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

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

June 12, 2025

Tong Xiao and Jingbo Zhu
June, 2025

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

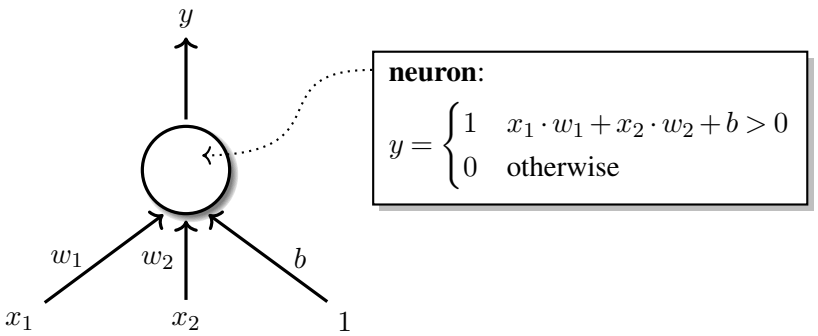


Figure 2.1: A perceptron with two input variables $\{x_1, x_2\}$ and an output variable y. There are two weights $\{w_1, w_2\}$, each corresponding to an input variable. The output depends on the sum of the weighted input variables and the bias term b, say, $y = 1$ if $x_1 \cdot w_1 + x_2 \cdot w_2 +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

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

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

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

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[2].

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

[1] 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 α and β, 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 $f(\mathbf{x}) + \mathbf{b}$.

[2] Since the step function is a linear combination of indicator functions, the perceptron is a linear classifier.

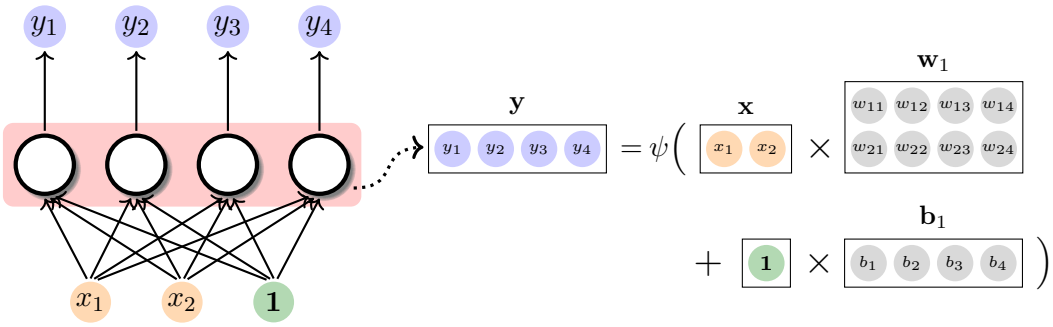


Figure 2.2: A single-layer perceptron involving four neurons. All these neurons receive information from the input variables $\{x_1, x_2\}$. The perceptron describes a process in that 1) we first transform the input vector of variables by an affine transformation $f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w}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

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

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

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:

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

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	$y_i = \max(s_1, ..., s_n)$

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. μ_i and σ_i^2 are the mean and variance respectively.

where $\mathbf{w}_1 \in \mathbb{R}^{3\times4}$, $\mathbf{w}_2 \in \mathbb{R}^{4\times3}$, $\mathbf{w}_3 \in \mathbb{R}^{3\times3}$, 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.

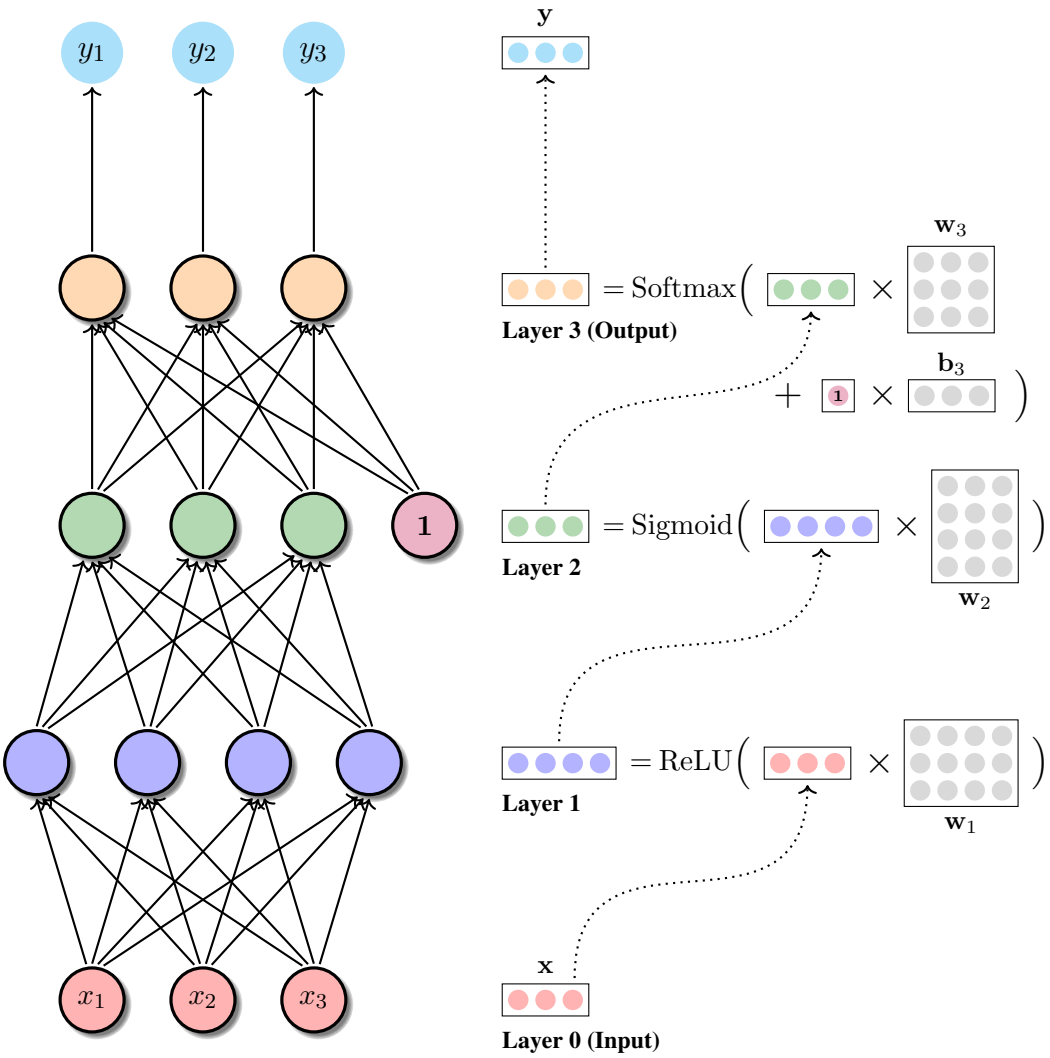


Figure 2.3: A multi-layer neural network. The input layer consists of three variables $\{x_1, x_2, x_3\}$. These variables are fully connected to all neurons of layer 1. The output of layer 1 is a new vector $\mathbf{h}_1 = \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_1)$. It is then fully connected to layer 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[3]. Consider, for example, three

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

functions:

$$\mathbf{y} = \mathbf{x} + \mathbf{w} \tag{2.6}$$
$$\mathbf{y} = \text{Softmax}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}) \tag{2.7}$$
$$\mathbf{y} = \text{Sigmoid}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) - \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_2) \tag{2.8}$$

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:

