chapters. Although these approaches address different issues, most of them essentially explore trade-offs between various performance factors. One important trade-off is between inference speed and accuracy. For example, techniques like quantization, pruning, and knowledge

---

<sup>3</sup>In LLMs, the experts are typically modular FFNs. So each expert is a part of the FFN component in the Transformer architecture.

<sup>4</sup><https://github.com/vllm-project/vllm>

<sup>5</sup><https://github.com/NVIDIA/TensorRT-LLM>

<sup>6</sup><https://github.com/huggingface/text-generation-inference>

distillation can significantly reduce computational overhead and latency but may introduce minor degradations in model performance. Conversely, preserving full precision or using larger models enhances accuracy but at the cost of slower inference and higher resource demands.

Another important consideration in LLM inference is the memory-compute trade-off. As in computer system design, we need to consider the balance between memory usage and computation required to generate the output. In particular, storing intermediate results such as KV caches during inference can significantly reduce redundant computation, but at the cost of increased memory usage. In KV caching, storing past attention states avoids recomputation of self-attention over previous tokens, thereby reducing compute time per token. However, as the number of tokens grows, so does the memory footprint of the KV cache, especially when processing very long sequences or multiple sequences in parallel. In response, various techniques have been developed to reduce memory consumption by partially recomputing intermediate states. For instance, chunked or windowed attention limits the attention span to a recent subset of tokens, reducing KV cache size at the cost of reduced context or additional compute if past information must be reprocessed.

Note that considering the memory-compute trade-off is a very general principle. It can be extended beyond attention mechanisms and Transformers to other components in system design. An example is the choice of data precision. Using lower-precision formats such as FP16 or INT8 can reduce both memory usage and memory bandwidth requirements, effectively alleviating pressure on the memory subsystem. However, lower precision may lead to numerical instability or slight accuracy degradation, requiring careful calibration or retraining. Thus, this trade-off can also be seen as a memory-compute-accuracy triangle, where improvements in one dimension may come at the expense of another.

Beyond speed, accuracy, and memory, several other dimensions also influence LLM inference efficiency. Some of these dimensions have been discussed in this chapter, while others have not. Here we outline them as follows.

- **Throughput vs. Latency:** In large-scale multi-user LLM serving scenarios, we often aim to maximize system throughput. For example, as discussed in this section, we can batch multiple requests together to increase the number of tokens processed at the same time. However, batching increases waiting time and may lead to higher per-request latency, especially for short or interactive requests. By contrast, optimizing for low latency often requires serving requests individually or in smaller batches, which underutilizes hardware resources and reduces throughput. Achieving a good balance depends on the quality-of-service requirements and user interaction patterns.
- **Generalization vs. Specialization:** General-purpose LLMs are trained to perform a wide range of tasks with a single set of parameters. While flexible, they may be less efficient or accurate for specific tasks. Specialized models can yield better performance and lower inference costs for targeted applications. However, maintaining multiple specialized models increases system complexity and storage requirements. The trade-off between maintaining a single general model versus multiple specialized models is an important system-level design choice.

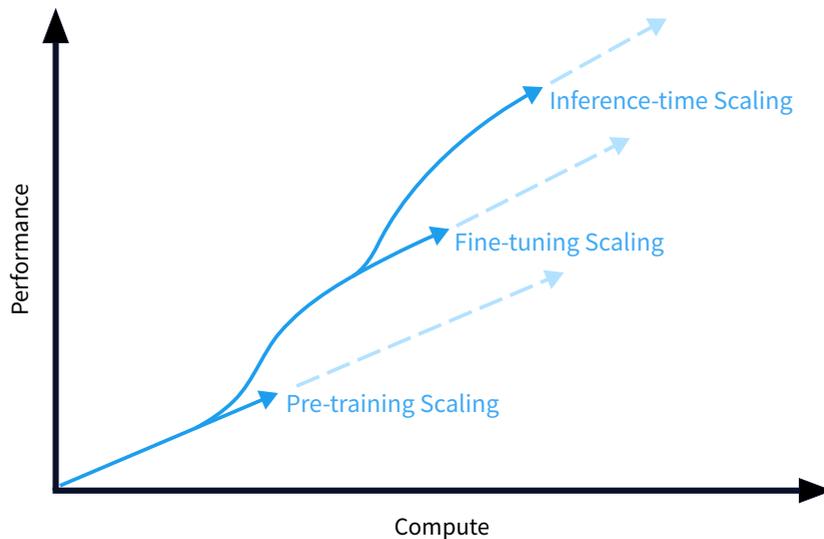


Figure 11.14: Scaling for pre-training, fine-tuning and inference stages [Briski, 2025].

- **Energy Efficiency vs. Performance:** High-performance inference often requires running large models at high throughput on powerful accelerators, which consumes considerable energy. This may be problematic for edge deployments or energy-sensitive environments. Techniques like model compression can improve energy efficiency, but usually with some degradation in output quality or increase in latency. Energy constraints thus introduce another important dimension in optimizing LLM inference.

## 11.3 Inference-time Scaling

Scaling laws can be considered one of the fundamental principles guiding the development of LLMs. In previous chapters, we discussed several times that scaling up training data, model size, and compute can effectively improve the performance of pretraining. In fact, scaling laws also apply to downstream stages such as fine-tuning and inference (see Figure 11.14). Here we consider **inference-time scaling**, which has been widely employed by recent LLMs to solve complex problems, such as complex math problems [Snell et al., 2025]. Unlike pre-training and fine-tuning scaling, which focuses on improving LLMs via parameter updates, inference-time scaling improves these models during inference without further training. This includes a large variety of methods which scale LLMs in different dimensions, such as ensembling multiple model outputs, increasing context length, adopting more aggressive decoding algorithms, and using external tools to extend model capabilities.

While inference-time scaling is wide-ranging, in this section we consider those methods that incorporate more compute into inference (called inference-time compute scaling). Here is

a list of inference-time (test-time) compute scaling methods, organized by category:

- **Context Scaling.** It involves scaling the input or context to improve generation (or potentially scale the output).
- **Search Scaling.** It involves increasing computational effort during decoding.
- **Output Ensembling.** It involves combining multiple model outputs.
- **Generating and Verifying Thinking Paths.** It involves guiding LLMs to generate and verify thinking paths for solving complex reasoning problems.

We will describe these methods in the following subsections.

### 11.3.1 Context Scaling

Context scaling improves LLM performance by extending the input to the model. A straightforward approach is to incorporate more helpful context during inference, allowing the model to condition its predictions on more prior information. One example is few-shot prompting. It augments the context with multiple input-output examples, and so the model can learn task behavior implicitly from these examples without parameter updates. On top of few-shot prompting, we can use chain-of-thought prompting to encourage the model to produce intermediate reasoning steps before final answers. Note that chain-of-thought prompting is one of the most important methods in addressing reasoning problems. By explicitly providing intermediate steps in problem-solving, we can prompt the model to break down complex tasks into simpler sub-tasks, which is found to be very beneficial for generating accurate and interpretable outputs.

Beyond extending the prompt with examples or reasoning steps, another approach to context scaling involves dynamically incorporating external knowledge. This is often achieved through RAG. RAG systems first retrieve relevant document snippets from a large collection of documents or a database based on the current input. These retrieved pieces of information are then added to the context provided to the LLM. This essentially expands the context to include timely or specialized external knowledge. By doing so, the model grounds its responses in specific knowledge found in the external source. The LLM thus can generate responses that are not only relevant to the input but also factually accurate and up-to-date.

However, as the context grows, these methods often suffer from the constraints of finite context window length. While model architectures and techniques (like efficient attention models) are continually evolving to support longer contexts, processing extremely long inputs still poses challenges. Increased computational cost is one factor. More critically, when the context window becomes very large, the model might struggle to attend effectively to the most relevant information (e.g., the “lost in the middle” phenomenon). Therefore, effective context scaling is not just about adding more information, but also about strategically selecting, structuring, and presenting the most pertinent information within the model’s processing capabilities.

Here we omit the detailed discussion of these methods, as they have already been covered in previous chapters. See Chapters 8 and 9 for more details, including prompting, RAG, and

long-sequence modeling methods.

### 11.3.2 Search Scaling

In LLMs, decoding is a search process that aims to efficiently find the best output sequence given the input sequence. Search scaling (or decoding scaling) typically involves two aspects: scaling the output length and scaling the search space.

Scaling the output length refers to increasing the number of tokens generated during inference. This is especially important in tasks that require long-form generation, such as story writing. More recently, generating outputs with long thinking paths has shown strong performance in math problem solving and code generation. For example, encouraging the model to generate long thinking paths before producing the final answers has been found to be very beneficial in performing complex reasoning. This idea has been widely used in developing recent LLMs for reasoning, such as [OpenAI \[2024\]](#)'s o1 and [Deepseek \[2025\]](#)'s R1. We will discuss more about output length scaling in Section 11.3.4.

Scaling the search space, on the other hand, refers to expanding the set of candidate output sequences considered during search, so that higher-quality outputs can be found. As discussed in Section 11.1.3, a simple example is that in beam search we increase the beam width to allow more candidate sequences to be explored in parallel at each decoding step. This increases the chance of discovering better outputs, especially in tasks where the optimal solution is not immediately apparent from local decisions.

In addition to decoding algorithm adjustments, it is also possible to explore compact structures to encode a large number of outputs. For example, we can construct and navigate a tree or graph of reasoning steps [[Yao et al., 2024](#)]. In this paradigm, each node represents a partial solution or intermediate step, and edges represent transitions between reasoning states. Such structured search enables the model to consider multiple paths simultaneously. Another related direction is Monte Carlo tree search-inspired decoding, where the model stochastically explores and scores different paths based on learned heuristics or external reward models.

Search scaling is a very general idea, and it is often implicitly involved in the design of search procedures that exploit search structure, heuristics, and model uncertainty. Many of the above methods have been discussed previously, though they were not originally developed with scaling as their primary goal. However, search scaling inherently comes with computational costs. Increasing beam width, for instance, directly translates to higher memory usage and longer inference times. In practice, there is often a point of diminishing returns, where further expansion of the search space yields marginal improvements in output quality at a significant computational expense. Therefore, an effective strategy often involves finding an optimal balance between scaling and computational feasibility.

### 11.3.3 Output Ensembling

If we have multiple model outputs, it is often beneficial to combine them to mitigate the impact of individual model errors and synthesize a superior final output. Each model might capture different aspects of the underlying data distribution or possess unique strengths and weaknesses. By ensembling, we can average out the noise or random errors present in individual predictions,

leading to a more stable and reliable outcome. In LLM ensembling, one of the simplest approaches is to average the probability distributions over the next token from each model, and select the best token using this averaged distribution. Or, if we regard the problem as a discrete decision-making task, majority voting can be employed. More sophisticated methods might involve re-ranking candidate outputs generated by different models based on a separate scoring function or even using a meta-learner to intelligently combine the predictions.

The “scaling” from output ensembling comes at the cost of running multiple models or sampling multiple outputs. This not only increases the latency of inference but also leads to the additional complexity of managing multiple models. But the quality of outputs does not continue to improve indefinitely as more models are added. In some cases, the benefits of output ensembling may diminish as the number of component models in the ensemble exceeds a certain threshold. Instead, the benefits of ensembling are generally greater when the individual models are diverse (i.e., they make different errors), even if there are a relatively small number of component models. Therefore, it is common practice to use a set of diverse LLMs which differ in their training data, model architectures, or fine-tuning objectives.

In LLMs, “scaling” often implies making things “bigger” for quality with more resources. However, in addition to scaling up the quality, scaling can mean more. It can also signify scaling up the robustness (making the system less prone to errors and more reliable) and exploration (covering a wider range of potential solutions). In output ensembling, these dimensions are naturally integrated. For instance, the very act of averaging or voting across different model outputs is a direct strategy to scale up robustness against individual model failures. Furthermore, by intentionally including varied models, ensembling increases the chances of discovering novel or superior solutions. In this sense, scaling is not limited to making models larger or running them longer — it also means strategies for making inference more robust, exploratory, and adaptive.

### 11.3.4 Generating and Verifying Thinking Paths

So far, we have viewed inference-time scaling as a general class of methods for scaling various aspects of inference, such as sequence length, model size, and/or search strategies. In fact, one successful application is the use of inference-time scaling to enhance the reasoning capabilities of LLMs. As we have seen, the reasoning performance of LLMs can be improved by using chain-of-thought methods. We can therefore make use of the chain-of-thought prompts to generate intermediate reasoning steps and reach a correct answer. However, reasoning problems are often so complicated that we cannot obtain high-quality solutions by providing simple chain-of-thought prompts. For example, when solving a math problem, we typically need to reason over a sequence of steps. At each step, we need to work out some intermediate result, verify it, and then determine what to do next. The reasoning path is not a fixed pattern but a dynamically generated thinking process that often involves trial-and-error, backtracking, and self-correction. This requires more sophisticated prompting strategies or search algorithms to navigate such complex reasoning. In this subsection, we focus on inference-scaling methods that go beyond simple chain-of-thought to address complex reasoning problems more effectively.

At a high level, methods for scaling the reasoning of LLMs can be categorized into two

classes:

- **Training-free Methods.** These methods aim to improve reasoning capabilities without requiring any modification or retraining of the pre-trained parameters. Instead, they focus on techniques applied during inference, such as sophisticated prompting strategies (e.g., chain-of-thought) and algorithmic control over the reasoning process (e.g., search).
- **Training-based Methods.** These methods involve further training or fine-tuning the model parameters to explicitly improve reasoning abilities, such as supervised fine-tuning on datasets with reasoning examples (e.g., math problems with step-by-step solutions).

In the following, we first discuss training-free methods, and then training-based methods.

### 1. Solution-level Search with Verifiers

Given an input sequence (e.g., a math problem), there are many possible output sequences (e.g., solutions to the problem). If we have a model to evaluate or verify each solution, we can select the best one. This is the fundamental principle behind methods like best-of- $N$  sampling, where multiple outputs are generated, and the optimal result is picked based on some selection mechanism. Such a selection process can be viewed as a search problem, which involves two components:

- **Search Algorithm.** This defines the strategy used to explore the space of possible output sequences (solutions) and generate a set of candidates. It can range from simple independent sampling to more sophisticated search techniques as discussed in Section 11.1.3.
- **Verifier.** This is a model or function responsible for evaluating the quality, correctness, or utility of each candidate solution generated by the search algorithm. It provides a score, a probability, or a judgment that allows the system to select the best among the candidates. The verifier can be another LLM, or even a set of predefined rules or heuristics.

Given an input problem  $x$ , we define that an output solution  $y$  can be represented as a sequence of reasoning steps:

$$\mathbf{y} = (a_1, a_2, \dots, a_{n_r}) \quad (11.37)$$

where  $a_i$  is the  $i$ -th reasoning step, and  $a_{n_r}$  is the last step which should contain the answer to the problem. See Figure 11.15 for an example of a multi-step reasoning path.

The search algorithm can efficiently generate a set of candidate solutions

$$\mathcal{D}_c = \{\mathbf{y}_1, \dots, \mathbf{y}_K\} \quad (11.38)$$

Then, we can use a verifier, which evaluates each solution by the function  $V(\mathbf{y})$ , to score

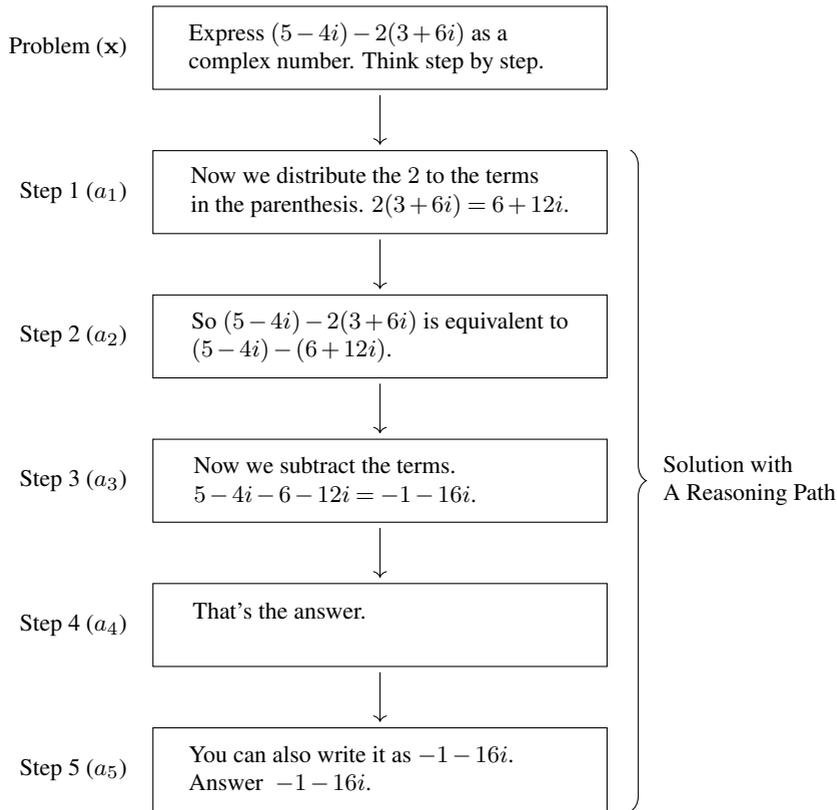


Figure 11.15: Illustration of multi-step reasoning. This example is from the PRM800K dataset [Lightman et al., 2024]. Given a math problem, the LLM is prompted to generate a thinking path (or reasoning path) consisting of several reasoning steps. Each step addresses a sub-problem based on the results of the previous steps. The answer to the original problem is contained in the last step.

the candidates in  $\mathcal{D}_c$ . The final output is the best candidate selected by the verifier

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \mathcal{D}_c} V(\mathbf{y}) \quad (11.39)$$

Although verifying the entire reasoning path is possible, a simpler alternative is to verify only the final reasoning step. In this way the verifier function  $V(\mathbf{y})$  is simplified to depend solely on the final answer contained within  $a_{n_r}$ . This can be achieved in various ways, depending on the nature of the problem and the expected answer format.

- For some math and coding problems, we can use off-the-shelf tools as verifiers. Examples include proof checkers for mathematical theorems, interpreters or compilers for code execution, and unit test systems for verifying program correctness against predefined test cases.
- If there is labeled data for evaluating the answer, such as human preference data, we can train a reward model on such data. The learned reward model is then used as the verifier

which assigns a scalar score to each candidate answer.

- If there are no existing systems or suitable reward models, we can use another LLM to act as the verifier. This LLM is prompted to assess the quality of the candidate answer. It could potentially be a more capable model, or the same LLM used with a specific “evaluator” prompt.
- Alternatively, simpler heuristic-based verifiers can be designed. A commonly used approach is to employ majority voting, where the most frequently occurring answer among a set of candidates is selected.

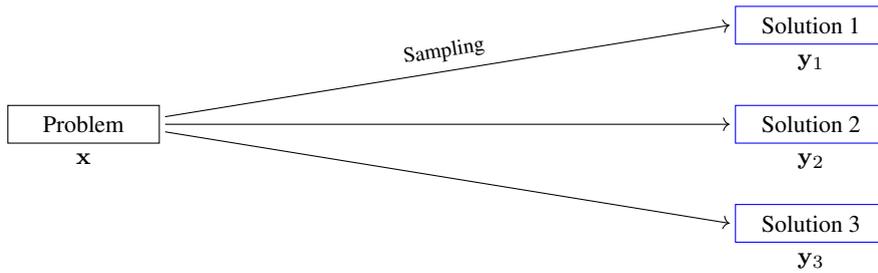
Based on these verifiers, we can search to obtain a set of candidate solutions for selection. One simple strategy, which is often referred to as **parallel scaling** [Brown et al., 2024; Snell et al., 2024], involves generating  $K$  candidate solutions by running the base LLM  $K$  times independently. In this process, we can adjust the temperature in sampling to control the diversity in the outputs. The verifier then assesses each of these  $K$  complete solutions, and the one with the highest score is selected as the final output. This is conceptually very similar to best-of- $N$  sampling, which in previous chapters we primarily described as a method of selecting the best one from a set of sampled outputs using a reward model.

Another approach is **sequential scaling**, which builds a sequence of solutions incrementally [Gou et al., 2024; Zhang et al., 2024]. It starts with an initial solution generated by the LLM with prompting. Then, we use a verifier (often the same LLM) to evaluate the solution. This can be seen as a critique stage. The output of this stage is some form of feedback, such as textual critiques pinpointing errors or suggesting improvements, numerical scores reflecting solution quality, or even a revised plan or intermediate step to guide the next generation. This feedback, along with the original problem and the current solution, is then used to prompt the LLM to generate a potentially improved solution. This can be seen as a refine stage. This critique-refine cycle can be repeated, forming an iterative loop:

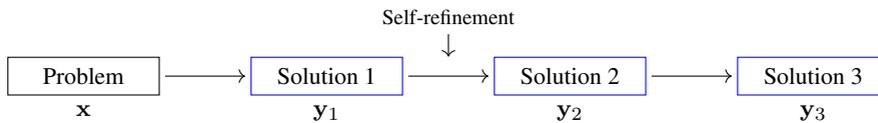
$$\mathbf{y}_{k+1} = \text{Refine}(\mathbf{x}, \mathbf{y}_k, \text{Feedback}(\mathbf{y}_k)) \quad (11.40)$$

where  $\text{Feedback}(\mathbf{y}_k)$  represents the feedback from the verifier. The  $\text{Refine}(\cdot)$  function generates the improved solution  $\mathbf{y}_{k+1}$  by prompting the LLM with the original problem  $\mathbf{x}$ , the previous solution  $\mathbf{y}_k$ , and this feedback. The process can be iterated for  $K$  times, or until the solution quality, as assessed by the verifier, converges to a satisfactory level. This iterative framework, where a solution is progressively improved through cycles of generation, evaluation (critique), and revision, is precisely what constitutes self-refinement [Shinn et al., 2023; Madaan et al., 2024]. In such scenarios, the role of the verifier is not just to pick the best complete solution from a static set, but to actively guide the generation process itself.