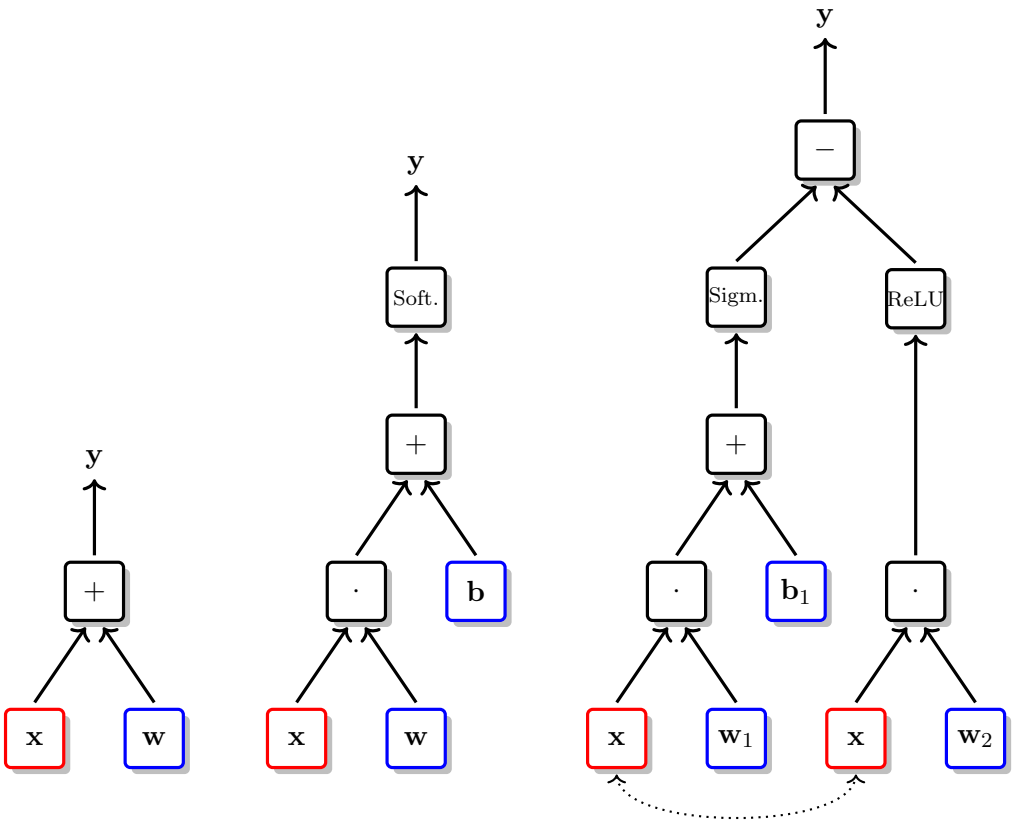
$$\mathbf{y} = \mathbf{h}_1 - \mathbf{h}_2 \tag{2.9}$$
$$\mathbf{h}_1 = \text{Sigmoid}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) \tag{2.10}$$
$$\mathbf{h}_2 = \text{ReLU}(\mathbf{x} \cdot \mathbf{w}_2) \tag{2.11}$$

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:

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



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

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 $\mathbf{x}$ and $\mathbf{w}$ respectively. This node reads the output of the nodes $\mathbf{x}$ and $\mathbf{w}$, and generates the output $\mathbf{y} = \mathbf{x} + \mathbf{w}$. Things are a bit interesting for larger graphs. In sub-graph (b), the output of the dot node (i.e., $\mathbf{x} \cdot \mathbf{w}$) is passed along the edge to the addition node. Then, the addition node computes the sum of $\mathbf{x} \cdot \mathbf{w}$ and $\mathbf{b}$ 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 $\frac{\partial L}{\partial \mathbf{w}_1}$, $\frac{\partial L}{\partial \mathbf{b}_1}$ and $\frac{\partial L}{\partial \mathbf{w}_2}$, 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:

$$\frac{\partial y}{\partial x} = \frac{\partial p}{\partial q} \cdot \frac{\partial q}{\partial x} \tag{2.13}$$

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:

$$\mathbf{y} = \mathbf{h}_1 \cdot \mathbf{w}_2 \tag{2.14}$$
$$\mathbf{h}_1 = \psi(\mathbf{h_2}) \tag{2.15}$$
$$\mathbf{h}_2 = \mathbf{h}_3 + \mathbf{b}_1 \tag{2.16}$$
$$\mathbf{h}_3 = \mathbf{x} \cdot \mathbf{w}_1 \tag{2.17}$$

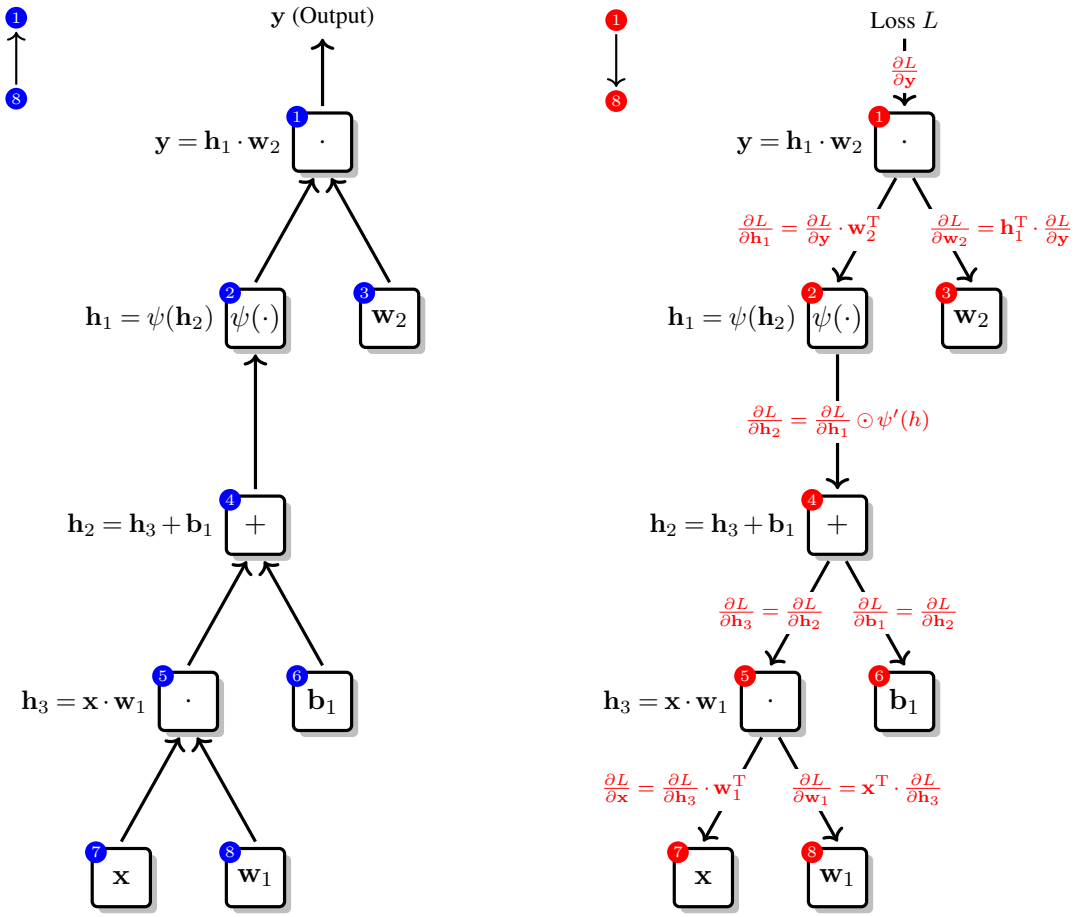
All these variables can be understood in a better way from a computation graph: each variable is a node of the graph, and nodes are connected by algebraic operations and function compositions. Taking Eq. (2.13) and some basic knowledge of calculus, we compute the derivatives of the variables, like these:

$$\text{node 1:} \quad \frac{\partial L}{\partial \mathbf{y}} = \delta_{\mathbf{y}} \tag{2.18}$$
$$\text{node 2:} \quad \frac{\partial L}{\partial \mathbf{h}_1} = \frac{\partial L}{\partial \mathbf{y}} \cdot \mathbf{w}_2^{\mathrm{T}} \tag{2.19}$$
$$\text{node 3:} \quad \frac{\partial L}{\partial \mathbf{w}_2} = \mathbf{h}_1^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{y}} \tag{2.20}$$
$$\text{node 4:} \quad \frac{\partial L}{\partial \mathbf{h}_2} = \frac{\partial L}{\partial \mathbf{h}_1} \odot \psi'(h) \tag{2.21}$$
$$\text{node 5:} \quad \frac{\partial L}{\partial \mathbf{h}_3} = \frac{\partial L}{\partial \mathbf{h}_2} \tag{2.22}$$
$$\text{node 6:} \quad \frac{\partial L}{\partial \mathbf{b}_1} = \frac{\partial L}{\partial \mathbf{h}_2} \tag{2.23}$$
$$\text{node 7:} \quad \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{h}_3} \cdot \mathbf{w}_1^{\mathrm{T}} \tag{2.24}$$
$$\text{node 8:} \quad \frac{\partial L}{\partial \mathbf{w}_1} = \mathbf{x}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{h}_3} \tag{2.25}$$

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 $w_1 ... w_m$, the probability $\Pr(w_1, ..., w_m)$ is



(a) Forward Pass (b) Backward Pass

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:

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

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

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

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

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 $w_{i-n+1}...w_i$, i.e., $\Pr(w_i|w_{i-n+1},...,w_{i-1}) = \frac{\text{count}(w_{i-n+1}...w_i)}{\text{count}(w_{i-n+1}...w_{i-1})}$. 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; 2003]. 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 $w_{i-n+1}...w_{i-1}$.

Figure 2.6 presents the architecture of the **feed-forward neural network based language model** (**FFNNLM**). The input is the context words $w_{i-n+1}...w_{i-1}$. 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 $V = \{$"I","you","he","'she","they"$\}$. The one-hot representation of "you" is

$$w(\text{"you"}) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{2.28}$$

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:

$$\mathbf{e} = \mathbf{x} \cdot \mathbf{C} \tag{2.29}$$

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

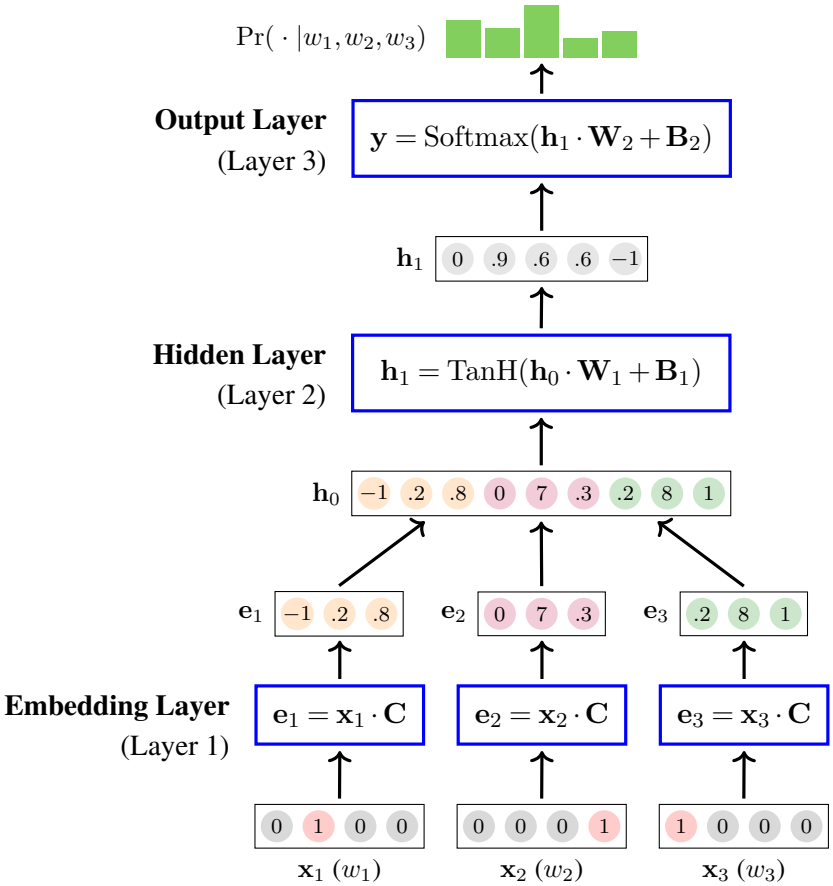


Figure 2.6: A neural language model [Bengio et al., 2003]. Blue boxes represent the layers of the neural network. The input is three context words in their one-hot representations $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$, and the output is the probability distribution of the next word $\Pr(w_4|w_1, w_2, w_3)$. 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 $\mathbf{h}_0$ such that the concatenated vector encodes all input information. Then, $\mathbf{h}_0$ is taken as the input to a normal layer that performs the mapping $\mathbf{h}_1 = \text{TanH}(\mathbf{h}_0 \cdot \mathbf{W}_1 + \mathbf{B}_1)$. The final layer reads $\mathbf{h}_1$ and produces a distribution over the vocabulary, i.e., $\mathbf{y} = \text{Softmax}(\mathbf{h}_1 \cdot \mathbf{W}_2 + \mathbf{B}_2)$ where $y_k = \Pr(V_k|w_1, w_2, w_3)$.

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

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

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

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

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:

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

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

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

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.

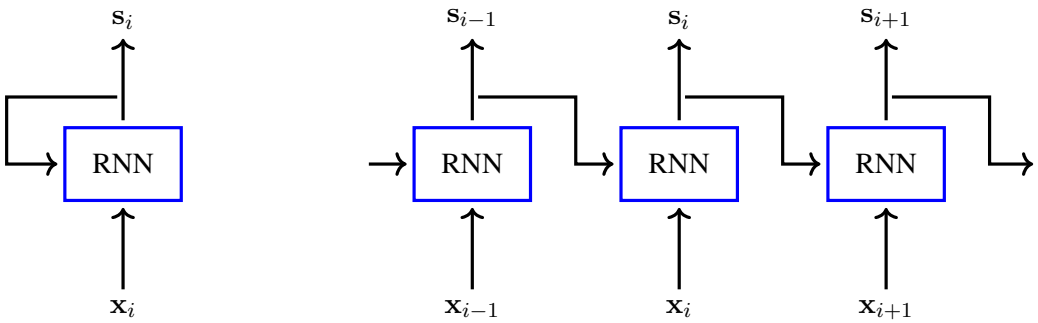
The main idea behind RNNs is to repeatedly utilize a **recurrent unit** (or **recurrent cell**) to compute the output at each position of an input sequence. To be more precise, given a sequence of vectors $\mathbf{x}_1 ... \mathbf{x}_m$, a standard **recurrent unit** can be described as a function $\mathrm{RNN}(\cdot)$ 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:

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

The state $\mathbf{s}_i$ can be viewed as a "memory" that summaries 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 $\mathrm{RNN}(\cdot)$:

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

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



(a) An RNN unit (b) Unrolling the RNN

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 $\mathbf{s}_{i-1}$ 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 $\mathrm{RNN}(\cdot)$. The simplest of these is to formulate $\mathrm{RNN}(\cdot)$ 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 $\mathrm{RNN}(\cdot)$ is given by:

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

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 ψ are $\mathrm{TanH}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, $\mathrm{ReLU}(\cdot)$, and among others. Eq. (2.35) is a single-layer neural network because it has the same form as Eqs. (2.3-2.4):

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

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 $x_1 ... x_m$. The filter ranging from position i to position $i+r-1$ is defined to be:

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

where r is the size of the receptive field, $\mathbf{x}_{[i,i+r-1]}$ is the sub-sequence $x_i ... x_{i+r-1}$, 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:

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

In this way, the convolutional layer transforms the input sequence $x_1 ... x_n$ to the output sequence $v_1 ... v_{\lfloor \frac{m}{stride} \rfloor}$[4]. 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 $stride \times i + r - 1 > m$, the input of the filter is incomplete. A commonly used solution is **padding**. It simply sets the features outside the input sequence

[4] $\lfloor \cdot \rfloor$ stands for the floor function.

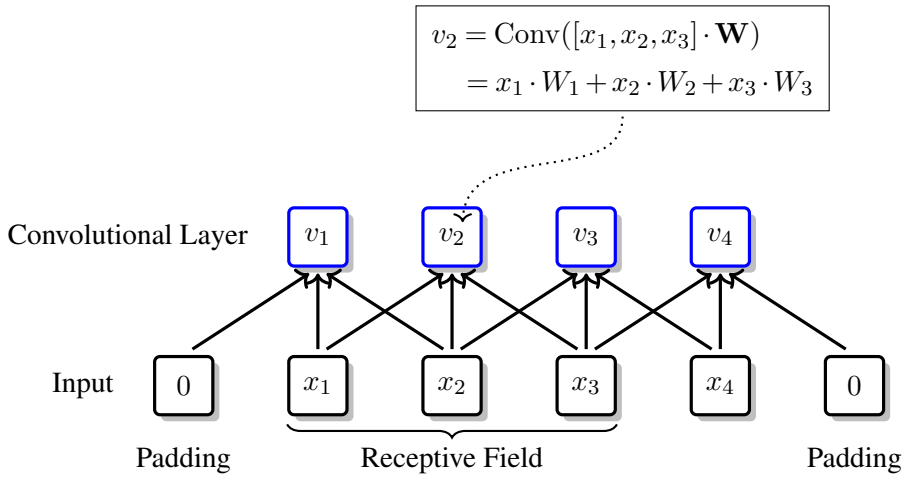


Figure 2.8: Convolution over a sequence of numbers $\{x_1, x_2, x_3, x_4\}$ ($r = 3$ and $stride = 1$). The receptive field defines the region in the input that is taken in computing the output. Here the receptive field has a size of 3, that is, the convolutional operation covers three consecutive numbers 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

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

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

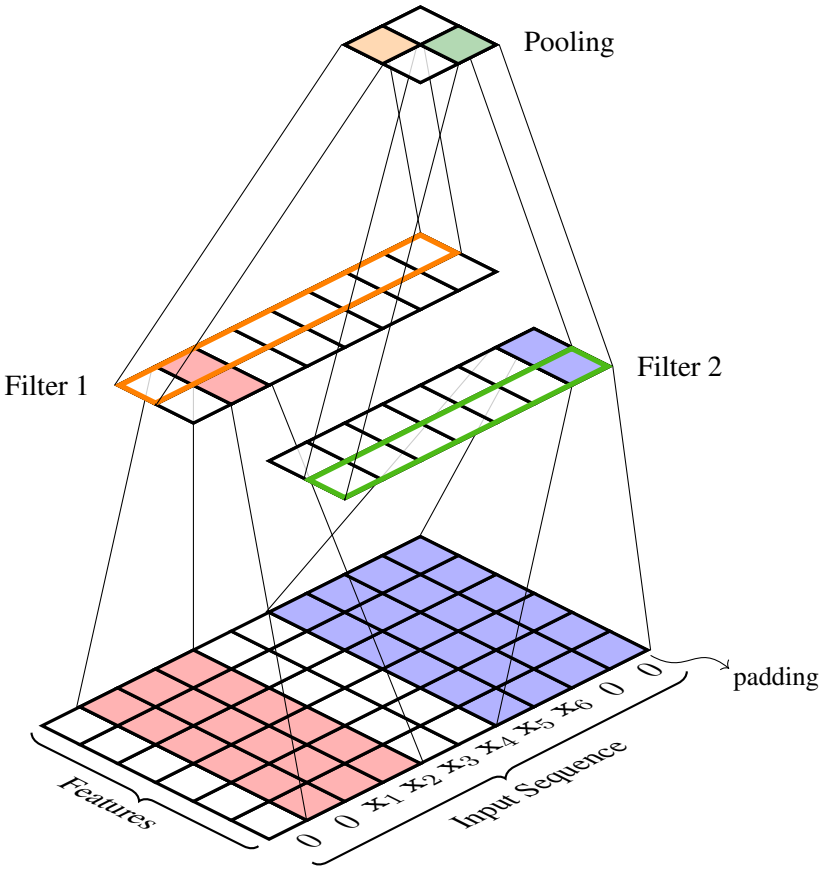


Figure 2.9: Example of CNN. There are two filters for the convolutional layer. The input is a sequence of 6 tokens represented in their feature vectors $\{\mathbf{x}_1, ..., \mathbf{x}_6\}$. To tile the filters to fit the input sequence, two padding vectors are attached to each end of the sequence. When 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 6×3 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 $\mathbf{s}_{i-1} \in \mathbb{R}^{d_h}$ 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 $\mathbf{s}_{i-1}$ 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 $\mathbf{s}_{i-1}$. This can be formulated as:

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

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:

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

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:

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

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

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

where $\mu \in \mathbb{R}^n$ and $\sigma \in \mathbb{R}^n$ are the mean and standard deviation of $\mathbf{s}$, respectively. ϵ 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 α and β together with other parameters.

We may implement Eq. (2.43) in several ways that differ in how to define μ and σ. 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 Ω_k. The mean and the standard deviation on Ω_k are then defined to be:

$$\mu_k = \frac{1}{|\Omega_k|} \cdot \sum_{s \in \Omega_k} s \tag{2.44}$$

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

Several methods are available to define Ω_k. For example, one can define Ω_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., 2016]. 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 $F(\mathbf{x})$, given

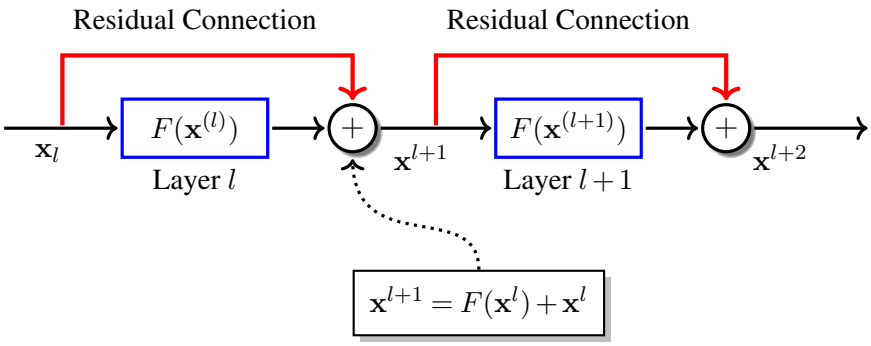


Figure 2.10: A 2-layer residual neural network. For each layer, there is a skip or shortcut connection (in red color) that directly adds the input to the output.