See Figure 11.16 for illustrations of parallel scaling and sequential scaling. Note that there are other ways to perform search and obtain different sets of candidate solutions. One alternative method is to organize search as a tree structure. This approach, often referred to as tree search, provides a more structured way to explore the space of possible reasoning paths. In solution-level search, each node of the tree represents a complete solution. During search, we need to expand a node to a set of child nodes, representing new solutions that can be considered



(a) Parallel Scaling



(b) Sequential Scaling

Figure 11.16: Illustrations of parallel scaling and sequential scaling. In parallel scaling, we obtain multiple solutions by running the LLM several times independently. In sequential scaling, the LLM generates an initial solution. Then, we use the LLM to refine it iteratively, with each refinement yielding a new, possibly better solution.

in verification. The expansion process typically involves taking an existing solution (the parent node) and using the LLM to generate variations or alternative solutions.

## 2. Step-level Search with Verifiers

While the methods discussed above primarily focus on generating complete solutions before final selection, the search process can also be integrated more deeply into the step-by-step generation of the reasoning path itself. This leads to approaches that perform step-level search with verifiers, where guidance or pruning occurs at intermediate reasoning steps  $\{a_1, \dots, a_{n_k}\}$  rather than only after a full solution  $y$  is formed.

Such fine-grained control is particularly beneficial for complex reasoning problems where a single incorrect intermediate step can render the entire subsequent reasoning chain invalid. By evaluating or guiding the generation at each intermediate step, the LLM can explore the reasoning space more effectively, potentially pruning unpromising paths early or allocating more resources to explore more plausible ones.

Step-level search with verifiers can also be modeled as a tree search problem. In this paradigm, each node (or state) corresponds to a partial reasoning path,  $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$ , representing the sequence of  $i$  reasoning steps taken so far (i.e., a path from the root node to the current node). The objective of the search process is to explore the underlying state space, starting from an initial empty path, to find a complete path that constitutes a correct solution. Note that we use  $\mathbf{a}_{\leq i}$  here to represent a partial reasoning path instead of  $\mathbf{y}_{\leq i}$ . While this makes notation a bit inconsistent with that used for representing complete solutions ( $y$ ) or full

paths in solution-level search, it serves to highlight the focus on individual actions or steps.

The core components of step-level search with verifiers are:

- **Node Representation.** A node is a partial reasoning path  $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$ . The root node is an empty path, and terminal nodes are complete reasoning paths.
- **Node Expansion.** Given a current partial path  $\mathbf{a}_{\leq i}$ , the LLM is used to generate one or more candidate next reasoning steps  $\{a_{i+1}^{(1)}, \dots, a_{i+1}^{(M)}\}$ . Each candidate step, when appended to  $\mathbf{a}_{\leq i}$ , forms a new potential partial path  $\mathbf{a}_{\leq i+1} = (a_1, \dots, a_i, a_{i+1}^{(j)})$ .
- **Verification.** The verifier  $V(\cdot)$  evaluates the quality of a newly generated step in the context of the current partial path  $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$  and the original problem  $\mathbf{x}$ . As with solution-level verification, step-level verifiers might output a numerical score, a categorical label, and textual feedback.
- **Search.** This governs how the search space is explored. Based on the evaluations from the verifier, the search strategy decides which partial paths to extend further, which to prune, and the order of exploration.

This step-by-step verification allows for dynamic adjustments to the reasoning process. If a step  $a_{i+1}$  is deemed incorrect or unpromising by  $V(\cdot)$ , the search algorithm can backtrack and explore alternative steps from  $\mathbf{a}_{\leq i}$ , or even from an earlier node  $\mathbf{a}_{\leq i'}$  (where  $i' < i$ ). Conversely, if a step is highly rated, resources can be focused on extending that path. See Figure 11.17 for an illustration of step-level search with verifiers.

Clearly, this search framework is very similar to that used in decoding methods for LLMs, as discussed in Section 11.1.3. For example, beam search maintains a set of  $K$  most promising partial sequences at each generation step. This is a form of step-level search where the “verifier” is implicitly the LLM’s own probability model, and the “search” is the pruning mechanism to maintain the beam size.

However, step-level search with explicit verifiers, as described here, presents differences from standard decoding. One of them is that the verifier can be a much more sophisticated component than just the raw output probabilities of the generative LLM. The design of step-level verifiers basically follows that of solution-level verification. A step-level verifier might be a language model that assesses the quality of an individual reasoning step within the context of the preceding path. This LLM can even be fine-tuned to enhance its verification capability. Alternatively, for domains with well-defined rules, it could be a symbolic engine or a set of programmatic checks. Furthermore, verifiers can be designed to predict the future utility or likelihood of success given the current partial path, drawing inspiration from value functions in reinforcement learning. Human expertise can also be incorporated to provide judgments on critical steps, especially in high-stakes scenarios.

One example of such a step-level verifier, particularly when using human feedback to assess intermediate progress, is the **process reward model (PRM)**. A PRM is typically a separate language model trained to output a scalar reward for each reasoning step  $a_{i'}$  within a partial path  $\mathbf{a}_{\leq i}$ . It provides a more direct and fine-grained supervisory signal compared to **outcome reward models (ORMs)** which only evaluate the final solution. However, the development

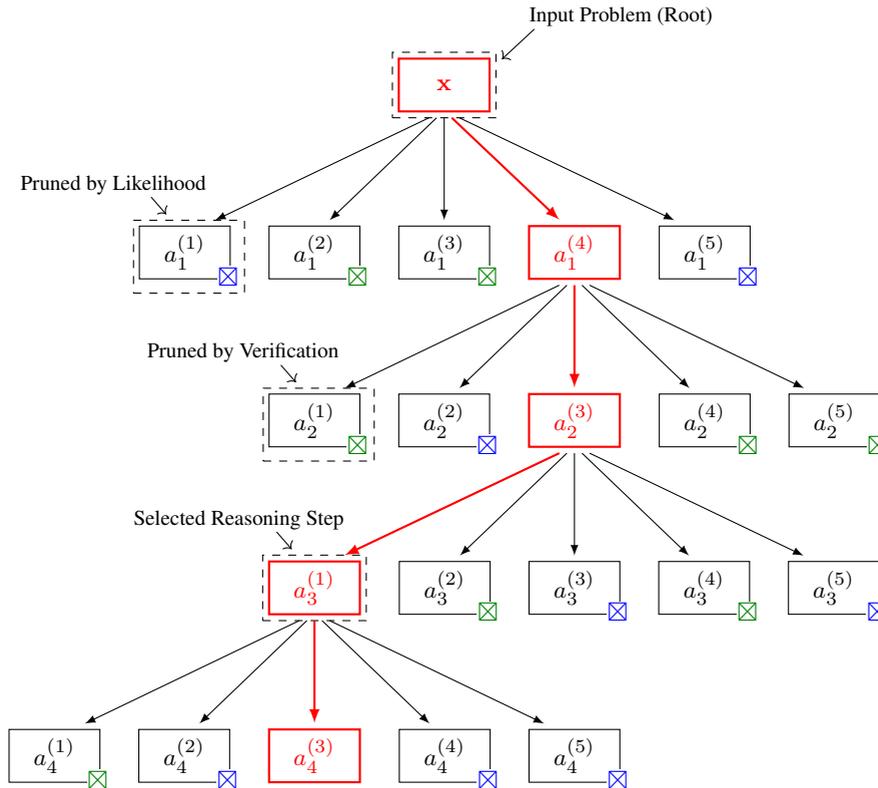


Figure 11.17: Illustration of step-level search with verifiers.  $a_i^{(j)}$  = the  $j$ -th candidate for the  $i$ -th reasoning step,  $\boxtimes$  = candidate pruned by the LLM’s output probability, and  $\boxplus$  = candidate pruned by the verifier. Given the input problem as the root node, we expand the tree by generating multiple reasoning steps at each expansion. Each candidate can be pruned by either likelihood (as in standard decoding) or step-level verification. The unpruned candidates are then expanded to generate further reasoning steps. The process is iterated until a complete reasoning chain leading to a final answer is generated, or until a predefined search limit is reached.

of PRMs relies on step-level human annotations, such as preferences on different next steps. Collecting supervision for each intermediate step is considerably more labor-intensive and requires greater cognitive effort from human annotators than simply labeling final outcomes.

One alternative approach to developing training data for step-level verification is to use LLMs to generate such annotations automatically. For example, we can take a strong LLM, referred to as a teacher model, and prompt it to first generate a complete reasoning path for a given problem. Then, at each intermediate step within this path, we can prompt the same teacher LLM (or another capable LLM) to generate several alternative candidate next steps in addition to the one it originally chose. The teacher LLM can then be prompted again to evaluate these alternatives. These evaluation results (e.g., correct vs. incorrect) can then serve as data annotations. Alternatively, the generalization capabilities of PRMs can be leveraged. We can train a PRM on tasks where step-level verification is easier and then generalize this PRM to other tasks with little or no additional training.

Note that step-level verification also comes with its own problems. Frequent verification, especially if using an LLM as the verifier, can substantially increase computational costs and latency. The design of effective step-level verifiers is non-trivial itself. An inaccurate verifier might prematurely discard good reasoning paths or fail to identify flawed ones, thereby misleading the search. This makes the development of such systems more complex and difficult.

### 3. Encouraging Long Thinking

So far in this subsection, most of the methods are implicitly based on a simple idea: generating longer reasoning paths can help. In addition to CoT and search with verifications, we can consider alternative methods to achieve this. For example, we can prompt the LLM by explicitly asking for extended deliberation. Beyond direct prompting, we can also make modifications to the decoding process itself, such as adjusting token limits or applying penalties for short outputs. Another approach is to employ multi-stage generation schemes where the model incrementally builds upon its reasoning.

### 4. Training-based Scaling

As well as considering inference-time scaling methods without training, we also wish to consider methods that can improve intrinsic reasoning capabilities of LLMs by modifying their parameters through further training. While such training-based scaling methods typically require additional training cost and computational resources, they instill stronger reasoning skills directly into the model parameters, which in turn can lead to more effective and efficient reasoning performance. We can even combine them with training-free methods for better inference-time scaling results.

Although our discussion here is restricted to reasoning problems, methods for training-based scaling are common. Most of them have been discussed in Chapter 10. Here, we will briefly describe how these methods can be applied to improving inference-time scaling for reasoning problems.

- **Fine-tuning on Reasoning Data.** One of the most direct ways to enhance reasoning is by fine-tuning pre-trained LLMs on datasets specifically curated for reasoning tasks. These datasets can range from simple input-output pairs to more structured formats that include step-by-step reasoning processes. Typical examples include datasets of math word problems, logical deduction exercises, or code generation with explanations. By training on such data, the model learns from common reasoning patterns, and thus can generate detailed and coherent reasoning paths at test time.
- **Reinforcement Learning for Reasoning.** If we regard a verifier as a reward model, we can see that the methods discussed in the previous subsection are a direct application of the reward model to reasoning problems, though they are training-free. Of course, we can apply this reward model to LLM fine-tuning. This follows a standard paradigm of reinforcement learning. Given a reward model, the LLM, acting as a policy, is fine-tuned using reinforcement learning algorithms. The LLM generates reasoning steps

or full solutions, receives feedback (rewards) from the reward model, and updates its parameters to produce outputs that maximize these rewards. This process aligns the LLM output with notions of high-quality reasoning, thereby encouraging the LLM to generate more reliable reasoning paths. Another key issue is the training of the reward model. Generally, this reward model could be an outcome reward model that evaluates the correctness or quality of the final answer, or a process reward model that assesses the quality of each intermediate reasoning step, as discussed in the context of step-level verifiers. In some cases, we can even develop a reward model based on simple rules, such as giving bonuses to longer outputs.

- **Knowledge Distillation for Reasoning.** In this approach, a smaller, more efficient student LLM is trained to mimic the reasoning outputs or internal representations of a larger, more capable teacher LLM. The teacher model might generate detailed reasoning steps for a variety of problems. The student model then learns to reproduce these high-quality reasoning demonstrations. This strategy makes stronger reasoning capabilities more accessible by deploying them in smaller models that are less computationally expensive at inference time.
- **Iterative Refinement.** Training-based scaling can also involve iterative refinement. For example, an LLM can generate solutions to a set of problems. These solutions and their reasoning paths are then verified, either by humans or automatic verifiers. The correct reasoning paths are subsequently added to the training data, and the LLM is further fine-tuned on this augmented dataset. This creates a cycle where the LLM progressively improves its reasoning capabilities through repeated generation, critique, and learning.

The primary advantage of these training-based scaling methods is that they endow the LLM with stronger inherent reasoning skills. This directly contributes to improved inference-time scaling in several ways: it can lead to more efficient inference, as the LLM might require less extensive search or fewer generation samples to arrive at a correct solution. Moreover, the base quality of generated steps or solutions is higher. Therefore, a well-trained LLM might generalize its learned reasoning abilities to novel problems more effectively than an LLM relying solely on in-context learning or training-free inference schemes.

On the other hand, training-based approaches also present challenges, compared to the training-free counterparts. The creation of high-quality, large-scale training datasets for reasoning can be expensive and labor-intensive. The fine-tuning process itself, particularly for the largest LLMs or when using RL, can be computationally intensive and require substantial engineering effort. There is also the risk of the model overfitting to the specific types of problems or reasoning styles present in the training data, potentially limiting its performance on out-of-distribution tasks.

## 11.4 Summary

In this chapter, we have discussed the inference issue for LLMs. We have presented the prefilling-decoding framework and related decoding algorithms for LLM inference. Then, we

have described several techniques for efficient inference. We have also discussed inference-time scaling, which has been considered one of the most important methods for improving LLM reasoning.

Inference over sequential data has long been a concern in AI [Wozengraft and Reiffen, 1961; Viterbi, 1967; Forney, 1972]. In the context of NLP, this line of work dates back to the very early days of speech recognition and statistical machine translation [Koehn, 2010], where researchers faced the challenge of efficiently searching vast hypothesis spaces to find the most probable output sequence. Techniques like beam search and various pruning strategies were developed then to make this computationally tractable. At that time, models were relatively weak, and much of the research focused on developing powerful search algorithms to reduce search errors. These foundational ideas continue to influence modern approaches.

As we enter the era dominated by deep learning methods, models based on deep neural networks have become extremely powerful. Even with very simple search algorithms, these models can achieve excellent results. In this context, inference no longer seems as “important” as it once was, and research attention has gradually shifted toward model architectures, training methods, and scaling up models.

However, history tends to repeat itself. With the rise of LLMs, inference has once again attracted significant attention. This renewed focus is primarily manifested in two aspects:

- The inference cost for LLMs is very high. For example, efficiently deploying LLMs in high-concurrency, low-latency scenarios remains a challenging problem, making inference efficiency critically important. In this context, efficient architecture designs, optimized search algorithms, and various inference optimization strategies hold substantial practical significance.
- Input and output sequence lengths have significantly increased in complex tasks. Especially in tasks like mathematical reasoning, the growth of sequence lengths further highlights the importance of inference efficiency. Moreover, scaling the inference process has recently proven to be an effective way to improve the reasoning capabilities of models. Therefore, achieving efficient inference scaling is emerging as a particularly promising research direction.

Inference is now a wide-ranging topic that encompasses many techniques. It involves not only the development of model architectures and decoding algorithms, but is increasingly shaped by the intricate engineering and sophisticated systems-level optimizations required to deploy LLMs effectively and efficiently. Many of these techniques are beyond the scope of NLP or a specific AI area. Instead, the frontier of LLM inference optimization now extends deeply into domains traditionally considered core computer science and engineering. This systemic perspective has brought many new ideas to the study of inference problems. Unfortunately, this chapter cannot cover all relevant techniques — indeed, that would be an almost impossible task in itself. Ultimately, the best way to better understand and master these techniques may still lie in hands-on practice.



## Bibliography

- [Ackley et al., 1985] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.
- [Acs and Kornai, 2016] Judit Acs and András Kornai. Evaluating embeddings on dictionary-based similarity. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 78–82, 2016.
- [Adi et al., 2016] Yossi Adi, Einat Kermany, Yonatan Belinkov, Ofer Lavi, and Yoav Goldberg. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. In *Proceedings of International Conference on Learning Representations*, 2016.
- [Agirre et al., 2009] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalová, Marius Pasca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 19–27, 2009.
- [Agrawal et al., 2023] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.
- [Agrawal et al., 2024] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.
- [Ainslie et al., 2020] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. Etc: Encoding long and structured inputs in transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 268–284, 2020.
- [Ainslie et al., 2023] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.
- [Akhbardeh et al., 2021] Farhad Akhbardeh, Arkady Arkhangorodsky, Magdalena Biesialska, Ondřej Bojar, Rajen Chatterjee, Vishrav Chaudhary, Marta R. Costa-jussa, Cristina España-Bonet, Angela Fan, Christian Federmann, Markus Freitag, Yvette Graham, Roman Grundkiewicz, Barry Haddow, Leonie Harter, Kenneth Heafield, Christopher Homan, Matthias Huck, Kwabena Amponsah-Kaakyire, Jungo Kasai, Daniel Khashabi, Kevin Knight, Tom Kocmi, Philipp Koehn, Nicholas Lourie, Christof Monz, Makoto Morishita, Masaaki Nagata, Ajay Nagesh, Toshiaki Nakazawa, Matteo Negri, Santanu Pal, Allahsera Auguste Tapo, Marco Turchi, Valentin Vydrin, and Marcos Zampieri. Findings of

- the 2021 conference on machine translation (WMT21). In *Proceedings of the Sixth Conference on Machine Translation*, pages 1–88, 2021.
- [Akyürek et al., 2023] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Alabdulmohsin et al., 2022] Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting neural scaling laws in language and vision. *Advances in Neural Information Processing Systems*, 35:22300–22312, 2022.
- [Alayrac et al., 2022] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.
- [Allal et al., 2024] Loubna Ben Allal, Anton Lozhkov, and Daniel van Strien. cosmopedia: how to create large-scale synthetic data for pre-training. <https://huggingface.co/blog/cosmopedia>, 2024.
- [Allauzen et al., 2014] Cyril Allauzen, Bill Byrne, Adrià de Gispert, Gonzalo Iglesias, and Michael Riley. Pushdown automata in statistical machine translation. *Computational Linguistics*, 40(3): 687–723, 2014.
- [Allen and Hospedales, 2019] Carl Allen and Timothy Hospedales. Analogies explained: Towards understanding word embeddings. In *International Conference on Machine Learning*, pages 223–231. PMLR, 2019.
- [Almazrouei et al., 2023] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.
- [Alzantot et al., 2018] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2890–2896, 2018.
- [Ammar et al., 2020] Waleed Ammar, George Mulcaire, Yulia Tsvetkov, Guillaume Lample, Chris Dyer, and Noah A Smith. Massively multilingual word embeddings. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.
- [Anderson et al., 2017] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 936–945, 2017.
- [Andreas et al., 2016] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.
- [Antol et al., 2015] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE*

- international conference on computer vision*, pages 2425–2433, 2015.
- [Arjovsky et al., 2016] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International conference on machine learning*, pages 1120–1128, 2016.
- [Aronoff and Fudeman, 2011] Mark Aronoff and Kirsten Fudeman. *What is morphology?*, volume 8. John Wiley & Sons, 2011.
- [Arora et al., 2017] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. In *International conference on learning representations*, 2017.
- [Artetxe et al., 2017] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 451–462, 2017.
- [Aschenbrenner, 2024] Leopold Aschenbrenner. Situational awareness: The decade ahead, 2024. URL <https://situational-awareness.ai/>.
- [Ash and Doléans-Dade, 1999] Robert B. Ash and Catherine A. Doléans-Dade. *Probability & Measure Theory*. Academic Press, 1999.
- [Askell et al., 2021] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.
- [Åström and Wittenmark, 2013] Karl J Åström and Björn Wittenmark. *Computer-controlled systems: theory and design*. Courier Corporation, 2013.
- [Atkinson and Shiffrin, 1968] Richard C Atkinson and Richard M Shiffrin. Human memory: A proposed system and its control processes. In *Psychology of learning and motivation*, volume 2, pages 89–195. Elsevier, 1968.
- [Auguste et al., 2017] Jeremy Auguste, Arnaud Rey, and Benoit Favre. Evaluation of word embeddings against cognitive processes: primed reaction times in lexical decision and naming tasks. In *Proceedings of the 2nd workshop on evaluating vector space representations for NLP*, pages 21–26, 2017.
- [Ba et al., 2016] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- [Bach et al., 2022] Stephen H. Bach, Victor Sanh, Zheng Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Févry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged Saeed AlShaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir R. Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. Promptsources: An integrated development environment and repository for natural language prompts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 93–104, 2022.
- [Bachlechner et al., 2021] Thomas Bachlechner, Bodhisattwa Prasad Majumder, Henry Mao, Gary Cottrell, and Julian McAuley. Rezero is all you need: Fast convergence at large depth. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 1352–1361. PMLR, 2021.
- [Baevski et al., 2020] Alexei Baevski, Steffen Schneider, and Michael Auli. vq-wav2vec: Self-