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

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

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:

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

or

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

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

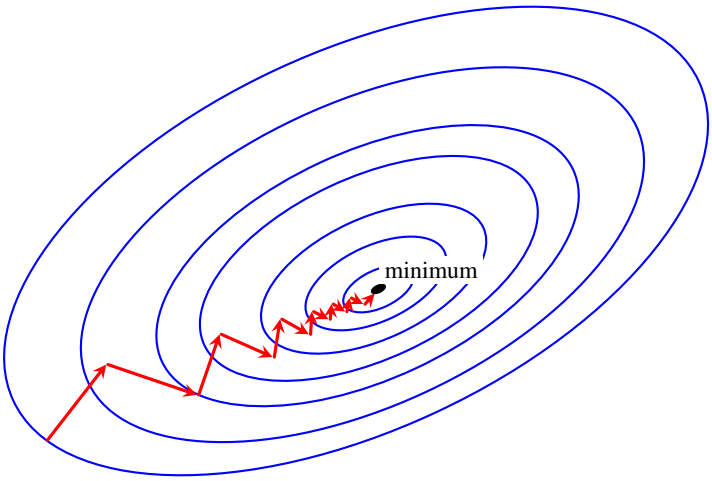


Figure 2.11: Gradient descent in a 2D space (blue lines stand for level sets). The goal is to find the parameters (i.e., values along the two dimensions) that minimize the value of a given loss function. Gradient descent does this by starting at a random point and stepping to the minimum in a number of updates of the parameters. In each update, it adjusts the parameters θ_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 θ_t, we obtain the new parameters θ_{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 θ_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 θ_t. The **update rule** (or **delta rule**) of gradient descent is given by the equation:

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

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 obtained by running the error-propagation algorithm presented in Section 2.1.3. Since θ_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 θ_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:

$$\theta_{t+1} = \theta_t + v_t \tag{2.50}$$

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

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

v_t retains some of the previous momentum (i.e., v_{t-1}), followed by a correction based on the gradient (i.e., $\frac{\partial L(\theta_t)}{\partial \theta_t}$). λ 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

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

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 θ_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[5]:

$$G_t = \sum_{i=1}^{t} \Big[\frac{\partial L(\theta_i)}{\partial \theta_i}\Big]^{\mathrm{T}} \cdot \frac{\partial L(\theta_i)}{\partial \theta_i} \tag{2.54}$$

[5] Given two vectors $\mathbf{a} = \begin{bmatrix} a_1 & \cdots & a_d \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_1 & \cdots & b_d \end{bmatrix}$, the outer product of $\mathbf{a}$ and $\mathbf{b}$ is:

$$\begin{aligned} \mathbf{a} \otimes \mathbf{b} &= \mathbf{a}^{\mathrm{T}} \cdot \mathbf{b} \\ &= \begin{bmatrix} a_1 \\ \vdots \\ a_d \end{bmatrix} \cdot \begin{bmatrix} b_1 & \cdots & b_d \end{bmatrix} \\ &= \begin{bmatrix} a_1 b_1 & \cdots & a_1 b_d \\ \vdots & \ddots & \vdots \\ a_d b_1 & \cdots & a_d b_d \end{bmatrix} \end{aligned} \tag{2.53}$$

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:

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

where $\text{diag}(G_t)$ is the diagonal of G_t, i.e., $\text{diag}(G_t)(k) = G_t(k,k)$. ϵ 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]:

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

where σ 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:

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

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:

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

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

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. β_1 and β_2 are also treated as the decay factors of these averages. For example, common choices for β_1 and β_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:

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

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

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:

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

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**[6], say 20,000 steps, or 100 epochs. As an alternative, we can perform training until convergence. For example, we can say that the training coverages 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 $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}_{\text{gold}})$ be a randomly selected sample from D. Given a neural

[6] A training epoch means that the trainer goes over the whole training dataset for one time.

network

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

the loss of SGD is defined to be:

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

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 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

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

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 $\{(\mathbf{x}^{(i_1)}, \mathbf{y}_{\text{gold}}^{(i_1)}), ..., (\mathbf{x}^{(i_m)}, \mathbf{y}_{\text{gold}}^{(i_m)})\}$. We can batch all input vectors and benchmark vectors as matrices:

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

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

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

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

Likewise, we can compute the batched loss

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

where $L(\mathbf{Y}_\theta, \mathbf{Y}_{\text{gold}})$ vectorizes the computing of $\sum_{k=1}^{m} L\left(\mathbf{y}_\theta^{(i_k)}, \mathbf{y}_{\text{gold}}^{(i_k)}\right)$. 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 minimums, 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_{in} and d_{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 $\mathbf{W}$:

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

  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_{in} and d_{out} in defining the distribution, like this:

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

  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:

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

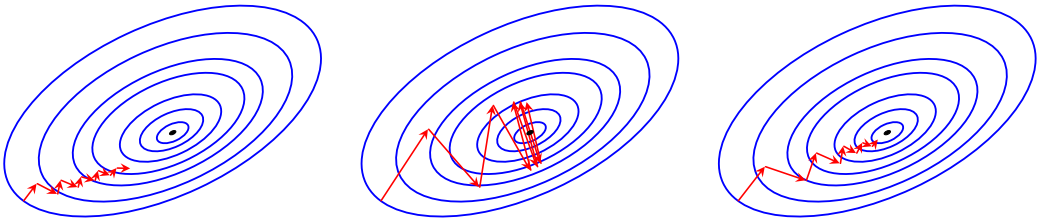
where l is the depth for a layer, and α_s is a hyper-parameter. Apart from this, several methods are proposed to address the initialization of deep neural networks, including the Lipschitz initialization [Xu et al., 2020], the T-Fixup initialization [Huang et al., 2020], the Admin initialization [Liu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



(a) A small learning rate (b) A large learning rate (c) A desirable learning rate

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:

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

where lr_0 is the initial learning rate, and n_{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_{warmup}, n_{decay}, and lr_0 are set to $4,000$, 1 and 1. We see that the learning rate increases linearly in the first n_{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., 2018; 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	$lr_{n_t} = \beta_i$	values $\{\beta_1, ..., \beta_m\}$
	if $\gamma_i \leq \frac{n_t}{n_{\text{decay}}} < \gamma_{i+1}$	thresholds $\{\gamma_1, ..., \gamma_m\}$
Exponential Decay	$lr_{n_t} = lr_0 \cdot \lambda^{\frac{n_t}{n_{\text{decay}}}}$	decay rate λ, init. lr. lr_0
(Drop) Exponential Decay	$lr_{n_t} = lr_0 \cdot \lambda^{\lfloor \frac{n_t}{n_{\text{decay}}} \rfloor}$	decay rate λ, init. lr. lr_0
Natural Exponential Decay	$lr_{n_t} = lr_0 \cdot \exp(-\lambda \cdot \frac{n_t}{n_{\text{decay}}})$	decay rate λ, 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 λ, 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 λ
Cosine Decay	$lr_{n_t} = lr_0 \cdot \big((1-\alpha) \cdot c_{\text{decay}} + \alpha\big)$	coefficient α
	$c_{\text{decay}} = \frac{1}{2} \cdot \big(1 + \cos(\pi \cdot \frac{n_t}{n_{\text{decay}}})\big)$	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.

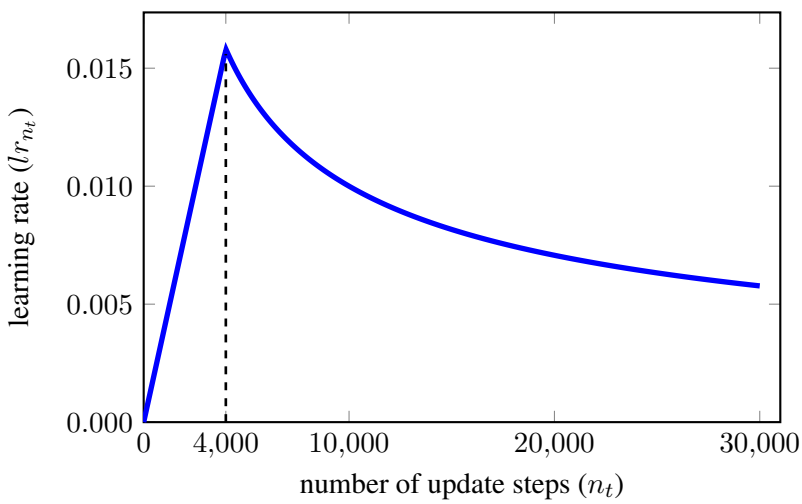


Figure 2.13: Learning rate scheduling: warmup and then decay ($n_{\text{warmup}} = 4,000$, $n_{\text{decay}} = 1$, and $lr_0 = 1$). The learning rate increases linearly with n_t for the first $4,000$ steps. Then, the learning rate follows an inverse square root function and decays as the learning continues. 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:

$$\hat{\theta} = \arg\min_{\theta} L(\theta) + \alpha \cdot R(\theta) \tag{2.74}$$

where $R(\theta)$ is the regularization term weighted by a coefficient α. 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 θ. Let us treat θ as a vector of parameters. The l_1 norm-based regularization term is given by

$$R(\theta) = \sum_i |\theta_i| \tag{2.75}$$

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

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

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.

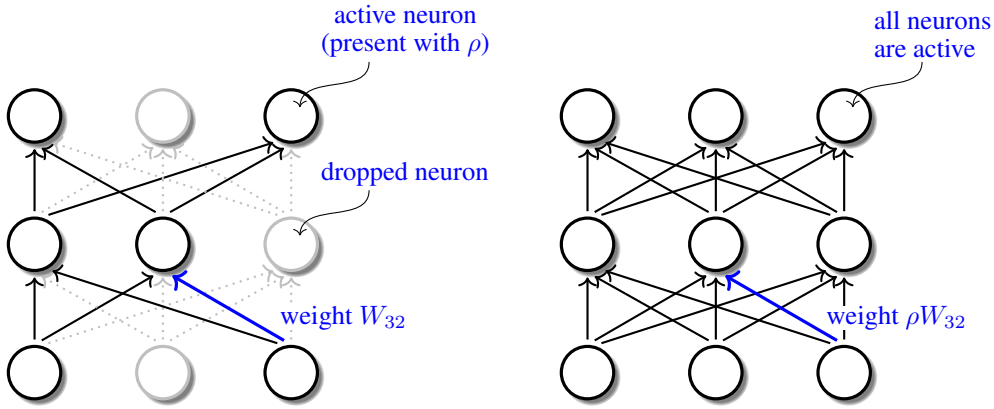
A way to mitigate or eliminate complex co-adaptations is to learn for each neuron to predict in the absence of other neurons. To this end, one can simply drop some of the neurons in training. This method is known as **dropout** [Srivastava et al., 2014]. Let n be the number of neurons of a layer. Given a probability ρ (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 ρ, and set to 0 with a possibility of $1-\rho$. Then, a dropout layer can be defined as

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

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[7]. 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 ρ in training, the learned weights are scaled down with ρ in

[7] For a single-layer network having n neurons, there are 2^n possible sub-networks.



(a) Training the dropout network

(b) Testing with the dropout network

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 ρ during training, the weights of the predicting network are scaled down with ρ.

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

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

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

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

Since multiplying $\frac{1}{\rho}\mathbf{W}$ with ρ 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 $\mathbf{p} = \begin{bmatrix} p_1 & \dots & p_n \end{bmatrix}$, 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 addictive 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

$$p_k = \frac{s_k}{\sum_{k=1}^{n} s_k} \tag{2.80}$$

Then, the smoothed version of p_k is defined as

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

It simply adds a quantity α to each s_k. The value of α 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 α chooses an extremely large value.

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

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

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 β 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:

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

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

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

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

where ϵ 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. ϵ) 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.

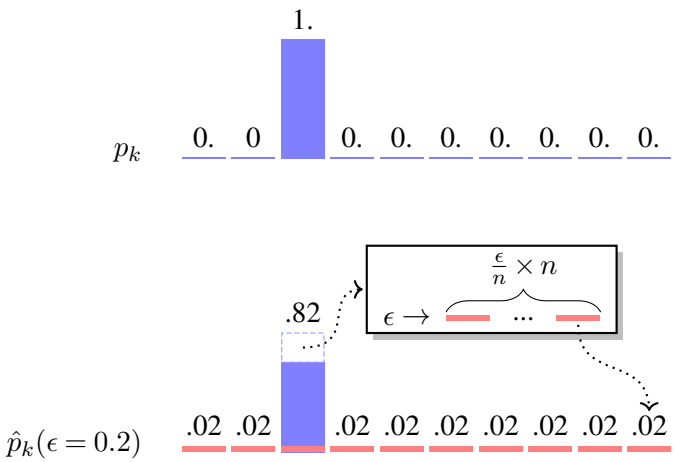


Figure 2.15: Smoothing a distribution by interpolating it with the uniform distribution: $\hat{p}_k = (1-\epsilon) \cdot p_k + \epsilon \cdot \frac{1}{n}$. For each dimension k, it subtracts an amount of ϵ from the probability p_k and redistributes this amount of probability evenly to all the variables, 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[8]. 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, 1996; Kumar and Byrne, 2004].

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

[8] 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

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

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

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

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

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

where μ_k is the mean of the distribution, and σ_k is its standard deviation. Often, μ_k is set to 0. σ_k is a hyper-parameter that is used to control the amount of noise we want to add. For example, a large σ_k means that the random noise spreads out in a large region centered around μ_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

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

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

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

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., 2013; Neelakantan et al., 2015]. For example, the addition of noise to the transformation matrix has the following form:

$$\mathbf{y} \quad = \quad \psi(\mathbf{x} \cdot (\mathbf{W} + \mathbf{g}_{\text{w}}) + \mathbf{B}) \tag{2.91}$$

where $\mathbf{g}_{\text{w}}$ 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:

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

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** [Sennrich et al., 2016]. 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., 2014; 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., 2020]. 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[9]. 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[10].

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:

[9] 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.