- supervised learning of discrete speech representations. In *Proceedings of ICLR 2020*, 2020.
- [Bahdanau et al., 2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [Bahdanau et al., 2016] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4945–4949. IEEE, 2016.
- [Bahl and Mercer, 1976] L. R. Bahl and R. L. Mercer. Part of speech assignment by a statistical decision algorithm. In *Proceedings of IEEE International Symposium on Information Theory*, pages 88–89, 1976.
- [Bai et al., 2021] Jiangang Bai, Yujing Wang, Yiren Chen, Yaming Yang, Jing Bai, Jing Yu, and Yunhai Tong. Syntax-bert: Improving pre-trained transformers with syntax trees. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3011–3020, 2021.
- [Bakarov, 2018] Amir Bakarov. A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536*, 2018.
- [Banarescu et al., 2013] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, 2013.
- [Bao et al., 2021] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Bapna et al., 2018] Ankur Bapna, Mia Xu Chen, Orhan Firat, Yuan Cao, and Yonghui Wu. Training deeper neural machine translation models with transparent attention. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3028–3033, 2018.
- [Barber, 2012] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.
- [Barham et al., 2022] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In *Proceedings of Machine Learning and Systems*, volume 4, pages 430–449, 2022.
- [Baroni and Lenci, 2010] Marco Baroni and Alessandro Lenci. Distributional memory: A general framework for corpus-based semantics. *Computational Linguistics*, 36(4):673–721, 2010.
- [Baroni et al., 2014] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don’t count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, 2014.
- [Barrault et al., 2020] Loïc Barrault, Magdalena Biesialska, Ondřej Bojar, Marta R. Costa-jussà, Christian Federmann, Yvette Graham, Roman Grundkiewicz, Barry Haddow, Matthias Huck, Eric Joanis, Tom Kocmi, Philipp Koehn, Chi-kiu Lo, Nikola Ljubešić, Christof Monz, Makoto Morishita, Masaaki Nagata, Toshiaki Nakazawa, Santanu Pal, Matt Post, and Marcos Zampieri. Findings of

- the 2020 conference on machine translation (WMT20). In *Proceedings of the Fifth Conference on Machine Translation*, pages 1–55, 2020.
- [Baum and Petrie, 1966] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.
- [Baum et al., 1970] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164–171, 1970.
- [Belinkov, 2022] Yonatan Belinkov. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48(1):207–219, 2022.
- [Belinkov and Bisk, 2018] Yonatan Belinkov and Yonatan Bisk. Synthetic and natural noise both break neural machine translation. In *International Conference on Learning Representations*, 2018.
- [Bello, 2020] Irwan Bello. Lambdanetworks: Modeling long-range interactions without attention. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Beltagy et al., 2020] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.
- [Bengio et al., 2015] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.
- [Bengio, 1991] Yoshua Bengio. *Artificial neural networks and their application to sequence recognition*. PhD thesis, McGill University, 1991.
- [Bengio et al., 1994] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [Bengio et al., 2000] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in Neural Information Processing Systems*, 13, 2000.
- [Bengio et al., 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003a.
- [Bengio et al., 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003b.
- [Bengio et al., 2006] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.
- [Bengio et al., 2013] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
- [Bengio et al., 2024] Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Trevor Darrell, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, Gillian K. Hadfield, Jeff Clune, Tegan Maharaj, Frank Hutter, Atilim Gunes Baydin, Sheila A. McIlraith, Qiqi Gao, Ashwin Acharya, David Krueger, Anca Dragan, Philip Torr, Stuart Russell, Daniel Kahneman, Jan Markus Brauner, and Sören Mindermann. Managing extreme ai risks amid rapid progress. *Science*, 384(6698):842–845, 2024.
- [Bentivogli and Giampiccolo, 2011] Luisa Bentivogli and Danilo Giampiccolo. Pascal recognizing

- textual entailment challenge (rte-7) at tac 2011. <https://tac.nist.gov/2011/RTE/>, 2011.
- [Berg-Kirkpatrick et al., 2012] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. An empirical investigation of statistical significance in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005, 2012.
- [Besta et al., 2024] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoeffler. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.
- [Bhattachamishra et al., 2020] Satwik Bhattachamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, 2020.
- [Bhattachasali et al., 2020] Shohini Bhattachasali, Jonathan Brennan, Wen-Ming Luh, Berta Franzluebbers, and John Hale. The alice datasets: fMRI & EEG observations of natural language comprehension. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 120–125, 2020.
- [Bickel and Doksum, 2015] Peter J Bickel and Kjell A Doksum. *Mathematical statistics: basic ideas and selected topics, volumes I-II package*. Chapman and Hall/CRC, 2015.
- [Biderman et al., 2021] Stella Biderman, Sid Black, Charles Foster, Leo Gao, Eric Hallahan, Horace He, Ben Wang, and Phil Wang. Rotary embeddings: A relative revolution. <https://blog.eleuther.ai/rotary-embeddings/>, 2021.
- [Birch et al., 2018] Alexandra Birch, Andrew Finch, Minh-Thang Luong, Graham Neubig, and Yusuke Oda. Findings of the second workshop on neural machine translation and generation. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, pages 1–10, 2018.
- [Bishop, 1995] Christopher Bishop. Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN’95*, pages 141–148, 1995a.
- [Bishop, 1995] Christopher M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1):108–116, 1995b.
- [Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [Blacoe and Lapata, 2012] William Blacoe and Mirella Lapata. A comparison of vector-based representations for semantic composition. In *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*, pages 546–556, 2012.
- [Blei, 2012] David M Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.
- [Blei et al., 2003] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.
- [Blum and Mitchell, 1998] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.
- [Bojanowski et al., 2017] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational*

*linguistics*, 5:135–146, 2017.

- [Bommasani et al., 2021] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, S. Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen A. Creel, Jared Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren E. Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas F. Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, O. Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir P. Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Benjamin Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, J. F. Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Robert Reich, Hongyu Ren, Frieda Rong, Yusuf H. Roohani, Camilo Ruiz, Jack Ryan, Christopher R’e, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishna Parasuram Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei A. Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *ArXiv*, 2021.
- [Bondarenko et al., 2021] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and overcoming the challenges of efficient transformer quantization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7947–7969, 2021.
- [Borji and Itti, 2012] Ali Borji and Laurent Itti. State-of-the-art in visual attention modeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):185–207, 2012.
- [Boulanger-Lewandowski et al., 2013] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Audio chord recognition with recurrent neural networks. In *Proceedings of 14th International Society for Music Information Retrieval Conference*, 2013.
- [Bourlard and Wellekens, 1990] Herve Bourlard and Christian J Wellekens. Links between markov models and multilayer perceptrons. *IEEE Transactions on pattern analysis and machine intelligence*, 12(12):1167–1178, 1990.
- [Bourlard and Morgan, 1993] Herve A. Bourlard and Nelson Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, USA, 1993.
- [Box et al., 2015] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control (4th ed.)*. John Wiley & Sons, 2015.
- [Bradley and Terry, 1952] Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- [Brandon et al., 2024] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*, 2024.
- [Breiman, 1996] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
- [Brill, 1992] Eric Brill. A simple rule-based part of speech tagger. In *Speech and Natural Language:*

- Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.
- [Briski, 2025] Kari Briski. How scaling laws drive smarter, more powerful ai, 2025.
- [Brown et al., 2024] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.
- [Brown et al., 1993] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.
- [Brown et al., 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.
- [Buckman et al., 2016] Jacob Buckman, Miguel Ballesteros, and Chris Dyer. Transition-based dependency parsing with heuristic backtracking. In *Proceedings of the 2016 Conference on empirical methods in natural language processing*, pages 2313–2318, 2016.
- [Bulatov et al., 2022] Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. *Advances in Neural Information Processing Systems*, 35:11079–11091, 2022.
- [Burchi and Vielzeuf, 2021] Maxime Burchi and Valentin Vielzeuf. Efficient conformer: Progressive downsampling and grouped attention for automatic speech recognition. In *Proceedings of 2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 8–15. IEEE, 2021.
- [Burges et al., 2005] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.
- [Burnham and Anderson, 2002] Kenneth P. Burnham and David R. Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer, 2002.
- [Burns et al., 2023] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023a.
- [Burns et al., 2023] Collin Burns, Jan Leike, Leopold Aschenbrenner, Jeffrey Wu, Pavel Izmailov, Leo Gao, Bowen Baker, and Jan Hendrik Kirchner. Weak-to-strong generalization, 2023b. URL <https://openai.com/index/weak-to-strong-generalization>.
- [Buttcher et al., 2016] Stefan Buttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. MIT Press, 2016.
- [Caballero et al., 2023] Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken neural

- scaling laws. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2023.
- [Campbell, 1997] Joseph P Campbell. Speaker recognition: A tutorial. *Proceedings of the IEEE*, 85(9): 1437–1462, 1997.
- [Cao et al., 2007] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.
- [Caron et al., 2021] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9650–9660, 2021.
- [Casacuberta et al., 2009] Francisco Casacuberta, Jorge Civera, Elsa Cubel, Antonio L Lagarda, Guy Lapalme, Elliott Macklovitch, and Enrique Vidal. Human interaction for high-quality machine translation. *Communications of the ACM*, 52(10):135–138, 2009.
- [Cer et al., 2018] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.
- [Chan et al., 2016] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4960–4964. IEEE, 2016.
- [Chang et al., 2024] Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077*, 2024.
- [Chang, 1967] Wing-Tsit Chang. *Reflections on things at hand*. Columbia University Press, 1967.
- [Chang and Collins, 2011] Yin-Wen Chang and Michael Collins. Exact decoding of phrase-based translation models through lagrangian relaxation. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 26–37, 2011.
- [Charniak, 1997] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997.
- [Chatfield, 2003] Chris Chatfield. *The analysis of time series: an introduction*. Chapman and hall/CRC, 2003.
- [Chaudhari et al., 2021] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. An attentive survey of attention models. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 12(5):1–32, 2021.
- [Chen et al., 2023] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023a.
- [Chen et al., 2018] Kehai Chen, Rui Wang, Masao Utiyama, Eiichiro Sumita, and Tiejun Zhao. Syntax-directed attention for neural machine translation. In *Proceedings of the AAAI conference on artificial intelligence*, 2018a.
- [Chen et al., 2023] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpapasus: Training a better alpaca with fewer data. *arXiv preprint arXiv:2307.08701*, 2023b.

- [Chen et al., 2024] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpapasus: Training a better alpaca with fewer data. In *The Twelfth International Conference on Learning Representations*, 2024a.
- [Chen et al., 2020] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International conference on machine learning*, pages 1691–1703. PMLR, 2020a.
- [Chen et al., 2018] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Mike Schuster, Noam Shazeer, Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 76–86, 2018b.
- [Chen et al., 2018] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018c.
- [Chen et al., 2022] Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, Long Zhou, Shuo Ren, Yanmin Qian, Yao Qian, Jian Wu, Michael Zeng, and Furu Wei. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.
- [Chen et al., 2023] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023c.
- [Chen and Goodman, 1999] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13:359–394, 1999.
- [Chen et al., 2020] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in neural information processing systems*, 33:15834–15846, 2020b.
- [Chen et al., 2015] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.
- [Chen and He, 2021] Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15750–15758, 2021.
- [Chen et al., 2020] Yen-Chun Chen, Linjie Li, Licheng Yu, Ahmed El Kholy, Faisal Ahmed, Zhe Gan, Yu Cheng, and Jingjing Liu. Uniter: Universal image-text representation learning. In *Proceedings of European conference on computer vision*, pages 104–120, 2020c.
- [Chen et al., 2024] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024b.
- [Chevalier et al., 2023] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3829–3846, 2023.

- [Chi et al., 2022] Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. Kerple: Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems*, 35:8386–8399, 2022.
- [Chi et al., 2023] Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. Dissecting transformer length extrapolation via the lens of receptive field analysis. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13522–13537, 2023.
- [Chiang, 2005] David Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd annual meeting of the association for computational linguistics (acl’05)*, pages 263–270, 2005.
- [Chiang, 2007] David Chiang. Hierarchical phrase-based translation. *computational linguistics*, 33(2): 201–228, 2007.
- [Chiang and Cholak, 2022] David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.
- [Chiang et al., 2023] David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders. *arXiv preprint arXiv:2301.10743*, 2023a.
- [Chiang et al., 2023] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%\* chatgpt quality, March 2023b. URL <https://lmsys.org/blog/2023-03-30-vicuna/>.
- [Child et al., 2019] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.
- [Chiu and Raffel, 2018] Chung-Cheng Chiu and Colin Raffel. Monotonic chunkwise attention. In *Proceedings of the 8th International Conference on Learning Representations ICLR*, 2018.
- [Cho et al., 2021] Jaemin Cho, Jie Lei, Hao Tan, and Mohit Bansal. Unifying vision-and-language tasks via text generation. In *International Conference on Machine Learning*, pages 1931–1942. PMLR, 2021.
- [Cho and Esipova, 2016] Kyunghyun Cho and Masha Esipova. Can neural machine translation do simultaneous translation? *arXiv preprint arXiv:1606.02012*, 2016.
- [Cho et al., 2014] Kyunghyun Cho, Bart van Merriënboer, Çağlar Guülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.
- [Choe and Charniak, 2016] Do Kook Choe and Eugene Charniak. Parsing as language modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2331–2336, 2016.
- [Chollet, 2021] François Chollet. *Deep Learning with Python (2nd ed.)*. Manning Publications, 2021.
- [Choromanski et al., 2020] Krzysztof Marcin Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Chorowski and Jaitly, 2017] Jan Chorowski and Navdeep Jaitly. Towards better decoding and language

- model integration in sequence to sequence models. *Proc. Interspeech 2017*, pages 523–527, 2017.
- [Chorowski et al., 2019] Jan Chorowski, Ron J Weiss, Samy Bengio, and Aäron Van Den Oord. Unsupervised speech representation learning using wavenet autoencoders. *IEEE/ACM transactions on audio, speech, and language processing*, 27(12):2041–2053, 2019.
- [Chorowski et al., 2015] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. *Advances in neural information processing systems*, 28, 2015.
- [Chowdhery et al., 2022] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [Christiano et al., 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.
- [Chu et al., 2023] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*, 2023.
- [Chuang et al., 2020] Yung-Sung Chuang, Chi-Liang Liu, Hung-yi Lee, and Lin-shan Lee. Speechbert: An audio-and-text jointly learned language model for end-to-end spoken question answering. In *Proceedings of Interspeech 2020*, pages 4168–4172, 2020.
- [Chung et al., 2022] Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed Huai hsin Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.
- [Chung et al., 2014] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Proceedings of NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [Church, 2011] Kenneth Church. A pendulum swung too far. *Linguistic Issues in Language Technology*, 6, 2011.
- [Church and Hanks, 1990] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29, 1990. URL <https://aclanthology.org/J90-1003>.

- [Clark et al., 2019] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert’s attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, 2019a.
- [Clark et al., 2019] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *Proceedings of International Conference on Learning Representations*, 2019b.
- [Clark et al., 2008] Stephen Clark, Bob Coecke, and Mehrnoosh Sadzadeh. A compositional distributional model of meaning. In *Proceedings of the Second Quantum Interaction Symposium (QI-2008)*, pages 133–140. Oxford, 2008.
- [Cline and Dhillon, 2014] Alan Kaylor Cline and Inderjit S. Dhillon. Computation of the singular value decomposition. In Leslie Hogben, editor, *Handbook of Linear Algebra (2nd ed.)*. CRC Press, 2014.
- [Cobbe et al., 2021] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- [Cohn et al., 2016] Trevor Cohn, Cong Duy Vu Hoang, Ekaterina Vymolova, Kaisheng Yao, Chris Dyer, and Gholamreza Haffari. Incorporating structural alignment biases into an attentional neural translation model. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 876–885, 2016.
- [Collobert and Weston, 2008] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICML08)*, pages 160–167, 2008.
- [Conneau et al., 2017] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, 2017a.
- [Conneau et al., 2017] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, 2017b.
- [Conneau et al., 2017] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1107–1116, 2017c.
- [Conneau et al., 2018] Alexis Conneau, Germán Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, 2018.
- [Conneau et al., 2020] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, 2020.
- [Cortes and Vapnik, 1995] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mind*,

- Machine Learning:273–297, 1995.
- [Coste et al., 2024] Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Cotterell and Schütze, 2015] Ryan Cotterell and Hinrich Schütze. Morphological word-embeddings. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1287–1292, 2015.
- [Cui et al., 2024] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. ULTRAFEEDBACK: Boosting language models with scaled AI feedback. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 9722–9744, 2024.
- [Currey and Heafield, 2018] Anna Currey and Kenneth Heafield. Multi-source syntactic neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2961–2966, 2018.
- [Cybenko, 1989] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [Dai et al., 2023] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4005–4019, 2023.
- [Dai et al., 2019] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.
- [Dao et al., 2022] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
- [Dao et al., 2023] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. <https://pytorch.org/blog/flash-decoding/>, 2023. Retrieved 2023-10-23.
- [Davis and Mermelstein, 1980] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE transactions on acoustics, speech, and signal processing*, 28(4):357–366, 1980.
- [Dayan et al., 1995] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.
- [de Gispert et al., 2010] Adrià de Gispert, Gonzalo Iglesias, Graeme Blackwood, Eduardo R. Banga, and William Byrne. Hierarchical phrase-based translation with weighted finite-state transducers and shallow-n grammars. *Computational linguistics*, 36(3):505–533, 2010.
- [Deepseek, 2025] Deepseek. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [Deerwester et al., 1990] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

- [Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.
- [Del Corro et al., 2023] Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *arXiv preprint arXiv:2307.02628*, 2023.
- [Deletang et al., 2024] Gregoire Deletang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Dempster et al., 1977] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.
- [Deng et al., 2022] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3369–3391, 2022.
- [Deoras et al., 2011] Anoop Deoras, Tomáš Mikolov, and Kenneth Church. A fast re-scoring strategy to capture long-distance dependencies. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1116–1127, 2011.
- [Devereux et al., 2010] Barry Devereux, Colin Kelly, and Anna Korhonen. Using fmri activation to conceptual stimuli to evaluate methods for extracting conceptual representations from corpora. In *Proceedings of the NAACL HLT 2010 First Workshop on Computational Neurolinguistics*, pages 70–78, 2010.
- [Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.
- [Di Gangi et al., 2019] Mattia Antonino Di Gangi, Matteo Negri, Roldano Cattoni, Roberto Dessi, and Marco Turchi. Enhancing transformer for end-to-end speech-to-text translation. In *Proceedings of Machine Translation Summit XVII: Research Track*, pages 21–31, 2019.
- [Ding et al., 2021] Ming Ding, Zhuoyi Yang, Wenyi Hong, Wendi Zheng, Chang Zhou, Da Yin, Junyang Lin, Xu Zou, Zhou Shao, Hongxia Yang, and Jie Tang. Cogview: Mastering text-to-image generation via transformers. *Advances in Neural Information Processing Systems*, 34:19822–19835, 2021.
- [Ding et al., 2024] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.
- [Doersch, 2016] Carl Doersch. Tutorial on variational autoencoders. *stat*, 1050:13, 2016.
- [Dolan and Brockett, 2005] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of Third International Workshop on Paraphrasing (IWP2005)*, 2005.
- [Dong et al., 2019] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language

- understanding and generation. *Advances in neural information processing systems*, 32, 2019.
- [Dong et al., 2022] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.
- [Dong et al., 2021] Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In *International Conference on Machine Learning*, pages 2793–2803. PMLR, 2021.
- [Dosovitskiy et al., 2021] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of ICLR 2021*, 2021.
- [Downey, 2021] Allen B. Downey. *Think Bayes: Bayesian Statistics in Python (2nd ed.)*. O’Reilly Media, 2021.
- [Dror et al., 2020] Rotem Dror, Lotem Peled-Cohen, and Segev Shlomov. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers, 2020.
- [Drozdoz et al., 2022] Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. Compositional semantic parsing with large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2022.
- [Dua et al., 2022] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1251–1265, 2022.
- [Dubey et al., 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [Dubois et al., 2024] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. AlpacaFarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Duchi et al., 2011] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.
- [Dufter et al., 2022] Philipp Dufter, Martin Schmitt, and Hinrich Schütze. Position information in transformers: An overview. *Computational Linguistics*, 48(3):733–763, 2022.
- [Dyer et al., 2013] Chris Dyer, Victor Chahuneau, and Noah A Smith. A simple, fast, and effective reparameterization of IBM model 2. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 644–648, 2013.
- [Ebrahimi et al., 2018] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 31–36, Melbourne, Australia, 2018.
- [Edunov et al., 2018] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding

- back-translation at scale. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 489–500, 2018.
- [Ee, 2017] Weinan Ee. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5:1–11, 02 2017.
- [Eikema and Aziz, 2020] Bryan Eikema and Wilker Aziz. Is map decoding all you need? the inadequacy of the mode in neural machine translation. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 4506–4520, 2020.
- [Eisenstein et al., 2023] Jacob Eisenstein, Chirag Nagpal, Alekh Agarwal, Ahmad Beirami, Alex D’Amour, DJ Dvijotham, Adam Fisch, Katherine Heller, Stephen Pfohl, Deepak Ramachandran, and Peter Shaw. Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking. *arXiv preprint arXiv:2312.09244*, 2023.
- [Elbayad et al., 2020] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Elman, 1990] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- [Elsken et al., 2019] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019a.
- [Elsken et al., 2019] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019b.
- [Erhan et al., 2010] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208, 2010.
- [Fan et al., 2018] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, 2018.
- [Fan et al., 2019] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *Proceedings of International Conference on Learning Representations*, 2019.
- [Fan et al., 2021] Haoqi Fan, Bo Xiong, Karttikeya Mangalam, Yanghao Li, Zhicheng Yan, Jitendra Malik, and Christoph Feichtenhofer. Multiscale vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6824–6835, 2021.
- [Fan et al., 2020] Yang Fan, Shufang Xie, Yingce Xia, Lijun Wu, Tao Qin, Xiang-Yang Li, and Tie-Yan Liu. Multi-branch attentive transformer. *arXiv preprint arXiv:2006.10270*, 2020.
- [Faruqui et al., 2016] Manaal Faruqui, Yulia Tsvetkov, Pushpendre Rastogi, and Chris Dyer. Problems with evaluation of word embeddings using word similarity tasks. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 30–35, 2016.
- [Fedus et al., 2022] William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667*, 2022a.
- [Fedus et al., 2022] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022b.
- [Fellbaum, 2005] Christiane Fellbaum. Wordnet and wordnets. In Keith Brown, editor, *Encyclopedia*

- of Language and Linguistics (2nd ed.)*. Elsevier, 2005.
- [Feng et al., 2016] Shi Feng, Shujie Liu, Nan Yang, Mu Li, Ming Zhou, and Kenny Zhu. Improving attention modeling with implicit distortion and fertility for machine translation. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3082–3092, 2016.
- [Feng et al., 2021] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, 2021.
- [Fernandes et al., 2023] Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. Bridging the gap: A survey on integrating (human) feedback for natural language generation. *Transactions of the Association for Computational Linguistics*, 11: 1643–1668, 2023.
- [Firth, 1957] John R Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.
- [Forney, 1972] GDJR Forney. Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. *IEEE Transactions on Information theory*, 18(3):363–378, 1972.
- [Franklin and Graesser, 1996] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International workshop on agent theories, architectures, and languages*, pages 21–35. Springer, 1996.
- [Freedman et al., 2007] David Freedman, Robert Pisani, and Roger Purves. *Statistics (4th ed.)*. W. W. Norton & Company, 2007.
- [Freedman, 2009] David A. Freedman. *Statistical Models: Theory and Practice (2nd ed.)*. Cambridge University Press, 2009.
- [Freitag and Al-Onaizan, 2017] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, 2017.
- [Freitag et al., 2022] Markus Freitag, David Grangier, Qijun Tan, and Bowen Liang. High quality rather than high model probability: Minimum bayes risk decoding with neural metrics. *Transactions of the Association for Computational Linguistics*, 10:811–825, 2022.
- [Frensch and Funke, 2014] Peter A Frensch and Joachim Funke. *Complex problem solving: The European perspective*. Psychology Press, 2014.
- [Friedl, 2006] Jeffrey Friedl. *Mastering Regular Expressions (3rd ed.)*. O’Reilly Media, 2006.
- [Fu et al., 2022] Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. Hungry hungry hippos: Towards language modeling with state space models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2022.
- [Fuller, 2009] Wayne A Fuller. *Introduction to statistical time series*. John Wiley & Sons, 2009.
- [Gage, 1994] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.
- [Gale et al., 2019] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural

- networks. *arXiv preprint arXiv:1902.09574*, 2019.
- [Ganguli et al., 2023] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamile Lukosiute, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, Dawn Drain, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jackson Kernion, Jamie Kerr, Jared Mueller, Joshua Landau, Kamal Ndousse, Karina Nguyen, Liane Lovitt, Michael Sellitto, Nelson Elhage, Noemí Mercado, Nova DasSarma, Oliver Rausch, Robert Lasenby, Robin Larson, Sam Ringer, Sandipan Kundu, Saurav Kadavath, Scott Johnston, Shauna Kravec, Sheer El Showk, Tamera Lanham, Timothy Telleen-Lawton, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, Christopher Olah, Jack Clark, Samuel R. Bowman, and Jared Kaplan. The capacity for moral self-correction in large language models. *arXiv preprint arXiv:2302.07459*, 2023.
- [Gao et al., 2023] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR, 2023a.
- [Gao et al., 2023] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023b.
- [Gao et al., 2023] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023c.
- [Garg et al., 2019] Sarthak Garg, Stephan Peitz, Udhyakumar Nallasamy, and Matthias Paulik. Jointly learning to align and translate with transformer models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4453–4462, 2019.
- [Garg et al., 2022] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.
- [Ge et al., 2024] Yuan Ge, Yilun Liu, Chi Hu, Weibin Meng, Shimin Tao, Xiaofeng Zhao, Hongxia Ma, Li Zhang, Boxing Chen, Hao Yang, Bei Li, Tong Xiao, and Jingbo Zhu. Clustering and ranking: Diversity-preserved instruction selection through expert-aligned quality estimation. *arXiv preprint arXiv:2402.18191*, 2024.
- [Gehring et al., 2017] Jonas Gehring, Michael Auli, David Grangier, and Yann Dauphin. A convolutional encoder model for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 123–135, 2017a.
- [Gehring et al., 2017] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International conference on machine learning*, pages 1243–1252. PMLR, 2017b.
- [Gelman et al., 2020] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis (2nd ed.)*. Chapman and Hall/CRC, 2020.
- [Gemma Team, 2024] Google DeepMind Gemma Team. Gemma: Open Models Based on Gemini Research and Technology, 2024.
- [Germann et al., 2004] Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. Fast and optimal decoding for machine translation. *Artificial Intelligence*, 154(1-2):127–143, 2004.

- [Géron, 2019] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems (2nd ed.)*. O’Reilly Media, 2019.
- [Gers et al., 2000] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.
- [Ghazvininejad et al., 2019] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-predict: Parallel decoding of conditional masked language models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6112–6121, 2019.
- [Gholami et al., 2022] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC, 2022.
- [Gildea and Jurafsky, 2002] Daniel Gildea and Daniel Jurafsky. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288, 2002.
- [Gladkova et al., 2016] Anna Gladkova, Aleksandr Drozd, and Satoshi Matsuoka. Analogy-based detection of morphological and semantic relations with word embeddings: what works and what doesn’t. In *Proceedings of the NAACL Student Research Workshop*, pages 8–15, 2016.
- [Glorot and Bengio, 2010] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [Goel and Byrne, 2000] Vaibhava Goel and William J Byrne. Minimum bayes-risk automatic speech recognition. *Computer Speech & Language*, 14(2):115–135, 2000.
- [Gomez et al., 2017] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems*, 30, 2017.
- [Goodfellow et al., 2015] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.
- [Goodfellow et al., 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [Goodhart, 1984] Charles AE Goodhart. *Problems of monetary management: the UK experience*. Springer, 1984.
- [Goodman, 1996] Joshua Goodman. Parsing algorithms and metrics. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 177–183, 1996a.
- [Goodman, 1996] Joshua Goodman. Parsing algorithms and metrics. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 177–183, 1996b.
- [Gordon et al., 2021] Mitchell A Gordon, Kevin Duh, and Jared Kaplan. Data and parameter scaling laws for neural machine translation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5915–5922, 2021.
- [Gou et al., 2021] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.
- [Gou et al., 2024] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen,

- et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Graves and Jaitly, 2014] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of International conference on machine learning*, pages 1764–1772, 2014.
- [Graves et al., 2006] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.
- [Graves et al., 2013] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *IEEE workshop on automatic speech recognition and understanding*, pages 273–278. IEEE, 2013a.
- [Graves et al., 2013] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013b.
- [Graves et al., 2014] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.
- [Graves et al., 2016] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adriá Puigdoménech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.
- [Gray, 1998] Robert M. Gray. Quantization. *IEEE transactions on information theory*, 44(6):2325–2383, 1998.
- [Grissom II et al., 2014] Alvin Grissom II, He He, Jordan Boyd-Graber, John Morgan, and Hal Daumé III. Don’t until the final verb wait: Reinforcement learning for simultaneous machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1342–1352, 2014.
- [Grover and Leskovec, 2016] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.
- [Gu and Dao, 2023] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.
- [Gu et al., 2021] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Gu et al., 2022] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35:35971–35983, 2022a.
- [Gu et al., 2022] Albert Gu, Karan Goel, Khaled Saab, and Chris Ré. Structured state spaces: Combining continuous-time, recurrent, and convolutional models. <https://hazyresearch.stanford.>

- [edu/blog/2022-01-14-s4-3](https://openai.com/blog/2022-01-14-s4-3), 2022b. Retrieved 2022-01-14.
- [Gu et al., 2017] Jiatao Gu, Graham Neubig, Kyunghyun Cho, and Victor O.K. Li. Learning to translate in real-time with neural machine translation. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL) Conference, 2017*, 2017.
- [Gu et al., 2018] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. Non-autoregressive neural machine translation. In *Proceedings of International Conference on Learning Representations*, 2018.
- [Gulati et al., 2020] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. *Proceedings of Interspeech 2020*, pages 5036–5040, 2020.
- [Gulcehre et al., 2016] Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. Noisy activation functions. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 3059–3068. PMLR, 2016.
- [Gulcehre et al., 2017] Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, and Yoshua Bengio. On integrating a language model into neural machine translation. *Computer Speech & Language*, 45: 137–148, 2017.
- [Gunasekar et al., 2023] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.
- [Guo et al., 2019] Maosheng Guo, Yu Zhang, and Ting Liu. Gaussian transformer: a lightweight approach for natural language inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6489–6496, 2019.
- [Guo et al., 2024] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Guo et al., 2020] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Xiangyang Xue, and Zheng Zhang. Multi-scale self-attention for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7847–7854, 2020.
- [Gupta and Berant, 2020] Ankit Gupta and Jonathan Berant. Gmat: Global memory augmentation for transformers. *arXiv preprint arXiv:2006.03274*, 2020.
- [Gupta et al., 2021] Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, pages 39–52, 2021.
- [Gupta et al., 2004] Madan Gupta, Liang Jin, and Noriyasu Homma. *Static and dynamic neural networks: from fundamentals to advanced theory*. John Wiley & Sons, 2004.
- [Guu et al., 2020] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *Proceedings of International conference on*

- machine learning*, pages 3929–3938. PMLR, 2020.
- [Guyon and Elisseff, 2003] Isabelle Guyon and André Elisseff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.
- [Haber and Ruthotto, 2017] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse problems*, 34(1):014004, 2017.
- [Hahn, 2020] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.
- [Hamilton, 1994] James Douglas Hamilton. *Time Series Analysis*. Princeton University Press, 1994.
- [Han et al., 2022] Kai Han, Yunhe Wang, Hanqing Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence*, 45(1): 87–110, 2022.
- [Han et al., 2020] Wei Han, Zhengdong Zhang, Yu Zhang, Jiahui Yu, Chung-Cheng Chiu, James Qin, Anmol Gulati, Ruoming Pang, and Yonghui Wu. Contextnet: Improving convolutional neural networks for automatic speech recognition with global context. In *Proceedings of Interspeech 2020*, pages 3610–3614, 2020.
- [Han et al., 2021] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. Pre-trained models: Past, present and future. *AI Open*, 2:225–250, 2021a.
- [Han et al., 2021] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021b.
- [Han et al., 2024] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.
- [Hannun et al., 2014] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, and Adam Coates. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.
- [Hao et al., 2019] Jie Hao, Xing Wang, Shuming Shi, Jinfeng Zhang, and Zhaopeng Tu. Multi-granularity self-attention for neural machine translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 887–897, 2019.
- [Hao et al., 2022] Yiding Hao, Dana Angluin, and Robert Frank. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810, 2022.
- [Harlap et al., 2018] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.
- [Harris, 1954] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.
- [Hasler et al., 2018] Eva Hasler, Adrià de Gispert, Gonzalo Iglesias, and Bill Byrne. Neural machine translation decoding with terminology constraints. In *Proceedings of the 2018 Conference of the*

- North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 506–512, 2018.
- [Hastie et al., 2009] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2009.
- [He et al., 2017] Di He, Hanqing Lu, Yingce Xia, Tao Qin, Liwei Wang, and Tie-Yan Liu. Decoding with value networks for neural machine translation. *Advances in Neural Information Processing Systems*, 30, 2017.
- [He et al., 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [He et al., 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.
- [He et al., 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Proceedings of ECCV 2016*, pages 630–645, 2016b.
- [He et al., 2019] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.
- [He et al., 2022] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.
- [He et al., 2021] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced bert with disentangled attention. In *Proceedings of International Conference on Learning Representations*, 2021.
- [He et al., 2016] Wei He, Zhongjun He, Hua Wu, and Haifeng Wang. Improved neural machine translation with smt features. In *Proceedings of the Thirtieth AAAI conference on artificial intelligence*, 2016c.
- [He et al., 2018] Xuanli He, Gholamreza Haffari, and Mohammad Norouzi. Sequence to sequence mixture model for diverse machine translation. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*, pages 583–592, 2018.
- [Heafield et al., 2021] Kenneth Heafield, Qianqian Zhu, and Roman Grundkiewicz. Findings of the WMT 2021 shared task on efficient translation. In *Proceedings of the Sixth Conference on Machine Translation*, pages 639–651, 2021.
- [Hendrycks and Gimpel, 2016] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [Hendrycks et al., 2020] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzić, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2744–2751, 2020.
- [Hendrycks et al., 2021] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *Proceedings of International Conference on Learning Representations*, 2021.

- [Hestness et al., 2017] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.
- [Hewitt, 2024] John Hewitt. Instruction following without instruction tuning, 2024. URL <https://nlp.stanford.edu/~johnhew/instruction-following.html>.
- [Hewitt and Liang, 2019] John Hewitt and Percy Liang. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743, 2019.
- [Hewitt et al., 2024] John Hewitt, Nelson F Liu, Percy Liang, and Christopher D Manning. Instruction following without instruction tuning. *arXiv preprint arXiv:2409.14254*, 2024.
- [Hildebrand and Vogel, 2008] Almut Silja Hildebrand and Stephan Vogel. Combination of machine translation systems via hypothesis selection from combined n-best lists. In *Proceedings of the 8th Conference of the Association for Machine Translation in the Americas: Student Research Workshop*, pages 254–261, 2008.
- [Hill et al., 2016] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1367–1377, 2016.
- [Hinton, 2018] Geoff Hinton. Coursera neural networks for machine learning lecture 6, 2018. URL [http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf).
- [Hinton et al., 2015] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- [Hinton, 2007] Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007.
- [Hinton and Roweis, 2002] Geoffrey E Hinton and Sam Roweis. Stochastic neighbor embedding. *Advances in neural information processing systems*, 15, 2002.
- [Hinton and Salakhutdinov, 2006] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
- [Hinton et al., 2006] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- [Hinton et al., 2012] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- [Hoang et al., 2017] Cong Duy Vu Hoang, Gholamreza Haffari, and Trevor Cohn. Towards decoding as continuous optimisation in neural machine translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 146–156, 2017.
- [Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [Hoffmann et al., 2022] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan

- Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- [Hokamp and Liu, 2017] Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, 2017.
- [Holmström and Koistinen, 1992] Lasse Holmström and Petri Koistinen. Using additive noise in back-propagation training. *IEEE Transactions on Neural Networks*, 3(1):24–38, 1992.
- [Holtzman et al., 2020] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *Proceedings of the 6th International Conference on Learning Representations ICLR*, 2020a.
- [Holtzman et al., 2020] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020b.
- [Honovich et al., 2023] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14409–14428, 2023.
- [Hopfield, 1982] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.
- [Hopfield, 1984] John J Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the national academy of sciences*, 81(10):3088–3092, 1984.
- [Hou et al., 2020] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33:9782–9793, 2020.
- [Houlsby et al., 2019] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.
- [Howard et al., 2019] Andrew Howard, Ruoming Pang, Hartwig Adam, Quoc V. Le, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, and Yukun Zhu. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.
- [Hsu et al., 2021] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:3451–3460, 2021.
- [Hu et al., 2021] Chi Hu, Chenglong Wang, Xiangnan Ma, Xia Meng, Yinqiao Li, Tong Xiao, Jingbo Zhu, and Changliang Li. Ranknas: Efficient neural architecture search by pairwise ranking. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2469–2480, 2021.

- [Hu et al., 2022] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.
- [Huang et al., 2018] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer: Generating music with long-term structure. In *Proceedings of International Conference on Learning Representations*, 2018.
- [Huang et al., 2012] Eric H Huang, Richard Socher, Christopher D Manning, and Andrew Y Ng. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 873–882, 2012.
- [Huang et al., 2016] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *Proceedings of the 14th European Conference*, pages 646–661. Springer, 2016.
- [Huang et al., 2017] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017a.
- [Huang, 2009] Liang Huang. Dynamic programming-based search algorithms in NLP. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, 2009.
- [Huang et al., 2017] Liang Huang, Kai Zhao, and Mingbo Ma. When to finish? optimal beam search for neural text generation (modulo beam size). In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2134–2139, 2017b.
- [Huang et al., 2020] Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving transformer optimization through better initialization. In *Proceedings of International Conference on Machine Learning*, pages 4475–4483. PMLR, 2020a.
- [Huang et al., 2019] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyounJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- [Huang et al., 2015] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.
- [Huang et al., 2020] Zhiheng Huang, Davis Liang, Peng Xu, and Bing Xiang. Improve transformer models with better relative position embeddings. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3327–3335, 2020b.
- [Hubel and Wiesel, 1959] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat’s striate cortex. *The Journal of physiology*, 148(3):574, 1959.
- [Hurley, 2011] Patrick Hurley. *A Concise Introduction to Logic (11th ed.)*. Wadsworth Publishing, 2011.
- [Hutchins et al., 2022] DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. *Advances in neural information processing systems*, 35: 33248–33261, 2022.

- [Hutchison et al., 2013] Keith A Hutchison, David A Balota, James H Neely, Michael J Cortese, Emily R Cohen-Shikora, Chi-Shing Tse, Melvin J Yap, Jesse J Bengson, Dale Niemeyer, and Erin Buchanan. The semantic priming project. *Behavior research methods*, 45(4):1099–1114, 2013.
- [Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [Ivanov et al., 2021] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. In *Proceedings of Machine Learning and Systems*, volume 3, pages 711–732, 2021.
- [Jackendoff, 1992] Ray S Jackendoff. *Semantic structures*, volume 18. MIT press, 1992.
- [Jacob et al., 2018] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
- [Jaderberg et al., 2015] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and koray kavukcuoglu. Spatial transformer networks. *Advances in neural information processing systems*, 28, 2015.
- [Jaeger and Haas, 2004] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80, 2004.
- [Jaegle et al., 2021] Andrew Jaegle, Sebastian Borgeaud, Jean-Baptiste Alayrac, Carl Doersch, Catalin Ionescu, David Ding, Skanda Koppula, Daniel Zoran, Andrew Brock, Evan Shelhamer, Olivier J. Hénaff, Matthew M. Botvinick, Andrew Zisserman, Oriol Vinyals, and João Carreira. Perceiver io: A general architecture for structured inputs & outputs. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Janssen, 2012] Theo M.V. Janssen. Compositionality: its historic context. In M. Werning, W. Hinzen, and E. Machery, editors, *The Oxford handbook of compositionality*. Oxford University Press, 2012.
- [Jean et al., 2015] Sébastien Jean, Orhan Firat, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. Montreal neural machine translation systems for wmt’15. In *Proceedings of the tenth workshop on statistical machine translation*, pages 134–140, 2015.
- [Jelinek, 1998] Frederick Jelinek. *Statistical methods for speech recognition*. MIT Press, 1998.
- [Jia and Liang, 2017] Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2021–2031, 2017.
- [Jiang et al., 2023] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Deendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023a.
- [Jiang et al., 2023] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LlmLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, 2023b.
- [Jiang et al., 2020] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:

- 423–438, 2020.
- [Jiao et al., 2020] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, 2020.
- [Jolliffe, 2002] Ian T Jolliffe. *Principal component analysis for special types of data*. Springer, 2002.
- [Joshi et al., 2017] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.
- [Joshi et al., 2020] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics*, 8:64–77, 2020.
- [Joulin et al., 2017] Armand Joulin, Édouard Grave, Piotr Bojanowski, and Tomáš Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, 2017.
- [Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. *Speech and Language Processing (2nd ed.)*. Prentice Hall, 2008.
- [Kahneman, 2011] Daniel Kahneman. *Thinking, fast and slow*. macmillan, 2011.
- [Kalchbrenner et al., 2014] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 655–665, 2014.
- [Kaplan et al., 2020] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [Katharopoulos et al., 2020] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.
- [Kelly and Stone, 1975] Edward F. Kelly and Philip J. Stone. *Computer recognition of English word senses*. American Elsevier Pub, 1975.
- [Kernes, 2021] Jonathan Kernes. Master positional encoding: Part i, 05 2021. URL <https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3>.
- [Khan et al., 2020] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial intelligence review*, 53 (8):5455–5516, 2020.
- [Khandelwal et al., 2019] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2019.
- [Khandelwal et al., 2020] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.