[10] 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

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

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:

$$\widetilde{\mathbf{x}} = \text{Dec}(\mathbf{h}) \tag{2.94}$$

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

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

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

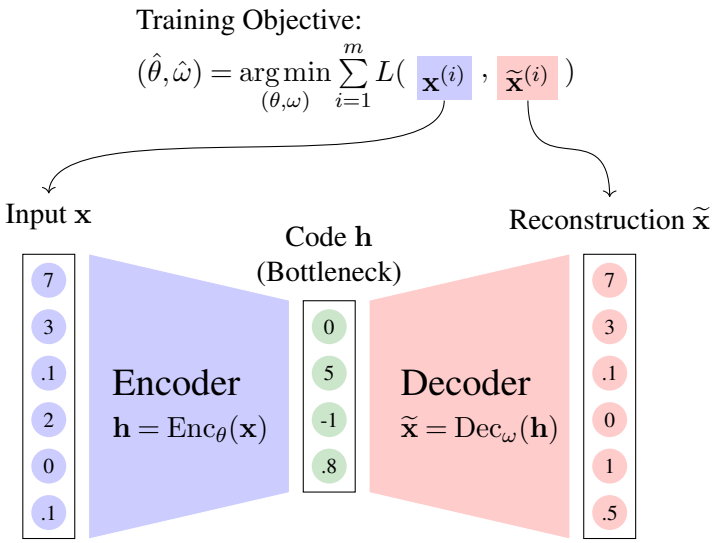


Figure 2.16: An undercomplete auto-encoder with an encoder, a decoder and a bottleneck layer sandwiched between them. An input $\mathbf{x}$ (left) is transformed into a code $\mathbf{h}$ (middle) and then a reconstruction $\widetilde{\mathbf{x}}$ (right). The parameters of both the encoder and decoder are optimized by minimizing the discrepancy between the input $\mathbf{x}$ and the reconstruction $\widetilde{\mathbf{x}}$ on a number of unlabeled samples $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$. 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 $\widetilde{\mathbf{x}}$, 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 $\widetilde{\mathbf{x}}$. Figure 2.16 shows an illustration of the undercomplete auto-encoder structure.

Given the loss function $L(\cdot)$, the encoder $\mathrm{Enc}_{\theta}(\cdot)$ and the decoder $\mathrm{Dec}_{\omega}(\cdot)$, 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 $\mathbf{x}_{\mathrm{new}}$, we generate a new representation

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

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

$$(\hat{\theta},\hat{\omega}) = \arg\min_{(\theta,\omega)} \sum_{i=1}^{m} L\left(\mathbf{x}^{(i)}, \mathrm{Dec}_{\omega}(\mathrm{Enc}_{\theta}(\mathbf{x}^{(i)}))\right) + \alpha \cdot R \tag{2.97}$$

where R is the regularization term accounting for some prior knowledge we want to impose on training, and α 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

$$R_{l_1} = \sum_{i=1}^{m} \sum_{k} \left| h_k^{(i)} \right| \tag{2.98}$$

$$R_{l_2} = \sum_{i=1}^{m} \sqrt{\sum_{k} \left(h_k^{(i)}\right)^2} \tag{2.99}$$

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[11]. Let $\bar{\mathbf{h}}$ denote the average code over $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$, where the value of $\bar{h}_k$ is the mean of the k-th variable of the code:

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

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 τ, then the regularization term can be defined as the sum of the KL divergence between $\bar{h}_k$ and q_k over all entries:

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

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]:

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

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 [12], and $\|\cdot\|_F$ is

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

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

the **Frobenius norm** of a matrix [13]. 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 $\mathrm{Enc}(\cdot)$ to behave more like a **contraction mapping**[14], 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 $\mathbf{x}$:

$$\mathbf{x}_{\text{noise}} \sim \Pr_{\text{noise}}(\mathbf{x}_{\text{noise}}|\mathbf{x}) \tag{2.106}$$

where $\Pr_{\text{noise}}(\cdot)$ 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:

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

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

$$\mathbf{x}_{\text{noise}} = \mathbf{x} + \mathbf{g} \tag{2.108}$$

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

---

a $d_h \times d_x$ matrix:

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

[13] For a matrix $\mathbf{A} \in \mathbb{R}^{d_h \times d_x}$, the Frobenius norm is given by the equation:

$$\|\mathbf{A}\| = \sqrt{\sum_{i,j} A_{i,j}^2} \tag{2.104}$$

[14] 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 ϵ such that for any $x_1, x_2 \in X$:

$$d(f(x_1), f(x_2)) \leq \epsilon \cdot d(x_1, x_2) \tag{2.105}$$

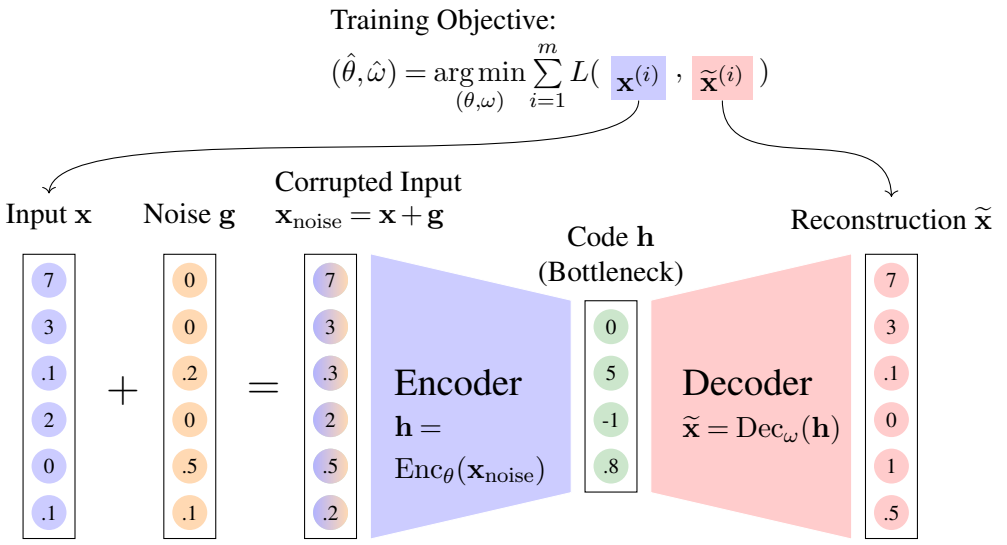


Figure 2.17: The structure of a denoising auto-encoder. An input $\mathbf{x}$ is first corrupted into a noisy or corrupted input $\mathbf{x}_{\text{noise}}$. Then, it is passed through an encoder to form a code $\mathbf{h}$. Then, the code is passed through a decoder to form a reconstructed input $\widetilde{\mathbf{x}}$. The training is performed by minimizing the loss between $\mathbf{x}$ and $\widetilde{\mathbf{x}}$. 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 $\widetilde{\mathbf{x}} = \text{Dec}(\text{Enc}(\mathbf{x}_{\text{noise}}))$. The training process is regular. We reuse Eq. (2.95) to minimize the loss of replacing $\mathbf{x}$ with $\widetilde{\mathbf{x}}$. Thus, we can rewrite Eq. (2.95) to adapt the objective to the denoising case:

$$(\hat{\theta},\hat{\omega}) = \arg\min_{(\theta,\omega)} \sum_{i=1}^{m} L\left(\mathbf{x}^{(i)}, \text{Dec}_\omega(\text{Enc}_\theta(\mathbf{x}^{(i)}_{\text{noise}}))\right) \tag{2.110}$$

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:

$$\Pr(\mathbf{x}) = \int \Pr(\mathbf{x}, \mathbf{h}) d\mathbf{h} \tag{2.111}$$

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

$$p_\omega(\mathbf{x}) = \int p_\omega(\mathbf{x}, \mathbf{h}) d\mathbf{h} \tag{2.112}$$

where $p_\omega(\mathbf{x}, \mathbf{h})$ is a probability density function parameterized by ω. 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:

$$p_\omega(\mathbf{x}) = \int p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h}) d\mathbf{h} \tag{2.113}$$

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 μ_p and variance σ_p:

$$p_\omega(\mathbf{x}|\mathbf{h}) = \text{Gaussian}(\mu_p, \sigma_p) \tag{2.114}$$

where μ_p and σ_p are determined by a decoding network $\text{Dec}_\omega(\cdot)$ (we will explain later on why it is called "decoding"):

$$(\mu_p, \sigma_p) = \text{Dec}_\omega(\mathbf{h}) \tag{2.115}$$

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:

$$p_\omega(\mathbf{h}|\mathbf{x}) = \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{p_\omega(\mathbf{x})} \tag{2.116}$$