- [Khayrallah et al., 2017] Huda Khayrallah, Gaurav Kumar, Kevin Duh, Matt Post, and Philipp Koehn. Neural lattice search for domain adaptation in machine translation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 20–25, 2017.
- [Khot et al., 2023] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Kidger, 2022] Patrick Kidger. On neural differential equations. *arXiv preprint arXiv:2202.02435*, 2022.
- [Kikuchi et al., 2016] Yuta Kikuchi, Graham Neubig, Ryohei Sasano, Hiroya Takamura, and Manabu Okumura. Controlling output length in neural encoder-decoders. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1328–1338, 2016.
- [Kim and Cho, 2021] Gyuwan Kim and Kyunghyun Cho. Length-adaptive transformer: Train once with length drop, use anytime with search. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6501–6511, 2021.
- [Kim et al., 2019] Najoung Kim, Roma Patel, Adam Poliak, Alex Wang, Patrick Xia, R. Thomas McCoy, Ian Tenney, Alexis Ross, Tal Linzen, Benjamin Van Durme, Samuel R. Bowman, and Ellie Pavlick. Probing what different nlp tasks teach machines about function word comprehension. In *Proceedings of the Eighth Joint Conference on Lexical and Computational Semantics (\*SEM 2019)*, pages 235–249, 2019.
- [Kim et al., 2023] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*, 2023.
- [Kim et al., 2021] Wonjae Kim, Bokyung Son, and Ildoo Kim. Vilt: Vision-and-language transformer without convolution or region supervision. In *Proceedings of International Conference on Machine Learning*, pages 5583–5594. PMLR, 2021.
- [Kim, 2014] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, October 2014.
- [Kim and Rush, 2016] Yoon Kim and Alexander M Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1317–1327, 2016.
- [Kim et al., 2016] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI conference on artificial intelligence*, 2016.
- [Kim and Awadalla, 2020] Young Jin Kim and Hany Hassan Awadalla. Fastformers: Highly efficient transformer models for natural language understanding. In *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*, pages 149–158, 2020.
- [Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

- [Kingma and Welling, 2014] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *Proceedings of 2nd International Conference on Learning Representations, ICLR 2014*, 2014.
- [Kingma and Welling, 2019] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 2019.
- [Kirkpatrick et al., 2017] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharmashan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114 (13):3521–3526, 2017.
- [Kiros et al., 2015] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. *Advances in neural information processing systems*, 28, 2015.
- [Kitaev et al., 2020] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Klein et al., 2017] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, 2017.
- [Klementiev et al., 2012] Alexandre Klementiev, Ivan Titov, and Binod Bhattacharai. Inducing crosslingual distributed representations of words. In *Proceedings of COLING 2012*, pages 1459–1474, 2012.
- [Klerke et al., 2015] Sigrid Klerke, Héctor Martínez Alonso, and Anders Søgaard. Looking hard: Eye tracking for detecting grammaticality of automatically compressed sentences. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, pages 97–105, 2015.
- [Knight, 1999] Kevin Knight. Decoding complexity in word-replacement translation models. *Computational linguistics*, 25(4):607–615, 1999.
- [Knight, 2009] Kevin Knight. Bayesian inference with tears, 2009.
- [Knight, 2018] Linda Knight. The sparrow tweets, 2018.
- [Kochenderfer and Wheeler, 2019] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. The MIT Press, 2019.
- [Koehn, 2004] Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer, 2004.
- [Koehn, 2010] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.
- [Koehn and Knowles, 2017] Philipp Koehn and Rebecca Knowles. Six challenges for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 28–39, 2017.
- [Koehn et al., 2003] Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–133, 2003.
- [Koehn et al., 2007] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical

- machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, 2007.
- [Kojima et al., 2022] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
- [Konishi and Kitagawa, 2007] Sadanori Konishi and Genshiro Kitagawa. *Information Criteria and Statistical Modeling*. Springer, 2007.
- [Korthikanti et al., 2023] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.
- [Krakovna et al., 2020] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. <https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity>, 2020.
- [Krebs et al., 2018] Alicia Krebs, Alessandro Lenci, and Denis Paperno. SemEval-2018 task 10: Capturing discriminative attributes. In *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 732–740, 2018.
- [Krizhevsky et al., 2017] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.
- [Kudo, 2018] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, 2018.
- [Kudo and Richardson, 2018] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018.
- [Kulikov et al., 2019] Iliia Kulikov, Alexander Miller, Kyunghyun Cho, and Jason Weston. Importance of search and evaluation strategies in neural dialogue modeling. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 76–87, 2019.
- [Kulis, 2013] Brian Kulis. Metric learning: A survey. *Foundations and Trends® in Machine Learning*, 5(4):287–364, 2013.
- [Kumar et al., 2016] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *International conference on machine learning*, pages 1378–1387, 2016.
- [Kumar et al., 2021] Sachin Kumar, Eric Malmi, Aliaksei Severyn, and Yulia Tsvetkov. Controlled text generation as continuous optimization with multiple constraints. *Advances in Neural Information Processing Systems*, 34:14542–14554, 2021.
- [Kumar and Byrne, 2004] Shankar Kumar and William Byrne. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*,

- pages 169–176, 2004a.
- [Kumar and Byrne, 2004] Shankar Kumar and William Byrne. Minimum bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 169–176, 2004b.
- [Kung and Peng, 2023] Po-Nien Kung and Nanyun Peng. Do models really learn to follow instructions? an empirical study of instruction tuning. *arXiv preprint arXiv:2305.11383*, 2023.
- [Kupiec, 1992] Julian Kupiec. Robust part-of-speech tagging using a hidden markov model. *Computer Speech & Language*, 6:225–242, 1992.
- [Kwon et al., 2023] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.
- [Lafferty et al., 2001] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning 2001*, pages 282–289, 2001.
- [Lagunas et al., 2021] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10619–10629, 2021.
- [Lai et al., 2015] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.
- [Lake and Baroni, 2018] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.
- [Lambert et al., 2024] Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.
- [Lample and Conneau, 2019] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.
- [Lample et al., 2016] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 260–270, 2016.
- [Lample et al., 2019] Guillaume Lample, Alexandre Sablayrolles, Marc’Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. *Advances in Neural Information Processing Systems*, 32, 2019.
- [Lan et al., 2020] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Landauer et al., 1998] Thomas K Landauer, Peter W Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.
- [Lang et al., 1990] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. A time-delay neural network

- architecture for isolated word recognition. *Neural networks*, 3(1):23–43, 1990.
- [Lapesa and Evert, 2014] Gabriella Lapesa and Stefan Evert. A large scale evaluation of distributional semantic models: Parameters, interactions and model selection. *Transactions of the Association for Computational Linguistics*, 2:531–546, 2014.
- [Lawson, 2003] Mark V. Lawson. *Finite Automata (1st ed.)*. Chapman and Hall/CRC, 2003.
- [Le and Mikolov, 2014] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.
- [Leblond et al., 2021] Rémi Leblond, Jean-Baptiste Alayrac, Laurent Sifre, Miruna Pislari, Lespiau Jean-Baptiste, Ioannis Antonoglou, Karen Simonyan, and Oriol Vinyals. Machine translation decoding beyond beam search. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8410–8434, 2021.
- [LeCun and Bengio, 1995] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- [LeCun et al., 1989] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [LeCun et al., 2012] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.
- [Lee et al., 2023] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.
- [Lee et al., 2017] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378, 2017.
- [Lee et al., 2020] Jason Lee, Elman Mansimov, and Kyunghyun Cho. Deterministic non-autoregressive neural sequence modeling by iterative refinement. In *2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018*, pages 1173–1182, 2020.
- [Lee et al., 2019] John Boaz Lee, Ryan A Rossi, Sungchul Kim, Nesreen K Ahmed, and Eunyee Koh. Attention models in graphs: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(6):1–25, 2019.
- [Lenci, 2018] Alessandro Lenci. Distributional models of word meaning. *Annual review of Linguistics*, 4:151–171, 2018.
- [Lepikhin et al., 2021] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Lester et al., 2021] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, 2021.
- [Leviathan et al., 2023] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of International Conference on Machine*

- Learning*, pages 19274–19286. PMLR, 2023.
- [Levy and Goldberg, 2014] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, 2014a.
- [Levy and Goldberg, 2014] Omer Levy and Yoav Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the eighteenth conference on computational natural language learning*, pages 171–180, 2014b.
- [Levy and Goldberg, 2014] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014c.
- [Levy et al., 2015] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the association for computational linguistics*, 3:211–225, 2015.
- [Lewis et al., 2020] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020a.
- [Lewis et al., 2020] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020b.
- [Li et al., 2024] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391*, 2024a.
- [Li et al., 2020] Bei Li, Hui Liu, Ziyang Wang, Yufan Jiang, Tong Xiao, Jingbo Zhu, Tongran Liu, and Changliang Li. Does multi-encoder help? a case study on context-aware neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3512–3518, 2020a.
- [Li et al., 2020] Bei Li, Ziyang Wang, Hui Liu, Yufan Jiang, Quan Du, Tong Xiao, Huizhen Wang, and Jingbo Zhu. Shallow-to-deep training for neural machine translation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 995–1005, 2020b.
- [Li et al., 2021] Bei Li, Ziyang Wang, Hui Liu, Quan Du, Tong Xiao, Chunliang Zhang, and Jingbo Zhu. Learning light-weight translation models from deep transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13217–13225, 2021a.
- [Li et al., 2022] Bei Li, Quan Du, Tao Zhou, Yi Jing, Shuhan Zhou, Xin Zeng, Tong Xiao, Jingbo Zhu, Xuebo Liu, and Min Zhang. Ode transformer: An ordinary differential equation-inspired model for sequence generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8335–8351, 2022a.
- [Li et al., 2022] Bei Li, Tong Zheng, Yi Jing, Chengbo Jiao, Tong Xiao, and Jingbo Zhu. Learning multiscale transformer models for sequence generation. In *International Conference on Machine Learning*, pages 13225–13241. PMLR, 2022b.
- [Li et al., 2023] Bei Li, Rui Wang, Junliang Guo, Kaitao Song, Xu Tan, Hany Hassan, Arul Menezes, Tong Xiao, Jiang Bian, and JingBo Zhu. Deliberate then generate: Enhanced prompting framework

- for text generation. *arXiv preprint arXiv:2305.19835*, 2023a.
- [Li, 2011] Hang Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Online access: Morgan & Claypool Synthesis Collection Five. Morgan & Claypool Publishers, 2011. ISBN 9781608457076.
- [Li et al., 2022] Hongkang Li, Meng Wang, Sijia Liu, and Pin-Yu Chen. A theoretical understanding of shallow vision transformers: Learning, generalization, and sample complexity. In *The Eleventh International Conference on Learning Representations*, 2022c.
- [Li et al., 2022] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110*, 2022d.
- [Li et al., 2021] Jicheng Li, Pengzhi Gao, Xuanfu Wu, Yang Feng, Zhongjun He, Hua Wu, and Haifeng Wang. Mixup decoding for diverse machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 312–320, 2021b.
- [Li et al., 2020] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, 2020c.
- [Li and Jurafsky, 2016] Jiwei Li and Dan Jurafsky. Mutual information and diverse decoding improve neural machine translation. *arXiv preprint arXiv:1601.00372*, 2016.
- [Li et al., 2016] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and William B Dolan. A diversity-promoting objective function for neural conversation models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 110–119, 2016.
- [Li et al., 2017] Jiwei Li, Will Monroe, and Dan Jurafsky. Learning to decode for future success. *arXiv preprint arXiv:1701.06549*, 2017a.
- [Li et al., 2017] Junhui Li, Deyi Xiong, Zhaopeng Tu, Muhua Zhu, Min Zhang, and Guodong Zhou. Modeling source syntax for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 688–697, 2017b.
- [Li et al., 2021] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. Align before fuse: Vision and language representation learning with momentum distillation. *Advances in neural information processing systems*, 34:9694–9705, 2021c.
- [Li et al., 2022] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning*, pages 12888–12900. PMLR, 2022e.
- [Li et al., 2024] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. In *The Twelfth International Conference on Learning Representations*, 2024b.
- [Li et al., 2023] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023b.
- [Li and Liang, 2021] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational*

- Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.
- [Li et al., 2019] Xintong Li, Guanlin Li, Lemaoy Liu, Max Meng, and Shuming Shi. On the word alignment from neural machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1293–1303, 2019.
- [Li et al., 2022] Yanghao Li, Chao-Yuan Wu, Haoqi Fan, Karttikeya Mangalam, Bo Xiong, Jitendra Malik, and Christoph Feichtenhofer. Mvitv2: Improved multiscale vision transformers for classification and detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4804–4814, 2022f.
- [Li et al., 2018] Yanyang Li, Tong Xiao, Yinqiao Li, Qiang Wang, Changming Xu, and Jingbo Zhu. A simple and effective approach to coverage-aware neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 292–297, 2018.
- [Li, 2023] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, pages 641–647, 2023.
- [Li et al., 2023] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353, 2023c.
- [Li et al., 2021] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021d.
- [Lialin et al., 2023] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023.
- [Liao et al., 2021] Kaiyuan Liao, Yi Zhang, Xuancheng Ren, Qi Su, Xu Sun, and Bin He. A global past-future early exit method for accelerating inference of pre-trained language models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2013–2023, 2021.
- [Lightman et al., 2024] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Likhomanenko et al., 2021] Tatiana Likhomanenko, Qiantong Xu, Gabriel Synnaeve, Ronan Collobert, and Alex Rogozhnikov. Cape: Encoding relative positions with continuous augmented positional embeddings. *Advances in Neural Information Processing Systems*, 34:16079–16092, 2021.
- [Lin et al., 2022] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 2022a.
- [Lin et al., 2022] Ye Lin, Shuhan Zhou, Yanyang Li, Anxiang Ma, Tong Xiao, and Jingbo Zhu. Multi-path transformer is better: A case study on neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5646–5656, 2022b.
- [Lin et al., 2017] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

- [Ling et al., 2015] Wang Ling, Chris Dyer, Alan W Black, Isabel Trancoso, Ramón Fernández, Silvio Amir, Luis Marujo, and Tiago Luís. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1520–1530, 2015.
- [Linzen, 2016] Tal Linzen. Issues in evaluating semantic spaces using word analogies. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 13–18, 2016.
- [Lippmann, 1989] Richard P Lippmann. Review of neural networks for speech recognition. *Neural computation*, 1(1):1–38, 1989.
- [Lipton et al., 2015] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.
- [Liu et al., 2024] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.
- [Liu et al., 2020] Fenglin Liu, Xuancheng Ren, Zhiyuan Zhang, Xu Sun, and Yuexian Zou. Rethinking skip connection with layer normalization. In *Proceedings of the 28th international conference on computational linguistics*, pages 3586–3598, 2020a.
- [Liu et al., 2023] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023a.
- [Liu and Motoda, 2012] Huan Liu and Hiroshi Motoda. *Feature selection for knowledge discovery and data mining*, volume 454. Springer Science & Business Media, 2012.
- [Liu et al., 2022] Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, 2022.
- [Liu et al., 2016] Lemao Liu, Masao Utiyama, Andrew Finch, and Eiichiro Sumita. Agreement on target-bidirectional neural machine translation. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 411–416, 2016a.
- [Liu et al., 2016] Lemao Liu, Masao Utiyama, Andrew Finch, and Eiichiro Sumita. Neural machine translation with supervised attention. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3093–3102, 2016b.
- [Liu et al., 2020] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, 2020b.
- [Liu et al., 2020] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5747–5763, November 2020c.
- [Liu et al., 2023] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023b.
- [Liu et al., 2018] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. In *Proceedings*

- of International Conference on Learning Representations*, 2018.
- [Liu et al., 2017] Shusen Liu, Peer-Timo Bremer, Jayaraman J Thiagarajan, Vivek Srikumar, Bei Wang, Yarden Livnat, and Valerio Pascucci. Visual exploration of semantic relationships in neural word embeddings. *IEEE transactions on visualization and computer graphics*, 24(1):553–562, 2017.
- [Liu et al., 2024] Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*, 2024b.
- [Liu, 2009] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.
- [Liu et al., 2023] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023c.
- [Liu et al., 2023] Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. *arXiv preprint arXiv:2311.12785*, 2023d.
- [Liu et al., 2024] Xinyu Liu, Runsong Zhao, Pengcheng Huang, Chunyang Xiao, Bei Li, Jingang Wang, Tong Xiao, and Jingbo Zhu. Forgetting curve: A reliable method for evaluating memorization capability for long-context models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4667–4682, 2024c.
- [Liu et al., 2023] Yang Liu, Yao Zhang, Yixin Wang, Feng Hou, Jin Yuan, Jiang Tian, Yang Zhang, Zhongchao Shi, Jianping Fan, and Zhiqiang He. A survey of visual transformers. *IEEE Transactions on Neural Networks and Learning Systems*, 2023e.
- [Liu et al., 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [Liu et al., 2020] Yuchen Liu, Junnan Zhu, Jiajun Zhang, and Chengqing Zong. Bridging the modality gap for speech-to-text translation. *arXiv preprint arXiv:2010.14920*, 2020d.
- [Longpre et al., 2023] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pages 22631–22648. PMLR, 2023.
- [Lopez, 2008] Adam Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3): 1–49, 2008.
- [Lu et al., 2016] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. Hierarchical question-image co-attention for visual question answering. *Advances in neural information processing systems*, 29, 2016.
- [Lund, 1995] Kevin Lund. Semantic and associative priming in high-dimensional semantic space. In *Proc. of the 17th Annual conferences of the Cognitive Science Society, 1995*, 1995.
- [Lund and Burgess, 1996] Kevin Lund and Curt Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior research methods, instruments, & computers*, 28(2):203–208, 1996.
- [Luong et al., 2015] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical*

- Methods in Natural Language Processing*, pages 1412–1421, 2015.
- [Ma et al., 2019] Mingbo Ma, Liang Huang, Hao Xiong, Renjie Zheng, Kaibo Liu, Baigong Zheng, Chuanqiang Zhang, Zhongjun He, Hairong Liu, Xing Li, Hua Wu, and Haifeng Wang. Stacl: Simultaneous translation with implicit anticipation and controllable latency using prefix-to-prefix framework. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3025–3036, 2019.
- [Ma and Hovy, 2016] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1064–1074, 2016.
- [Ma et al., 2023] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. Mega: Moving average equipped gated attention. In *The Eleventh International Conference on Learning Representations*, 2023.
- [Ma et al., 2024] Xuezhe Ma, Xiaomeng Yang, Wenhan Xiong, Beidi Chen, Lili Yu, Hao Zhang, Jonathan May, Luke Zettlemoyer, Omer Levy, and Chunting Zhou. Megalodon: Efficient llm pretraining and inference with unlimited context length. *arXiv preprint arXiv:2404.08801*, 2024.
- [Madaan et al., 2024] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Malaviya et al., 2018] Chaitanya Malaviya, Pedro Ferreira, and André FT Martins. Sparse and constrained attention for neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 370–376, 2018.
- [Manning and Schütze, 1999] Chris Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
- [Manning, 2022] Christopher D Manning. Human language understanding & reasoning. *Daedalus*, 151(2):127–138, 2022.
- [Manning et al., 2008] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [Manning et al., 2020] Christopher D Manning, Kevin Clark, John Hewitt, Urvashi Khandelwal, and Omer Levy. Emergent linguistic structure in artificial neural networks trained by self-supervision. *Proceedings of the National Academy of Sciences*, 117(48):30046–30054, 2020.
- [Marcus, 1993] Gary F Marcus. Negative evidence in language acquisition. *Cognition*, 46(1):53–85, 1993.
- [Markman, 2013] Arthur B Markman. *Knowledge representation*. Psychology Press, 2013.
- [Markov, 1913] AA Markov. Essai d’une recherche statistique sur le texte du roman. *Eugene Onegin*” illustrant la liaison des épreuve en chain (‘Example of a statistical investigation of the text of “Eugene Onegin” illustrating the dependence between samples in chain”). In: *Izvestia Imperatorskoi Akademii Nauk (Bulletin de l’Académie Impériale des Sciences de St.-Pétersbourg)*. 6th ser, 7:153–162, 1913.
- [Martins et al., 2022] Pedro Henrique Martins, Zita Marinho, and André FT Martins.  $\infty$ -former: Infinite memory transformer-former: Infinite memory transformer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages

- 5468–5485, 2022.
- [Maruf et al., 2019] Sameen Maruf, André FT Martins, and Gholamreza Haffari. Selective attention for context-aware neural machine translation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3092–3102, 2019.
- [Masoudnia and Ebrahimpour, 2014] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *The Artificial Intelligence Review*, 42(2):275, 2014.
- [Matusov et al., 2006] Evgeny Matusov, Nicola Ueffing, and Hermann Ney. Computing consensus translation for multiple machine translation systems using enhanced hypothesis alignment. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 33–40, 2006.
- [Mavi et al., 2024] Vaibhav Mavi, Anubhav Jangra, and Adam Jatowt. Multi-hop question answering. *Foundations and Trends® in Information Retrieval*, 17(5):457–586, 2024.
- [McCallum et al., 2000] Andrew McCallum, Dayne Freitag, and Fernando C. N. Pereira. Maximum entropy markov models for information extraction and segmentation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 591–598, 2000.
- [McCann et al., 2017] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. *Advances in neural information processing systems*, 30, 2017.
- [McCarley et al., 2019] JS McCarley, Rishav Chakravarti, and Avirup Sil. Structured pruning of a bert-based question answering model. *arXiv preprint arXiv:1910.06360*, 2019.
- [McClave and Sincich, 2006] James T. McClave and Terry Sincich. *Statistics (10th ed.)*. Prentice Hall, 2006.
- [McCulloch and Pitts, 1943] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
- [McElreath, 2020] Richard McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and STAN (2nd ed.)*. Chapman and Hall/CRC, 2020.
- [McNamara, 2005] Timothy P McNamara. *Semantic priming: Perspectives from memory and word recognition*. Psychology Press, 2005.
- [Meister et al., 2020] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8:795–809, 2020.
- [Merrill et al., 2022] William Merrill, Ashish Sabharwal, and Noah A Smith. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022.
- [Meyer and Schvaneveldt, 1971] David E Meyer and Roger W Schvaneveldt. Facilitation in recognizing pairs of words: evidence of a dependence between retrieval operations. *Journal of experimental psychology*, 90(2):227, 1971.
- [Mi et al., 2016] Haitao Mi, Baskaran Sankaran, Zhiguo Wang, and Abe Ittycheriah. Coverage embedding models for neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 955–960, 2016a.
- [Mi et al., 2016] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. Supervised attentions for neural

- machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2283–2288, 2016b.
- [Michel et al., 2019] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.
- [Micikevicius et al., 2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *Proceedings of International Conference on Learning Representations*, 2018.
- [Mielke et al., 2021] Sabrina J. Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp. *arXiv preprint arXiv:2112.10508*, 2021.
- [Miettinen, 1999] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 1999.
- [Mikolov et al., 2010] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of Interspeech*, pages 1045–1048, 2010.
- [Mikolov et al., 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, 2013a.
- [Mikolov et al., 2013] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013b.
- [Mikolov et al., 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 3111–3119, 2013c.
- [Mikolov et al., 2013] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013d.
- [Miller et al., 2016] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1400–1409, 2016.
- [Min et al., 2019] Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. Multi-hop reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109, 2019.
- [Minaee et al., 2024] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.
- [Minsky and Papert, 1969] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT press, 1969.
- [Mishra et al., 2022] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-

- task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, 2022.
- [Mitchell and Lapata, 2010] Jeff Mitchell and Mirella Lapata. Composition in distributional models of semantics. *Cognitive science*, 34(8):1388–1429, 2010.
- [Mitchell, 1997] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.
- [Mnih and Kavukcuoglu, 2013] Andriy Mnih and Koray Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. *Advances in neural information processing systems*, 26, 2013.
- [Mnih et al., 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pages 1928–1937, 2016.
- [Mohri et al., 2018] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning (2nd ed.)*. MIT Press, 2018.
- [Mohtashami and Jaggi, 2024] Amirkeivan Mohtashami and Martin Jaggi. Random-access infinite context length for transformers. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Montague, 1974] Richard Montague. Universal grammar. In R. Thomason, editor, *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.
- [Mu et al., 2024] Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Müller and Sennrich, 2021] Mathias Müller and Rico Sennrich. Understanding the properties of minimum bayes risk decoding in neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 259–272, 2021.
- [Munkhdalai et al., 2024] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.
- [Murphy, 2012] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
- [Murray and Chiang, 2018] Kenton Murray and David Chiang. Correcting length bias in neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 212–223, 2018.
- [Nagel et al., 2021] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.
- [Nair and Hinton, 2009] Vinod Nair and Geoffrey E Hinton. 3d object recognition with deep belief nets. *Advances in neural information processing systems*, 22, 2009.
- [Nakano et al., 2021] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

- [Narayanan et al., 2021] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.
- [Neelakantan et al., 2015] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.
- [Neisser, 2014] Ulric Neisser. *Cognitive Psychology: Classic Edition*. Psychology Press, 2014.
- [Nesterov, 1983] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate  $o(1/k^2)$ . In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.
- [Ng et al., 1999] Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.
- [Nguyen et al., 2020] Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. Tree-structured attention with hierarchical accumulation. In *Proceedings of the 8th International Conference on Learning Representations ICLR*, 2020.
- [Nvidia, 2025] Nvidia. Nvidia nim llms benchmarking. <https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html>, 2025. Retrieved 2025-03-17.
- [Och, 2003] Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st annual meeting of the Association for Computational Linguistics*, pages 160–167, 2003.
- [Och and Ney, 2002] Franz Josef Och and Hermann Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 295–302, 2002.
- [Och and Ney, 2003] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.
- [Olive, 2022] David Olive. Robust statistics, 2022. URL <http://parker.ad.siu.edu/Olive/ol-bookp.htm>.
- [Olshausen and Field, 1997] Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.
- [Oord et al., 2017] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.
- [Oord et al., 2018] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.
- [OpenAI, 2024] OpenAI. Learning to reason with llms, September 2024. URL <https://openai.com/index/learning-to-reason-with-llms/>.
- [Opitz and Maclin, 1999] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.
- [Oppenheim and Schaffer, 1975] Alan V Oppenheim and Ronald W Schaffer. Digital signal processing(book). *Prentice-Hall*, 1975.

- [Orvieto et al., 2023] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. *arXiv preprint arXiv:2303.06349*, 2023.
- [Osgood, 1952] Charles E Osgood. The nature and measurement of meaning. *Psychological bulletin*, 49(3):197, 1952.
- [Ott et al., 2018] Myle Ott, Michael Auli, David Grangier, and Marc’Aurelio Ranzato. Analyzing uncertainty in neural machine translation. In *International Conference on Machine Learning*, pages 3956–3965. PMLR, 2018a.
- [Ott et al., 2018] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9, October 2018b.
- [Ott et al., 2019] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, 2019.
- [Ouyang et al., 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- [Padó and Lapata, 2007] Sebastian Padó and Mirella Lapata. Dependency-based construction of semantic space models. *Computational Linguistics*, 33(2):161–199, 2007.
- [Pal et al., 2023] Koyena Pal, Jiuding Sun, Andrew Yuan, Byron C Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, pages 548–560, 2023.
- [Pan et al., 2022] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022.
- [Pan et al., 2024] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024.
- [Pang et al., 2002] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*, pages 79–86, 2002.
- [Papineni et al., 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- [Parisi et al., 2022] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- [Parisi et al., 2019] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71,

- 2019.
- [Park et al., 2019] Wonpyo Park, Dongju Kim, Yan Lu, and Minsu Cho. Relational knowledge distillation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3967–3976, 2019.
- [Parmar et al., 2018] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pages 4055–4064. PMLR, 2018.
- [Pascanu et al., 2013] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.
- [Patel et al., 2024] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.
- [Pearson, 1901] Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.
- [Penedo et al., 2023] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.
- [Peng et al., 2019] Baoyun Peng, Xiao Jin, Jiaheng Liu, Dongsheng Li, Yichao Wu, Yu Liu, Shunfeng Zhou, and Zhaoning Zhang. Correlation congruence for knowledge distillation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5007–5016, 2019.
- [Peng et al., 2023] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Gv, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan S. Wind, Stanislaw Wozniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.
- [Peng et al., 2024] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Peng et al., 2021] H Peng, N Pappas, D Yogatama, R Schwartz, N Smith, and L Kong. Random feature attention. In *Proceedings of International Conference on Learning Representations (ICLR 2021)*, 2021.
- [Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.
- [Pérez et al., 2018] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. In *Proceedings of International Conference on Learning Representations*, 2018.

- [Perozzi et al., 2014] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.
- [Peters et al., 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.
- [Petroni et al., 2019] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. Language models as knowledge bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2463–2473, 2019.
- [Pham et al., 2019] Ngoc-Quan Pham, Thai-Son Nguyen, Jan Niehues, Markus Müller, Sebastian Stüker, and Alexander Waibel. Very deep self-attention networks for end-to-end speech recognition. *arXiv preprint arXiv:1904.13377*, 2019.
- [Picone, 1993] Joseph W Picone. Signal modeling techniques in speech recognition. *Proceedings of the IEEE*, 81(9):1215–1247, 1993.
- [Pires et al., 2023] Telmo Pessoa Pires, António V Lopes, Yannick Assogba, and Hendra Setiawan. One wide feedforward is all you need. *arXiv preprint arXiv:2309.01826*, 2023.
- [Plackett, 1975] Robin L Plackett. The analysis of permutations. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 24(2):193–202, 1975.
- [Plaut et al., 1986] David C Plaut, Steven J Nowlan, and Geoffrey E Hinton. Experiments on learning by back propagation. Technical report, Carnegie-Mellon University, 1986.
- [Polyak, 1964] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.
- [Pope et al., 2023] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023.
- [Porter, 1980] Martin F Porter. An algorithm for suffix stripping. *Program*, 1980.
- [Post and Vilar, 2018] Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1314–1324, 2018.
- [Prasad et al., 2023] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3845–3864, 2023.
- [Prechelt, 1998] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
- [Press et al., 2021] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Press et al., 2022] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with

- linear biases enables input length extrapolation. In *Proceedings of International Conference on Learning Representations*, 2022.
- [Press et al., 2023] Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, 2023.
- [Provilkov et al., 2020] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. Bpe-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1882–1892, 2020.
- [Pryzant et al., 2023] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.
- [Qiu et al., 2020] Jiezhong Qiu, Hao Ma, Omer Levy, Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2555–2565, 2020a.
- [Qiu et al., 2020] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020b.
- [Rabiner and Juang, 1993] Lawrence Rabiner and Bing-Hwang Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc., 1993.
- [Rabiner and Gold, 1975] Lawrence R Rabiner and Bernard Gold. Theory and application of digital signal processing. *Prentice-Hall*, 1975.
- [Radford et al., 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.
- [Radford et al., 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 2019.
- [Radford et al., 2021] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.
- [Rae et al., 2019] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *Proceedings of International Conference on Learning Representations*, 2019a.
- [Rae et al., 2019] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2019b.
- [Rafailov et al., 2024] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Raffel et al., 2017] Colin Raffel, Minh-Thang Luong, Peter J Liu, Ron J Weiss, and Douglas Eck. Online and linear-time attention by enforcing monotonic alignments. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2837–2846, 2017.
- [Raffel et al., 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang,

- Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [Ramachandran et al., 2017] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [Ramshaw and Marcus, 1995] Lance Ramshaw and Mitch Marcus. Text chunking using transformation-based learning. In *Third Workshop on Very Large Corpora*, 1995.
- [Reddy, 1976] D Raj Reddy. Speech recognition by machine: A review. *Proceedings of the IEEE*, 64(4):501–531, 1976.
- [Reimers and Gurevych, 2019] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019.
- [Reisinger and Mooney, 2010] Joseph Reisinger and Raymond Mooney. Multi-prototype vector-space models of word meaning. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 109–117, 2010.
- [Ren et al., 2017] Zhou Ren, Xiaoyu Wang, Ning Zhang, Xutao Lv, and Li-Jia Li. Deep reinforcement learning-based image captioning with embedding reward. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 290–298, 2017.
- [Rifai et al., 2011] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of International Conference on Machine Learning*, 2011.
- [Rogers et al., 2018] Anna Rogers, Shashwath Hosur Ananthakrishna, and Anna Rumshisky. What’s in your embedding, and how it predicts task performance. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2690–2703, 2018.
- [Rolnick et al., 2019] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.
- [Romero et al., 2014] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.
- [Rosenblatt, 1957] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.
- [Rosenfeld et al., 2020] Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Ross, 1924] William David Ross. *Aristotle’s metaphysics*. Clarendon Press, 1924.
- [Rosti et al., 2007] Antti-Veikko Rosti, Spyros Matsoukas, and Richard Schwartz. Improved word-level system combination for machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 312–319, 2007.
- [Roy et al., 2021] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

- [Ruan et al., 2024] Junhao Ruan, Long Meng, Weiqiao Shan, Tong Xiao, and Jingbo Zhu. A survey of llm surveys. <https://github.com/NiuTrans/ABigSurveyOfLLMs>, 2024.
- [Rubenstein and Goodenough, 1965] Herbert Rubenstein and John B Goodenough. Contextual correlates of synonymy. *Communications of the ACM*, 8(10):627–633, 1965.
- [Rubin et al., 2022] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, 2022.
- [Ruder, 2017] Sebastian Ruder. Deep learning for nlp best practices. <https://runder.io/deep-learning-nlp-best-practices/index.html>, 2017.
- [Rumelhart et al., 1986] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [Rush and Collins, 2012] Alexander M Rush and MJ Collins. A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research*, 45:305–362, 2012.
- [Rush et al., 2015] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389, 2015.
- [Russell, 2019] Stuart Russell. *Human Compatible: Artificial Intelligence and the Problem of Controls*. Viking, 2019.
- [Russell and Norvig, 2010] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd ed.)*. Prentice Hall, 2010.
- [Sanh et al., 2020] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.
- [Sanh et al., 2022] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *Proceedings of International Conference on Learning Representations*, 2022.
- [Sankaran et al., 2016] Baskaran Sankaran, Haitao Mi, Yaser Al-Onaizan, and Abe Ittycheriah. Temporal attention model for neural machine translation. *arXiv preprint arXiv:1608.02927*, 2016.
- [Santacroce et al., 2023] Michael Santacroce, Zixin Wen, Yelong Shen, and Yuanzhi Li. What matters in the structured pruning of generative language models? *arXiv preprint arXiv:2302.03773*, 2023.
- [Santos and Gatti, 2014] Cícero dos Santos and Maíra Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.
- [Schacter and Buckner, 1998] Daniel L Schacter and Randy L Buckner. Priming and the brain. *Neuron*, 20(2):185–195, 1998.
- [Schapire, 1990] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):

197–227, 1990.

- [Schick et al., 2023] Timo Schick, Jane A. Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Schick et al., 2024] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Schlag et al., 2021] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *Proceedings of International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.
- [Schmidhuber, 2015] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [Schnabel et al., 2015] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015a.
- [Schnabel et al., 2015] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015b.
- [Schneider et al., 2019] Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised pre-training for speech recognition. In *INTERSPEECH*, 2019.
- [Schulman et al., 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pages 1889–1897, 2015.
- [Schulman et al., 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [Schuster and Nakajima, 2012] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, pages 5149–5152, 2012.
- [Schuster et al., 2022] Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.
- [Schwartz et al., 2020] Roy Schwartz, Gabriel Stanovsky, Swabha Swayamdipta, Jesse Dodge, and Noah A Smith. The right tool for the job: Matching model and instance complexities. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6640–6651, 2020.
- [See, 2018] Abigail See. Deep learning, structure and innate priors: A discussion between yann lecun and christopher manning, 02 2018. URL <http://www.abigailsee.com/2018/02/21/deep-learning-structure-and-innate-priors.html>.
- [See et al., 2017] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for*

- Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, 2017.
- [Seni et al., 2010] Giovanni Seni, John Elder, and Robert Grossman. *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Morgan and Claypool Publishers, 2010.
- [Sennrich et al., 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, 2016a.
- [Sennrich et al., 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, 2016b.
- [Seo et al., 2017] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *Proceedings of International Conference on Learning Representations*, 2017.
- [Shannon, 1948] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948a.
- [Shannon, 1948] Claude E. Shannon. A mathematical theory of communication. Report, Bell Labs, 1948b.
- [Shannon, 1951] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.
- [Shaw et al., 2018] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.
- [Shazeer, 2019] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.
- [Shazeer, 2020] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.
- [Shazeer et al., 2017] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarsz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proceedings of International Conference on Learning Representations*, 2017.
- [Shen et al., 2020] Dinghan Shen, Mingzhi Zheng, Yelong Shen, Yanru Qu, and Weizhu Chen. A simple but tough-to-beat data augmentation approach for natural language understanding and generation. *arXiv preprint arXiv:2009.13818*, 2020a.
- [Shen et al., 2020] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020b.
- [Shen et al., 2016] Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Minimum risk training for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1683–1692, 2016.

- [Shen et al., 2019] Tianxiao Shen, Myle Ott, Michael Auli, and Marc’ Aurelio Ranzato. Mixture models for diverse machine translation: Tricks of the trade. In *International conference on machine learning*, pages 5719–5728, 2019.
- [Shi et al., 2016] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural mt learn source syntax? In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 1526–1534, 2016.
- [Shinn et al., 2023] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
- [Shoeybi et al., 2019] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [Shorten and Khoshgoftaar, 2019] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.
- [Silver et al., 2017] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.
- [Singhal, 2005] Amit Singhal. Introducing the knowledge graph: things, not strings, 2005.
- [Skalse et al., 2022] Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.
- [Skorski et al., 2021] Maciej Skorski, Alessandro Temperoni, and Martin Theobald. Revisiting weight initialization of deep neural networks. In *Asian Conference on Machine Learning*, pages 1192–1207. PMLR, 2021.
- [Smith et al., 2017] Samuel L Smith, David HP Turban, Steven Hamblin, and Nils Y Hammerla. Offline bilingual word vectors, orthogonal transformations and the inverted softmax. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.
- [Smith et al., 2018] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don’t decay the learning rate, increase the batch size. In *Proceedings of the 6th International Conference on Learning Representations ICLR*, 2018.
- [Snell et al., 2022] Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. *arXiv preprint arXiv:2209.15189*, 2022.
- [Snell et al., 2024] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- [Snell et al., 2025] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [So et al., 2019] David So, Quoc Le, and Chen Liang. The evolved transformer. In *Proceedings of International conference on machine learning*, pages 5877–5886. PMLR, 2019.
- [Socher et al., 2011] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural

- scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.
- [Socher et al., 2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.
- [Søgaard, 2016] Anders Søgaard. Evaluating word embeddings with fmri and eye-tracking. In *Proceedings of the 1st workshop on evaluating vector-space representations for NLP*, pages 116–121, 2016.
- [Solorio-Fernández et al., 2020] Saúl Solorio-Fernández, J Ariel Carrasco-Ochoa, and José Fco Martínez-Trinidad. A review of unsupervised feature selection methods. *Artificial Intelligence Review*, 53(2):907–948, 2020.
- [Song et al., 2019] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936. PMLR, 2019.
- [Sperber et al., 2018] Matthias Sperber, Jan Niehues, Graham Neubig, Sebastian Stüker, and Alex Waibel. Self-attentional acoustic models. In *Proceedings of Interspeech 2018*, pages 3723–3727, 2018.
- [Srivastava et al., 2014] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [Srivastava et al., 2015] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.
- [Stahlberg and Byrne, 2019] Felix Stahlberg and Bill Byrne. On nmt search errors and model errors: Cat got your tongue? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3356–3362, 2019.
- [Stahlberg et al., 2016] Felix Stahlberg, Eva Hasler, Aurelien Waite, and Bill Byrne. Syntactically guided neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 299–305, 2016.
- [Sternberg, 1996] Robert J Sternberg. *Cognitive psychology*. Harcourt Brace College Publishers, 1996.
- [Stewart, 1993] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.
- [Stiennon et al., 2020] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.
- [Stock et al., 2021] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme model compression. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Strubell et al., 2018] Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. Linguistically-informed self-attention for semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5027–5038, 2018.

- [Su et al., 2021] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.
- [Su et al., 2024] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
- [Su et al., 2022] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. *Advances in Neural Information Processing Systems*, 35:21548–21561, 2022.
- [Sukhbaatar et al., 2015] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. *Advances in neural information processing systems*, 28, 2015.
- [Sukhbaatar et al., 2019] Sainbayar Sukhbaatar, Édouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 331–335, 2019.
- [Sun et al., 2023] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.
- [Sun et al., 2020] Zewei Sun, Shujian Huang, Hao-Ran Wei, Xin-yu Dai, and Jiajun Chen. Generating diverse translation by manipulating multi-head attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8976–8983, 2020a.
- [Sun et al., 2020] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020b.
- [Sundermeyer et al., 2012] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Proceedings of the Thirteenth annual conference of the international speech communication association*, 2012.
- [Sutskever, 2013] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, 2013.
- [Sutskever et al., 2013] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.
- [Sutskever et al., 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- [Sutton and McCallum, 2012] Charles Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.
- [Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.
- [Szabó, 2020] Zoltán Gendler Szabó. Compositionality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2020 edition, 2020.
- [Szegedy et al., 2014] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations*, 2014a.

- [Szegedy et al., 2014] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of 2nd International Conference on Learning Representations (ICLR 2014)*, 2014b.
- [Szegedy et al., 2016] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- [Szepesvári, 2010] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.
- [Tai et al., 2015] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, 2015.
- [Talmor and Berant, 2018] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. *arXiv preprint arXiv:1803.06643*, 2018.
- [Tan and Le, 2019] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
- [Tang et al., 2015] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.
- [Tank and Hopfield, 1987] David W Tank and JJ Hopfield. Neural computation by concentrating information in time. *Proceedings of the National Academy of Sciences*, 84(7):1896–1900, 1987.
- [Taori et al., 2023] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca), 2023.
- [Taskar et al., 2005] Ben Taskar, Simon Lacoste-Julien, and Dan Klein. A discriminative matching approach to word alignment. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 73–80, 2005.
- [Tay et al., 2020] Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *Proceedings of International Conference on Machine Learning*, pages 9438–9447. PMLR, 2020a.
- [Tay et al., 2020] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020b.
- [Team et al., 2024] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.
- [Teknium, 2023] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL <https://huggingface.co/datasets/teknium/OpenHermes-2.5>.
- [Telgarsky, 2016] Matus Telgarsky. Benefits of depth in neural networks. In *Conference on learning theory*, pages 1517–1539. PMLR, 2016.
- [Tenney et al., 2019] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical

- nlp pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601, 2019a.
- [Tenney et al., 2019] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R Thomas McCoy, Najoung Kim, Benjamin Van Durme, Sam Bowman, Dipanjan Das, and Ellie Pavlick. What do you learn from context? probing for sentence structure in contextualized word representations. In *Proceedings of International Conference on Learning Representations*, 2019b.
- [Timonin et al., 2022] Denis Timonin, BoYang Hsueh, and Vinh Nguyen. Accelerated inference for large transformer models using nvidia triton inference server. <https://developer.nvidia.com/blog/accelerated-inference-for-large-transformer-models-using-nvidia-fastertranz> 2022.
- [Tissier et al., 2017] Julien Tissier, Christophe Gravier, and Amaury Habrard. Dict2vec: Learning word embeddings using lexical dictionaries. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 254–263, 2017.
- [Tjong Kim Sang, 2002] Erik F. Tjong Kim Sang. Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *Proceedings of COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*, 2002.
- [Tjong Kim Sang and Buchholz, 2000] Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task chunking. In *Proceedings of Fourth Conference on Computational Natural Language Learning and the Second Learning Language in Logic Workshop*, 2000.
- [Touvron et al., 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- [Touvron et al., 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- [Trentin and Gori, 2001] Edmondo Trentin and Marco Gori. A survey of hybrid ann/hmm models for automatic speech recognition. *Neurocomputing*, 37(1-4):91–126, 2001.
- [Tsvetkov et al., 2015] Yulia Tsvetkov, Manaal Faruqui, Wang Ling, Guillaume Lample, and Chris Dyer. Evaluation of word vector representations by subspace alignment. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2049–2054, 2015.
- [Tu et al., 2016] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. Modeling coverage for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for*

- Computational Linguistics (Volume 1: Long Papers)*, pages 76–85, 2016.
- [Tulving and Schacter, 1990] Endel Tulving and Daniel L. Schacter. Priming and human memory systems. *Science*, 247(4940):301–306, 1990.
- [Uesato et al., 2022] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.
- [Uffink, 2017] Jos Uffink. Boltzmann’s Work in Statistical Physics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2017 edition, 2017.
- [Ulyanov et al., 2016] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.
- [Van der Maaten and Hinton, 2008] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
- [Vapnik and Chervonenkis, 1971] Vladimir Vapnik and Alexey Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–279, 1971.
- [Vaswani et al., 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems*, volume 30, 2017.
- [Veličković et al., 2018] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.
- [Vijayakumar et al., 2018] Ashwin Vijayakumar, Michael Cogswell, Ramprasaath Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search for improved description of complex scenes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [Vincent et al., 2010] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.
- [Vinyals et al., 2015] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *Advances in neural information processing systems*, 28, 2015.
- [Viterbi, 1967] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 1967.
- [Vogel et al., 1996] Stephan Vogel, Hermann Ney, and Christoph Tillmann. Hmm-based word alignment in statistical translation. In *COLING 1996 Volume 2: The 16th International Conference on Computational Linguistics*, 1996.
- [Voita et al., 2018] Elena Voita, Pavel Serdyukov, Rico Sennrich, and Ivan Titov. Context-aware neural machine translation learns anaphora resolution. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1264–1274, 2018.
- [Voita et al., 2019] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages

- 5797–5808, 2019.
- [Von Oswald et al., 2023] Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *Proceedings of International Conference on Machine Learning*, pages 35151–35174. PMLR, 2023.
- [Waibel et al., 1989] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.
- [Wallace et al., 2019] Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do nlp models know numbers? probing numeracy in embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5307–5315, 2019.
- [Wang et al., 2024] Chenglong Wang, Hang Zhou, Yimin Hu, Yifu Huo, Bei Li, Tongran Liu, Tong Xiao, and Jingbo Zhu. Esrl: Efficient sampling-based reinforcement learning for sequence generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19107–19115, 2024.
- [Wang et al., 2020] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. Hat: Hardware-aware transformers for efficient natural language processing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7675–7688, 2020a.
- [Wang et al., 2022] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. Deepnet: Scaling transformers to 1,000 layers. *arXiv preprint arXiv:2203.00555*, 2022a.
- [Wang et al., 2022] Hongyu Wang, Shuming Ma, Shaohan Huang, Li Dong, Wenhui Wang, Zhiliang Peng, Yu Wu, Payal Bajaj, Saksham Singhal, Alon Benhaim, Barun Patra, Zhun Liu, Vishrav Chaudhary, Xia Song, and Furu Wei. Foundation transformers. *arXiv preprint arXiv:2210.06423*, 2022b.
- [Wang et al., 2022] Jue Wang, Ke Chen, Gang Chen, Lidan Shou, and Julian McAuley. Skipbert: Efficient inference with shallow layer skipping. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7287–7301, 2022c.
- [Wang and Yoon, 2021] Lin Wang and Kuk-Jin Yoon. Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks. *IEEE transactions on pattern analysis and machine intelligence*, 44(6):3048–3068, 2021.
- [Wang et al., 2023] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023a.
- [Wang et al., 2023] Peihao Wang, Rameswar Panda, Lucas Torroba Hennigen, Philip Greengard, Leonid Karlinsky, Rogerio Feris, David Daniel Cox, Zhangyang Wang, and Yoon Kim. Learning to grow pretrained models for efficient transformer training. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023b.
- [Wang et al., 2018] Qiang Wang, Fuxue Li, Tong Xiao, Yanyang Li, Yinqiao Li, and Jingbo Zhu. Multi-layer representation fusion for neural machine translation. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3015–3026, 2018a.
- [Wang et al., 2019] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th*

- Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, 2019a.
- [Wang et al., 2020] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020b.
- [Wang et al., 2019] Wei Wang, Vincent Wenchen Zheng, Han Yu, and Chunyan Miao. A survey of zero-shot learning. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10:1–37, 2019b.
- [Wang et al., 2018] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018b.
- [Wang et al., 2022] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747*, 2022d.
- [Wang et al., 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023c.
- [Wang et al., 2020] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys*, 53(3):1–34, 2020c.
- [Wang et al., 2022] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sapat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, 2022e.
- [Wang et al., 2023] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources. *Advances in Neural Information Processing Systems*, 36:74764–74786, 2023d.
- [Wang et al., 2023] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, 2023e.
- [Wang et al., 2023] Zhenyi Wang, Enneng Yang, Li Shen, and Heng Huang. A comprehensive survey of forgetting in deep learning beyond continual learning. *arXiv preprint arXiv:2307.09218*, 2023f.
- [Wang et al., 2020] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6151–6162, 2020d.
- [Warstadt et al., 2019] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

- [Webster and Kit, 1992] Jonathan J Webster and Chunyu Kit. Tokenization as the initial phase in nlp. In *Proceedings of COLING 1992 volume 4: The 14th international conference on computational linguistics*, 1992.
- [Wei et al., 2022] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *Proceedings of International Conference on Learning Representations*, 2022a.
- [Wei et al., 2022] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022b.
- [Wei et al., 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022c.
- [Weiss et al., 2021] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *Proceedings of International Conference on Machine Learning*, pages 11080–11090. PMLR, 2021.
- [Welleck et al., 2023] Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Weng, 2021] Lilian Weng. How to train really large models on many gpus? *lilianweng.github.io*, Sep 2021. URL <https://lilianweng.github.io/posts/2021-09-25-train-large/>.
- [Werbos, 1990] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [Weston et al., 2015] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*, 2015.
- [Wiener, 1960] Norbert Wiener. Some moral and technical consequences of automation: As machines learn they may develop unforeseen strategies at rates that baffle their programmers. *Science*, 131(3410):1355–1358, 1960.
- [Wiggs and Martin, 1998] Cheri L Wiggs and Alex Martin. Properties and mechanisms of perceptual priming. *Current opinion in neurobiology*, 8(2):227–233, 1998.
- [Wiher et al., 2022] Gian Wiher, Clara Meister, and Ryan Cotterell. On decoding strategies for neural text generators. *Transactions of the Association for Computational Linguistics*, 10:997–1012, 2022.
- [Williams et al., 2018] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, 2018.
- [Williams, 1992] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.
- [Williams and Peng, 1990] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.
- [Williams and Zipser, 1989] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

- [Wingate et al., 2022] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. Prompt compression and contrastive conditioning for controllability and toxicity reduction in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5621–5634, 2022.
- [Wittgenstein, 1953] Ludwig Wittgenstein. *Philosophical investigations. Philosophische Untersuchungen*. Macmillan, 1953.
- [Wold et al., 1987] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.
- [Wolpert, 1996] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computatoin*, 8(7):1341–1390, 1996.
- [Wolpert and Macready, 1997] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
- [Wozengraft and Reiffen, 1961] John M. Wozengraft and Barney Reiffen. *Sequential Decoding*. The MIT Press, 1961.
- [Wright and Ma, 2022] John Wright and Yi Ma. *High-Dimensional Data Analysis with Low-Dimensional Models: Principles, Computation, and Applications*. Cambridge University Press, 2022.
- [Wu et al., 2023] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023a.
- [Wu et al., 2018] Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *Proceedings of International Conference on Learning Representations*, 2018a.
- [Wu et al., 2019] Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *Proceedings of International Conference on Learning Representations*, 2019.
- [Wu et al., 2024] Wilson Wu, John X Morris, and Lionel Levine. Do language models plan for future tokens? *arXiv preprint arXiv:2404.00859*, 2024.
- [Wu et al., 2020] Xuanfu Wu, Yang Feng, and Chenze Shao. Generating diverse translation from model distribution with dropout. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1088–1097, 2020a.
- [Wu et al., 2016] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [Wu et al., 2021] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Wu and He, 2018] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

- [Wu et al., 2023] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b.
- [Wu et al., 2020] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020b.
- [Wu et al., 2018] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8817–8826, 2018b.
- [Xia et al., 2024] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.
- [Xiao et al., 2024] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *Proceedings of The Twelfth International Conference on Learning Representations*, 2024.
- [Xiao et al., 2013] Tong Xiao, Jingbo Zhu, and Tongran Liu. Bagging and boosting statistical machine translation systems. *Artificial Intelligence*, 195:496–527, 2013.
- [Xiao et al., 2019] Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. Sharing attention weights for fast transformer. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*, pages 5292–5298, 2019.
- [Xie et al., 2017] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.
- [Xie et al., 2022] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In *Proceedings of International Conference on Learning Representations*, 2022.
- [Xin et al., 2020] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, 2020.
- [Xiong et al., 2020] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tiejun Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pages 10524–10533, 2020.
- [Xu et al., 2024] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Xu and Mcauley, 2023] Canwen Xu and Julian Mcauley. A survey on dynamic neural networks for natural language processing. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 2325–2336, 2023.
- [Xu et al., 2021] Chen Xu, Bojie Hu, Yanyang Li, Yuhao Zhang, Shen Huang, Qi Ju, Tong Xiao, and

- Jingbo Zhu. Stacked acoustic-and-textual encoding: Integrating the pre-trained models into speech translation encoders. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2619–2630, 2021a.
- [Xu et al., 2023] Chen Xu, Rong Ye, Qianqian Dong, Chengqi Zhao, Tom Ko, Mingxuan Wang, Tong Xiao, and Jingbo Zhu. Recent advances in direct speech-to-text translation. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-23): Survey Track*, pages 6796–6804, 2023a.
- [Xu et al., 2023] Chen Xu, Yuhao Zhang, Chengbo Jiao, Xiaoqian Liu, Chi Hu, Xin Zeng, Tong Xiao, Anxiang Ma, Huizhen Wang, and Jingbo Zhu. Bridging the granularity gap for acoustic modeling. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10816–10833, 2023b.
- [Xu et al., 2020] Hongfei Xu, Qihui Liu, Josef van Genabith, Deyi Xiong, and Jingyi Zhang. Lipschitz constrained parameter initialization for deep transformers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 397–402, July 2020.
- [Xu et al., 2015] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.
- [Xu et al., 2023] Peng Xu, Xiatian Zhu, and David A Clifton. Multimodal learning with transformers: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023c.
- [Xu et al., 2021] Zenan Xu, Daya Guo, Duyu Tang, Qinliang Su, Linjun Shou, Ming Gong, Wanjun Zhong, Xiaojun Quan, Daxin Jiang, and Nan Duan. Syntax-enhanced pre-trained model. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5412–5422, 2021b.
- [Yang et al., 2024] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [Yang et al., 2018] Baosong Yang, Zhaopeng Tu, Derek F Wong, Fandong Meng, Lidia S Chao, and Tong Zhang. Modeling localness for self-attention networks. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4449–4458, 2018a.
- [Yang et al., 2018] Yilin Yang, Liang Huang, and Mingbo Ma. Breaking the beam search curse: A study of (re-) scoring methods and stopping criteria for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3054–3059, 2018b.
- [Yang et al., 2023] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 2023a.
- [Yang et al., 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.
- [Yang et al., 2023] Zi Yang, Samridhi Choudhary, Siegfried Kunzmann, and Zheng Zhang. Quantization-aware and tensor-compressed training of transformers for natural language understanding. *arXiv*

- preprint arXiv:2306.01076*, 2023b.
- [Yang et al., 2016] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.
- [Yao et al., 2024] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Yao et al., 2007] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26:289–315, 2007.
- [Yarowsky, 1994] David Yarowsky. Decision lists for lexical ambiguity resolution: Application to accent restoration in Spanish and French. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 88–95, 1994.
- [Yarowsky, 1995] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.
- [Ye et al., 2021] Rong Ye, Mingxuan Wang, and Lei Li. End-to-end speech translation via cross-modal progressive training. *arXiv preprint arXiv:2104.10380*, 2021.
- [Yin et al., 2023] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. A survey on multimodal large language models. *arXiv preprint arXiv:2306.13549*, 2023.
- [You et al., 2020] Weiqiu You, Simeng Sun, and Mohit Iyyer. Hard-coded gaussian attention for neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7689–7700, 2020.
- [Yu et al., 2022] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [Yu et al., 2023] Yaodong Yu, Sam Buchanan, Druv Pai, Tianzhe Chu, Ziyang Wu, Shengbang Tong, Benjamin D Haeffele, and Yi Ma. White-box transformers via sparse rate reduction. *arXiv preprint arXiv:2306.01129*, 2023a.
- [Yu et al., 2023] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. Towards better chain-of-thought prompting strategies: A survey. *arXiv preprint arXiv:2310.04959*, 2023b.
- [Yuksel et al., 2012] Seniha Esen Yuksel, Joseph N Wilson, and Paul D Gader. Twenty years of mixture of experts. *IEEE transactions on neural networks and learning systems*, 23(8):1177–1193, 2012.
- [Yun et al., 2019] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *Proceedings of International Conference on Learning Representations*, 2019.
- [Zaheer et al., 2020] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, C. Alberti, S. Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, L. Yang, and A. Ahmed. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33: 17283–17297, 2020.
- [Zaslavskiy et al., 2009] Mikhail Zaslavskiy, Marc Dymetman, and Nicola Cancedda. Phrase-based

- statistical machine translation as a traveling salesman problem. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 333–341, 2009.
- [Zeiler, 2012] Matthew D Zeiler. Adadelata: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [Zellers et al., 2018] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 93–104, 2018.
- [Zhang et al., 2021] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.
- [Zhang and Sennrich, 2019] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.
- [Zhang et al., 2018] Biao Zhang, Deyi Xiong, and Jinsong Su. Accelerating neural transformer via an average attention network. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1789–1798, 2018a.
- [Zhang et al., 2019] Biao Zhang, Ivan Titov, and Rico Sennrich. Improving deep transformer with depth-scaled initialization and merged attention. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 898–909, 2019a.
- [Zhang et al., 2020] Jiajun Zhang, Long Zhou, Yang Zhao, and Chengqing Zong. Synchronous bidirectional inference for neural sequence generation. *Artificial Intelligence*, 281:103234, 2020a.
- [Zhang et al., 2019] Juexiao Zhang, Yubei Chen, Brian Cheung, and Bruno A Olshausen. Word embedding visualization via dictionary learning. *arXiv preprint arXiv:1910.03833*, 2019b.
- [Zhang et al., 2020] Wei Emma Zhang, Quan Z Sheng, Ahoud Alhazmi, and Chenliang Li. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(3):1–41, 2020b.
- [Zhang et al., 2015] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.
- [Zhang et al., 2018] Xiangwen Zhang, Jinsong Su, Yue Qin, Yang Liu, Rongrong Ji, and Hongji Wang. Asynchronous bidirectional decoding for neural machine translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018b.
- [Zhang and Yang, 2021] Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021.
- [Zhang et al., 2024] Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. In *ACL (Findings)*, 2024.
- [Zhang et al., 2020] Zhuosheng Zhang, Yuwei Wu, Junru Zhou, Sufeng Duan, Hai Zhao, and Rui Wang. Sg-net: Syntax-guided machine reading comprehension. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 9636–9643, 2020c.
- [Zhang et al., 2023] Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and Hai Zhao. Igniting language intelligence: The hitchhiker’s guide from chain-of-thought reasoning to language agents. *arXiv*

- preprint arXiv:2311.11797*, 2023a.
- [Zhang et al., 2023] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*, 2023b.
- [Zhao et al., 2006] Hai Zhao, Chang-Ning Huang, Mu Li, and Bao-Liang Lu. Effective tag set selection in Chinese word segmentation via conditional random field modeling. In *Proceedings of the 20th Pacific Asia Conference on Language, Information and Computation*, pages 87–94, 2006.
- [Zhao et al., 2024] Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. *arXiv preprint arXiv:2402.04833*, 2024.
- [Zhao et al., 2023] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie, and Ji rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.
- [Zheng et al., 2019] Baigong Zheng, Renjie Zheng, Mingbo Ma, and Liang Huang. Simpler and faster learning of adaptive policies for simultaneous translation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1349–1354, 2019.
- [Zheng et al., 2018] Zaixiang Zheng, Hao Zhou, Shujian Huang, Lili Mou, Xinyu Dai, Jiajun Chen, and Zhaopeng Tu. Modeling past and future for neural machine translation. *Transactions of the Association for Computational Linguistics*, 6:145–157, 2018.
- [Zhong et al., 2024] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.
- [Zhou et al., 2023] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinu Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023a.
- [Zhou et al., 2023] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023b.
- [Zhou et al., 2021] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11106–11115, 2021.
- [Zhou et al., 2017] Long Zhou, Wenpeng Hu, Jiajun Zhang, and Chengqing Zong. Neural system combination for machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 378–384, 2017.
- [Zhou et al., 2020] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33:18330–18341, 2020.

- [Zhou et al., 2023] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023c.
- [Zhou, 2012] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall/CRC, 2012a.
- [Zhou, 2012] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2012b.
- [Zoph and Le, 2016] Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *Proceedings of International Conference on Learning Representations*, 2016.
- [Zoph et al., 2020] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33:3833–3845, 2020.