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 $q_\theta(\mathbf{h}|\mathbf{x})$:

$$q_\theta(\mathbf{h}|\mathbf{x}) \approx p_\omega(\mathbf{h}|\mathbf{x}) \tag{2.117}$$

where θ 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:

$$q_\theta(\mathbf{h}|\mathbf{x}) = \text{Gaussian}(\mu_q, \sigma_q) \tag{2.118}$$
$$(\mu_q, \sigma_q) = \text{Enc}_\theta(\mathbf{x}) \tag{2.119}$$

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 $\Pr(\mathbf{x})$: 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

$$\begin{aligned}
\log \Pr(\mathbf{x}) &\equiv \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log p_\omega(\mathbf{x}) \\
&= \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{p_\omega(\mathbf{h}|\mathbf{x})} \\
&= \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{p_\omega(\mathbf{h}|\mathbf{x})} \cdot \frac{q_\theta(\mathbf{h}|\mathbf{x})}{q_\theta(\mathbf{h}|\mathbf{x})} \\
&= \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{q_\theta(\mathbf{h}|\mathbf{x})}{p_\omega(\mathbf{h}|\mathbf{x})} + \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})}
\end{aligned} \tag{2.120}$$

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.,

$$D(q_\theta(\mathbf{h}|\mathbf{x})||p_\omega(\mathbf{h}|\mathbf{x})) = \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{q_\theta(\mathbf{h}|\mathbf{x})}{p_\omega(\mathbf{h}|\mathbf{x})} \tag{2.121}$$

Thus, given $D(q_\theta(\mathbf{h}|\mathbf{x})||p_\omega(\mathbf{h}|\mathbf{x})) \geq 0$, we have[15]

$$\begin{aligned}
\log \Pr(\mathbf{x}) &\geq \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log \frac{p_\omega(\mathbf{h}) \cdot p_\omega(\mathbf{x}|\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})} \\
&= \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \left[\log p_\omega(\mathbf{x}|\mathbf{h}) + \log \frac{p_\omega(\mathbf{h})}{q_\theta(\mathbf{h}|\mathbf{x})} \right] \\
&= \mathbb{E}_{\mathbf{h}\sim q_\theta(\mathbf{h}|\mathbf{x})} \log p_\omega(\mathbf{x}|\mathbf{h}) + D(p_\omega(\mathbf{h})||q_\theta(\mathbf{h}|\mathbf{x}))
\end{aligned} \tag{2.122}$$

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 $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$:

$$(\hat{\theta}, \hat{\omega}) = \arg\min_{(\theta,\omega)} \sum_{i=1}^{m} L\left(\mathbf{x}^{(i)}, \theta, \omega\right) \tag{2.123}$$

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:

[15] The KL divergence between p and q is zero only if $p = q$, and is positive otherwise.

$$\text{Training Objective: } (\hat{\theta}, \hat{\omega}) = \arg\min_{(\theta,\omega)} \sum_{i=1}^{m} L(\mathbf{x}^{(i)}, \theta, \omega)$$

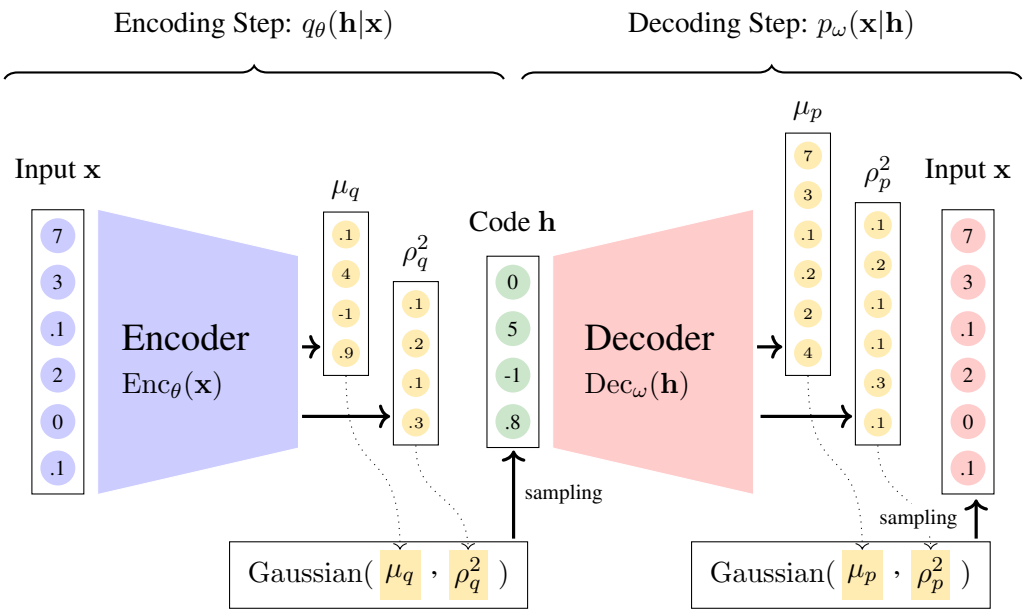


Figure 2.18: The generative story of a variational auto-encoder. For an input sample $\mathbf{x}$, we generate a latent variable $\mathbf{h}$ by using an encoder $q_\theta(\mathbf{h}|\mathbf{x})$. In the encoding step, a neural network $\text{Enc}_\theta(\cdot)$ is first used to produce the mean and variance of a Gaussian distribution, say, μ_q and σ_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 μ_p and variance σ_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 μ_q and variance σ_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 μ_p and variance σ_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[16].

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.

[16] URLs to a few popular online tutorials: https://pytorch.org/tutorials, https://keras.io/examples/nlp, and https://www.tensorflow.org/tutorials

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.

[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.

[Ba et al., 2016] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[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.

[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(Feb):1137–1155, 2003.

[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.

[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.

[Chiang and Cholak, 2022] David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.

[Chollet, 2021] François Chollet. *Deep Learning with Python (2nd ed.)*. Manning Publications, 2021.

[Cybenko, 1989] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[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.

[Doersch, 2016] Carl Doersch. Tutorial on variational autoencoders. *stat*, 1050:13, 2016.

[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.

[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.

[Elman, 1990] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[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.

[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.

[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.

[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.

[Goodman, 1996] Joshua Goodman. Parsing algorithms and metrics. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 177–183, 1996.

[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, 2013.

[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.

[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, 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.

[Hinton, 2007] Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007.

[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.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[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.

[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.

[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, 2020.

[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.

[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.

[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.

[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.

[Knight, 2009] Kevin Knight. Bayesian inference with tears, 2009.

[Kochenderfer and Wheeler, 2019] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. The MIT Press, 2019.

[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, 2004.

[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.

[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 2020.

[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.

[Minsky and Papert, 1969] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT press, 1969.

[Mohri et al., 2018] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning (2nd ed.)*. MIT Press, 2018.

[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.

[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.

[Nesterov, 1983] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[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.

[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 2018.

[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.

[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.

[Prechelt, 1998] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[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.

[Rosenblatt, 1957] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[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.

[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, 2016.

[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.

[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.

[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.

[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.

[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*, 2014.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Xu et al., 2020] Hongfei Xu, Qiuhui 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.

[Yao et al., 2007] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26:289–315, 2007.

[Zeiler, 2012] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[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 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, 2020.