# Index

- $k$ -NN, 445
- $k$ -NN LM, 447
- $k$ -NN language modeling, 447
- $k$ -nearest neighbors, 42, 445
- $l_2$  regularization, 100
- $l_p$  norm, 21
- $n$ -gram language modeling, 80
- $p$ -norm, 21
- $p$ -norm distance, 22
- $p$ -value, 58
- (Higher-order) Runge-Kutta Methods, 306
- 0-1 Loss, 44
- 0-1 Masking, 286
  
- long-term memory, 182
  
- A\* search, 262
- A2C, 560
- absolute positional encoding, 197, 273
- action-value function, 555
- AdaDelta, 93
- AdaGrad, 92
- Adam, 93
- Adaptive Gradient Descent, 92
- Adaptive Moment Estimation, 93
- add- $\alpha$  smoothing, 36
- Additive Attention, 221
- additive smoothing, 36
- addressing, 237
- advantage, 559
- advantage actor-critic, 560
- adversarial machine learning, 107
- adversarial samples, 107
- affine transformation, 72
- Agent, 416
  
- AGI, 361
- AIC, 53
- ALiBi, 456
- alignment, 415
- alignment link, 66
- alignment scores, 220
- alternative hypothesis, 58
- AR processes, 180
- artificial general intelligence, 361
- Artificial neural networks, 71
- Associativity, 20
- attacks, 107
- attention field, 313
- attention head, 229
- attention weight, 219
- attention with linear biases, 456
- auto-encoders, 108
- auto-encoding, 46, 62
- auto-regressive, 61
- automated machine learning, 336, 515
- automatic prompt design, 515
- AutoML, 336, 515
- autonomous agents, 513
- autoregressive processes, 180
- averaging pooling, 87
  
- back translation, 263
- back-propagation through time, 177
- back-translation, 107
- background task, 153
- backward pass, 78
- BART, 385
- batch, 95
- batch gradient descent, 95
- batching, 95

- Beam search, 247
- beam size, 247
- beam width, 247
- Bernoulli naive Bayes, 33
- BERT, 365
- Best-of- $N$  sampling, 583
- BGD, 95
- bi-directional models, 180
- BIC, 53
- bilinear transform, 333
- binary classification, 27
- binary variable, 22
- Boltzmann distribution, 104
- BoN sampling, 583
- BPE, 131
- BPTT, 177
- Bradley-Terry model, 562
- branches, 288
- Byte Pair Encoding, 131
  
- CAE, 112
- calculation annotation, 488
- canonicalization, 125
- capacity, 182
- catastrophic forgetting, 401
- categories, 26
- causal language modeling, 374
- chain of thought, 489
- chain rule, 23
- chain rule of differentiation, 77
- chain-of-thought prompting, 422
- checkpoint ensembling, 261
- chunking, 61
- Classification, 26
- classification model, 27
- classification system, 26
- classifier, 26
- Cloze, 294
- CNN, 85
- co-adaptation, 101
- co-attention, 237
- code, 109
- coding, 182
- column vector, 19
- combinatorial optimization problems, 254
- Commutativity, 20
- completion, 370
- component models, 49
- component systems, 260
- compositional generalization, 499
- compositional models, 209
- Compositionality, 140
- Computation graphs, 75
- conditional computation, 337
- Conditional Probability, 23
- conditional random fields, 130, 204
- connectionist temporal classification, 202
- Constant Initialization, 96
- constituent systems, 260
- constrained optimization, 258
- constrained search, 266
- context window, 143
- Contextuality, 141
- Continuous batching, 611
- continuous memory, 321
- continuous optimization, 256
- contraction mapping, 113
- contractive auto-encoder, 112
- contrastive learning, 45
- Contrastive Loss, 45
- convolution, 189
- convolution kernels, 85
- convolution operation, 188
- convolution product, 189
- convolutional layer, 85
- Convolutional neural networks, 85
- correlation coefficients, 144
- corrupted input, 114
- corruption, 114
- cosine, 196
- Cosine Attention, 220
- cost functions, 43
- CoT, 489
- COT prompting, 422

- coverage, 240
- coverage vector, 241
- CRFs, 130, 204
- cross-entropy, 25
- cross-lingual language models, 395
- cross-validation, 56
- CTC, 202
- cumulative reward, 556
- curve fitting, 43
  
- DAEs, 113
- data augmentation, 107, 263, 342
- decay factor, 93
- decision boundary, 31
- decision surface, 31
- decoder, 109
- decoder-only architecture, 273
- Decoding, 212, 593
- decoding, 203
- decoding blocks, 272
- decoding layers, 272
- decoding system, 214
- deduction, 39
- deep learning, 71
- deep neural network, 41
- deep neural networks, 71
- deliberate-then-generate, 504
- delta rule, 91
- demonstrations, 371
- denoising, 113
- Denoising auto-encoders, 113
- dependent variable, 49
- depth, 74
- depth growth, 303
- depth-first search, 255
- diagonal state-space models, 335
- diagonalization, 335
- dilated context window, 315
- direct preference optimization, 575
- discriminant function, 30
- discriminative models, 32, 33
- Distance-based Loss, 43
- distant reward, 38
- distributed representation, 139
- distributed representations, 80
- distribution, 24
- distributional hypothesis, 142
- distributional representation, 142
- distributional semantics, 142, 169
- distributional word representation, 142
- Distributivity, 20
- Divergence-based Loss, 43
- Document Rotation, 385
- dot product, 20
- Dot-product Attention, 220
- DPO, 575
- dropout, 101
- dropout rate, 101
- DTG, 504
- duration, 182
- Dynamic Neural Networks, 337
  
- early exit classifier, 340
- early stop, 48
- early stopping, 94, 337
- early-stop, 249
- edge probing, 293
- effective number of parameters, 53
- embedding matrix, 82
- emergent abilities, 292, 433
- emission probability, 204
- Emission-like Features, 205
- encoder, 109, 171
- encoder-decoder, 63
- encoder-decoder architecture, 212
- encoder-decoder attention, 225
- encoder-only architecture, 273
- encoding blocks, 269
- encoding layers, 269
- encoding system, 214
- end-to-end memory networks, 183
- ensemble learning, 55
- ensembling, 260
- error gradient, 177

- Error-based Loss, 46
- error-driven learning, 34
- error-propagation, 78
- Euclidean norm, 22
- Euler method, 306
- event extraction, 67
- evidence lower bound (ELBO), 117
- expectation, 24
- expected embedding, 257
- expected representation, 221
- expected value, 24
- exploding and vanishing gradient problems, 177
- external memories, 183, 445
- Extrapolation, 453
- Extrinsic Evaluation, 163
  
- factor analysis, 153
- fastText, 162
- feature, 28
- feature learning, 108
- feature map, 189
- Feature mapping, 40
- Feature selection, 152
- feature sub-spaces, 226
- feature vector, 72
- feed-forward neural network based language model, 80
- feed-forward neural networks, 74
- few-shot COT prompting, 423
- few-shot learning, 57
- FFNNLM, 80
- FFNNs, 74
- filters, 85
- Fisher's linear discriminant, 152
- Fixed Learning Rates, 97
- forget gate, 184
- forward pass, 76
- fractional count, 136
- frames, 200
- Frobenius norm, 113
- fully connected, 73
  
- gate, 87
- gated linear unit, 428
- gated recurrent units, 185
- gaussian error linear unit, 428
- Gaussian mixture models, 209
- Gaussian naive Bayes, 33
- Gaussian noise, 106
- GeLU, 428
- generalization, 47
- generalization error, 50
- generative models, 32
- Global vectors, 157
- GloVe, 157
- GLU, 429
- GMMs, 209
- GPT, 365
- GQA, 451
- gradient descent, 91
- Gradient Descent with Momentum, 92
- graphical models, 203
- greedy strategy, 246
- Grouped query attention, 451
- GRUs, 185
  
- hard prompts, 519
- head, 229
- heads, 275
- held-out data, 48
- hidden layer, 82
- hidden Markov model, 130
- hidden Markov models, 203
- hinge loss, 45
- histogram pruning, 248
- HMM, 130
- HMMs, 203
- homonymy, 141
- human preference alignment, 533
- hypothesis selection, 261
  
- ICA, 153
- ICL, 422
- ICT, 371
- identity matrix, 18

- IDF, 146
- image captioning, 213
- image-to-text generation, 213
- importance sampling, 564
- impulse noise, 114
- in-context learning, 371, 422, 467
- independent component analysis, 153
- independent variable, 49
- indicator function, 34
- induction, 39
- inductive bias, 39
- inductive inference, 38
- inductive reasoning, 38
- Inference Engine, 611
- inference-time scaling, 621
- information extraction, 67
- information gain, 134
- Initialization with Predefined Distributions, 96
- input gate, 184
- input inversion, 546
- instruction alignment, 533
- instruction fine-tuning, 411, 536
- interactive machine translation, 252
- interference, 396
- internal memories, 445
- Interpolation, 453
- Intrinsic Evaluation, 163
- inverse document frequency, 146
- irreducible error, 435
- iteration-based scheduling, 611
  
- Jacobian matrix, 112
- Joint Probability, 23
  
- kernel fusion, 347
- kernel methods, 40, 324
- key-value cache, 438, 591
- keyword extraction, 67
- Knowledge distillation, 341
- Kullback-Leibler (KL) divergence, 25
- KV cache, 438, 591
  
- label mapping, 479
- label smoothing, 105
- labeled samples, 26
- labels, 26
- language modeling, 61
- large-margin training, 44
- Lasso regularization, 100
- latent Dirichlet allocation, 152
- latent semantic analysis, 147
- latent semantic indexing, 147
- layer, 72
- layer dropout, 305
- Layer-sensitive Initialization, 96
- LDA, 152
- Learning from Human Feedback, 415
- learning rate, 91
- Learning Rate Decay, 97
- learning rate scheduling, 97
- least-to-most prompting, 495
- LeCun initialization, 96
- left-singular vectors, 148
- lemma, 125
- lemmatization, 125
- length normalization, 240
- length reward, 239
- lexical semantics, 140
- linear attention model, 326
- Linear classifier, 29
- linear discriminant analysis, 152
- linear discriminant function, 30
- Linear Multi-step Methods, 306
- linear transformation, 72
- linear-chain CRF, 205
- linearized trees, 210
- linearly separable, 40
- linguistic regularity, 165
- Lipschitz constant, 302
- local attention, 234
- local truncation error, 306
- log-linear, 33
- logistic regression, 33
- Long short-term memory, 184
- long-context LLMs, 436

- loss functions, 43
- low-rank approximation, 149
- LSA, 147
- LSI, 147
- LSTM, 184
  
- MAE, 44
- MAP, 263
- margin, 44
- Margin-based Loss, 44
- Marginal Probability, 23
- Markov assumption, 130
- masked language model, 294
- masked language modeling, 366, 374
- masking noise, 114
- matrix, 18
- matrix addition, 19
- Matrix product, 21
- matrix-matrix product, 21
- max pooling, 86
- maximum a posteriori, 263
- maximum likelihood estimation, 32
- maximum norm, 22
- mBERT, 394
- MBR, 264
- MDL, 53
- mean, 24
- mean absolute error, 44
- mean square error, 44
- memory, 182
- memory cell, 184
- memory-based methods, 445
- MERT, 47
- metric learning, 65
- mini-batch gradient descent, 95
- minibatch, 282
- minimum Bayes risk, 264
- minimum description length, 53
- minimum error-rate training, 47
- minimum-spanning tree, 292
- mining, 49
- mixture model, 55
- mixture-of-experts, 309
- MLE, 32
- MML, 53
- mode, 264
- model averaging, 55
- model capacity, 52
- model complexity, 52
- model depth, 74
- model errors, 50
- Model evaluation, 50
- model function, 336
- model growth, 303
- Model selection, 50
- model width, 74
- MoE, 309
- momentum, 92
- monotonicity, 254
- morphological analysis, 124
- moving average, 332
- MQA, 451
- MSE, 44
- multi-branch neural networks, 229
- Multi-class classification, 27
- Multi-head attention, 226
- multi-head attention, 275
- Multi-label classification, 28
- multi-layer attention, 232
- multi-layer neural network, 73
- multi-lingual BERT, 394
- multi-query attention, 328, 451
- multinomial naive Bayes, 33
- multiple linear regression model, 180
- multiplicative attention, 220
- multivariate regression, 43
  
- named entity recognition, 61
- NAS, 336, 515
- NER, 61
- Neural Architecture Search, 336
- neural architecture search, 515
- neural language models, 80
- neural machine translation, 215

- neural nets, 71
- neural networks, 71
- neural Turing machines, 183
- neurons, 71
- next sentence prediction, 377
- NMT, 215
- non-autoregressive decoding, 266
- non-autoregressive generation, 266
- Non-linear activation functions, 41
- non-parametric, 58
- Non-parametric methods, 41
- norm, 21
- normalization, 88, 125
- NSP, 377
- nucleus sampling, 601
- null hypothesis, 58
  
- objective function, 42
- Occam's Razor, 42
- ODEs, 305
- offline reinforcement learning, 578
- one-hot, 138
- one-hot representations, 80
- one-shot COT prompting, 423
- online sequence-to-sequence system, 252
- OOV, 35
- open-vocabulary, 35
- optimization, 42
- ordinary differential equations, 305
- ORMs, 629
- orthogonal vectors, 148
- out-of-vocabulary, 35
- outcome reward models, 629
- Outcome-based Approaches, 580
- output equation, 332
- over-translation, 240
- overfitting, 39
- overoptimization problem, 573
  
- padding, 85
- pairwise method, 45
- parallel scaling, 627
- parameter sharing, 85
- parametric methods, 41
- parametric test, 58
- parse tree, 63
- part-of-speech tagging, 60
- patches, 354
- paths, 288
- PCA, 108, 149
- perceptrons, 71
- Performance Estimation, 516
- performance function, 557
- performance gap recovered, 551
- permuted language modeling, 375
- PGR, 551
- Plackett-Luce model, 569
- PMI, 144
- pointwise mutual information, 134, 144
- polysemy, 141
- pooling layer, 85
- POS tagging, 60
- positional encoding, 194
- post-norm, 270
- post-training quantization, 346
- PPO, 419, 566
- pre-norm, 277
- pre-training, 38
- Prefilling, 593
- prefix fine-tuning, 523
- prefix language modeling, 381
- prime word, 167
- priming, 166
- principal component, 151
- principal component analysis, 149
- principal component coefficients, 149
- principal component loadings, 149
- principal components analysis, 108
- PRM, 629
- Probability, 22
- probability density, 23
- probability distribution, 24
- probability function, 22
- probability measure, 22
- probes, 291

- probing classifier, 292
- probing predictor, 292
- problem decomposition, 492
- process reward model, 629
- Process-based Approaches, 580
- product rule, 23
- progressive downsampling, 344
- prompt embeddings, 528
- prompt engineering, 467
- prompt optimization, 515
- Prompt Search Space, 515
- prompting engineering, 419
- proximal policy optimization, 419, 566
  
- Q-value function, 555
- QKV attention, 223
- query-key-value attention, 223
  
- RAG, 448
- random processes, 173
- random variable, 22
- Ranking, 49
- Ranking-based Loss, 45
- ratio function, 564
- receiver, 221
- receptive field, 85, 187
- reconstruction loss, 109
- rectified linear unit, 428
- recurrent cell, 83, 184
- Recurrent neural networks, 83
- recurrent unit, 83
- regression, 43
- regular expressions, 125
- reinforcement learning, 38
- reinforcement learning from human feedback, 416, 534
- rejection sampling, 584
- relation extraction, 67, 482
- relative entropy, 25
- relative positional encoding, 197
- relative positional representation, 297
- ReLU, 428
- remove-one, 136
- reordering problem, 234
- reparameterization trick, 117
- request-level scheduling, 611
- reset gate, 185
- residual connections, 89, 179
- Residual neural networks, 89
- retrieval-augmented approach, 322
- retrieval-augmented generation, 448
- return, 556
- reversible residual networks, 328
- reward gaming, 573
- reward hacking, 573
- Reward Model, 416
- Ridge regularization, 100
- right-singular vectors, 148
- risk, 46
- RLHF, 416, 534
- RMSProp, 93
- RNNs, 83
- RoBERTa, 393
- robust statistics, 105
- routing model, 309
- row vector, 19
- RPR, 297
  
- salt-and-pepper noise, 114
- sample efficient, 57, 547
- samples, 26
- saturating activation functions, 178
- Scalar, 18
- scalar product, 19
- scaled dot-product attention, 220
- scaling laws, 345, 433
- Scheduler, 611
- search errors, 50, 248
- search problem, 50
- self-attention, 226
- self-consistency, 507
- self-instruct, 543
- self-paced reading, 167
- self-supervised learning, 38, 367
- self-training, 367

- semi-orthogonal, 148
- semi-supervised learning, 37
- semi-unitary, 148
- sender, 221
- sensory memory, 182
- sentence embedding, 210
- sentence length prediction, 294
- Sentence Reordering, 385
- sentence-level depth-adaptive model, 338
- seq2seq, 211
- Sequence Encoding Models, 368
- Sequence Generation Models, 368
- Sequence labeling, 60
- sequence labeling, 129
- sequence-to-sequence, 62, 211
- sequential scaling, 627
- SFT, 415, 534
- SGD, 94
- shallow-to-deep training, 303
- shared encoder, 352
- short-term memory, 182
- shortcut connections, 89
- shrinkage estimator, 104
- significance level, 58
- significance tests, 58
- similarity function, 65
- similarity learning, 65
- simplex, 257
- simultaneous translation, 252
- sine, 195
- single-label classification, 28
- single-layer attention, 232
- single-layer neural network, 73
- single-layer perceptrons, 71
- single-round prediction, 538
- singular value decomposition, 147
- singular values, 148
- skip connections, 89, 179
- SMT, 240
- Soft Masking, 287
- soft prompts, 519
- soft word alignment matrix, 66
- source sequence, 212
- source-side sequence, 212
- Span Masking, 385
- span prediction, 67
- sparse attention models, 296
- sparse auto-encoders, 111
- sparse coding, 111
- sparse expert models, 338
- sparsity penalty, 111
- sparsity ratio, 313
- Speculative decoding, 604
- speculative execution, 604
- speech encoder, 352
- speed-accuracy trade-off, 250
- SSMs, 332
- standard deviation, 24
- standardization, 89
- state equation, 332
- state variables, 332
- state-space models, 332
- state-value function, 555
- Statistical language modeling, 61
- statistical machine translation, 240
- statistical parsing, 63
- steepest descent, 91
- stem, 126
- stemming, 126
- step function, 72
- stochastic gradient descent, 94
- stochastic processes, 173
- Stopping criterion, 94
- Strong Ceiling Performance, 550
- Structure prediction, 49
- structure prediction, 27
- structured pruning, 342
- sub-layer dropout, 305
- Sub-problem Generation, 495
- Sub-problem Solving, 495
- sub-space method, 260
- subword, 131
- suffix stripping, 126
- superficial alignment hypothesis, 547

- Supervised dimension reduction, 152
- Supervised Fine-tuning, 415
- supervised fine-tuning, 534
- Supervised learning, 37
- supervised learning, 367
- support vector machines, 41
- Surface Forms of Words and Sentences, 294
- surrogate objective, 565
- SVD, 147
- Syntactic and Semantic Labels, 293
- syntactic hierarchy, 290
- syntactic parser, 63
- syntax tree, 63
- syntax-aware Transformer encoders, 284
- system combination methods, 260
- systematic error, 54
  
- t-distributed stochastic neighbor embedding, 168
- t-SNE, 168
- T5, 380
- tags, 26
- target sequence, 212
- target word, 167
- target-side sequence, 212
- TD, 560
- teacher forcing, 239
- teacher-student training, 341
- template filling, 67
- temporal difference, 560
- term frequency, 145
- term frequency-inverse document frequency, 146
- term-document co-occurrence matrix, 145
- term-term co-occurrence matrix, 143
- test error, 50
- text completion, 483
- text embedding, 210
- text encoder, 352
- text generation, 212
- text transformation, 483
- text-to-image generation, 213
  
- TF, 145
- TF-IDF, 146
- the Akaike information criterion, 53
- the bag-of-words (BOW) model, 28
- the Bayesian approach, 46
- the Bayesian information criterion, 53
- the Bayesian risk, 46
- the CBOW model, 155
- The continuous bag-of-words model, 155
- The continuous skip-gram model, 156
- the curse of dimensionality, 39
- The Expectation Step, 136
- the Expectation-Maximization (EM) algorithm, 135
- the kernel trick, 41
- The learning problem, 27
- The Maximization Step, 136
- the minimum message length, 53
- The modeling problem, 27
- the multi-store model, 182
- The no free lunch theorem, 52
- The prediction problem, 27
- the principle of compositionality, 140
- the skip-gram model, 156
- the Vapnik-Chervonenkis dimension, 52
- the VC dimension, 52
- threshold pruning, 248
- time series, 173
- Token Deletion, 385
- Token Masking, 385
- token pruning, 344
- token-level depth-adaptive model, 338
- Tokenization, 123
- tokens, 123
- Topic models, 152
- training epochs, 94
- training error, 50
- training step, 91
- transcription, 200
- transcription labels, 200
- transcription units, 200
- transfer learning, 341

- Transformer, 269
- Transformer-XL, 319
- transition probability, 204
- Transition-like Features, 205
- translation language modeling, 395
- transpose, 19
- tree linearization, 284
- Trees, 292
- trust regions, 565
- Tustin's method, 333
  
- under-translation, 240
- undercomplete auto-encoder, 110
- underfitting, 47
- undirected graphical models, 205
- unfolded, 83
- uni-directional models, 180
- unigram, 134
- unrolled, 83
- unseen words, 36
- unstructured pruning, 342
- unsupervised bilingual dictionary induction, 37
- unsupervised learning, 37, 367
- update gate, 185
- update rule, 91
- update step, 91
  
- VAEs, 115
- validation data, 48
- value function, 262
- value-based search, 262
- variable, 22
- variance, 24
- variational auto-encoders, 115
- Vector, 18
- vector database, 321
- vector function, 30
- Vision Transformer, 353
- visual question answering, 237, 355
- ViT, 353
- Viterbi decoding, 203
- vLBL, 162
  
- VQA, 237, 355
  
- Warmup and Decay, 97
- Weak Performance, 550
- weak-to-strong generalization, 550
- Weak-to-strong Performance, 550
- weight decay, 100
- weight sharing, 85
- Weighted Dot-product Attention, 220
- width, 308
- Word alignment, 234
- word alignment, 66
- word alignment weight matrix, 66
- word clustering, 37
- word distance, 164
- word embedding, 82, 139
- Word Representation Learning, 123
- word segmentation, 124
- word semantic distance, 164
- word sense, 137
- word sense disambiguation, 142
- word-document co-occurrence matrix, 145
- word-word co-occurrence matrix, 143
- Word2Vec, 155
- WSD, 142
  
- Xavier initialization, 96
- XLMs, 395
  
- zero matrix, 18
- zero padding, 282
- zero-shot COT, 423
- zero-shot learning, 414
- Zipf's Law, 36