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 9

Prompting

In the context of LLMs, *prompting* refers to the method of providing an LLM with a specific input or cue to generate a desired output or perform a task. For example, if we want the LLM to translate a sentence from English to Chinese, we can prompt it like this

> Translate the text from English to Chinese.
>
> Text: The early bird catches the worm.
>
> Translation: ____

Prompting is crucial for LLMs because it directly influences how effectively these models understand and respond to user queries. A well-crafted prompt can guide an LLM to generate more accurate, relevant, and contextually appropriate responses. Furthermore, this process can be iteratively refined. By analyzing the responses of the LLM, users can adjust their prompts to align more closely with their specific needs. Given the importance of prompting in applying LLMs, prompt design has become an essential skill for users and developers working with LLMs. This leads to an active research area, called **prompt engineering**, in which we design effective prompts to make better use of LLMs and enhance their practical utility in real-world applications.

An important concept related to prompting is **in-context learning**. When prompting an LLM, we can add new information to the context, such as demonstrations of problem-solving. This allows the LLM to learn from this context how to solve the problem. Here is an example of prompting LLMs with a few demonstrations of how to classify text based on sentiment polarity.

> Here are some examples of text classification.
> Example 1: We had a delightful dinner together. → Label: Positive
> Example 2: I'm frustrated with the delays. → Label: Negative
> What is the label for "That comment was quite hurtful."?
> Label: ____

In-context learning is often seen as an emergent ability of LLMs that arises after pre-training. Though LLMs can be trained or tuned to perform new tasks, in-context learning provides a very efficient way to adapt these models without any training or tuning effort. Perhaps this is one of the most notable features of LLMs: they indeed learn general knowledge about the world and language during pre-training, which we can easily apply to new challenges. Moreover, in-context learning reflects the broader trend of making AI systems more generalizable and user-friendly. Instead of requiring specialized engineers to fine-tune models for every unique task, users can interact with LLMs in a more intuitive way, simply providing examples or adjusting the context as needed.

In this chapter, we focus on prompting techniques in LLMs. We begin by considering several interesting prompt designs commonly used in prompt engineering. Then, we discuss a series of refinements to these methods. Finally, we explore approaches for automating prompt design.

9.1 General Prompt Design

This section presents basic concepts in prompt design, along with examples of how to prompt LLMs for various NLP tasks. Since the effectiveness of prompting is highly dependent on the LLMs being used, prompts often vary across different LLMs, making it difficult to provide a comprehensive list of prompts for all LLMs and downstream tasks. Therefore, this discussion is not focused on any specific LLM. Instead, the goal is to provide guiding principles for prompt design.

9.1.1 Basics

The term *prompt* is used in many different ways. In this chapter we define a prompt as the input text to an LLM, denoted by $\mathbf{x}$. The LLM generates a text $\mathbf{y}$ by maximizing the probability $\Pr(\mathbf{y}|\mathbf{x})$. In this generation process, the prompt acts as the condition on which we make predictions, and it can contain any information that helps describe and solve the problem.

A prompt can be obtained using a prompt template (or template for short) [Liu et al., 2023a]. A template is a piece of text containing placeholders or variables, where each placeholder can be filled with specific information. Here are two templates for asking the LLM for weekend suggestions.

> Please give me some suggestions for a fun weekend.
>
> ——

> If $\{*\text{premise}*\}$, what are your suggestions for a fun weekend.
>
> ——

In the first template, we simply instruct the LLM to return some suggestions. So the template is just a piece of text with no variables. In the second template, the variable $\{*\text{premise}*\}$ needs to be specified by the users to provide a premise for making suggestions. For example, if we input

$$\text{premise} \quad = \quad \text{the weather is nice this weekend}$$

then we can generate a prompt

> If the weather is nice this weekend,
> what are your suggestions for a fun weekend.
>
> ——

We can also design a template with multiple variables. Here is an example in which we compare the two sentences in terms of their semantic similarity.

> Here is a sentence
> $\{*\text{sentence1}*\}$
> Here is another sentence
> $\{*\text{sentence2}*\}$
>
> Compute the semantic similarity between the two sentences
>
> ——

A popular way to format prompts is to write each input or output in a "name:content" style. For example, we can describe a conversation between two people, named John and David, and use the LLM to continue the conversation. A template of such prompts is given by

John: {∗utterance1∗}
David: {∗utterance2∗}
John: {∗utterance3∗}
David: {∗utterance4∗}
John: {∗utterance5∗}
David: {∗utterance6∗}
John: {∗utterance7∗}
David: ____

The "name:content" format can be used to define the task that we want the LLM to perform. For example, given that "Q" and "A" are commonly used abbreviations for "Question" and "Answer", respectively, we can use the following template to do question-answering.

Q: {∗question∗}
A: ____

This format can be used to describe more complex tasks. For example, the following is an example of providing a specification for a translation task

Task: Translation
Source language: English
Target language: Chinese
Style: Formal text
Template: Translate the following sentence: {∗sentence∗}
____

In practical systems, it is common to represent and store such data in key-value pairs, such as the JSON format[1].

When the problem is difficult to describe in an attribute-based manner, it is more common to instruct LLMs with a clear and detailed description. There are many ways to do this. One

[1]The JSON representation is

```
{
    "Task": "Translation"
    "Source language": "English"
    "Target language": "Chinese"
    "Style": "Formal text"
    "Template": "Translate the following sentence: {∗sentence∗}"
}
```

example is to assign a role to LLMs and provide sufficient context. The following is a template that instructs an LLM to act as an expert and answer questions from children.

> You are a computer scientist with extensive knowledge in the field of deep learning.
>
> Please explain the following computer-related concept to a child around 10 years old, using simple examples whenever possible.
>
> {∗concept∗}
>
> ____

Here the text "You are a computer scientist ... deep learning. " is sometimes called system information, and is provided to help the LLM understand the context or constraints of the task it is being asked to perform.

9.1.2 In-context Learning

Learning can occur during inference. In-context learning is one such method, where prompts involve demonstrations of problem-solving, and LLMs can learn from these demonstrations how to solve new problems. Since we do not update model parameters in this process, in-context learning can be viewed as a way to efficiently activate and reorganize the knowledge learned in pre-training without additional training or fine-tuning. This enables quick adaptation of LLMs to new problems, pushing the boundaries of what pre-trained LLMs can achieve without task-specific adjustments.

In-context learning can be illustrated by comparing three methods: zero-shot learning, one-shot learning and few-shot learning. Zero-shot learning, as its name implies, does not involve a traditional "learning" process. It instead directly applies LLMs to address new problems that were not observed during training. In practice, we can repetitively adjust prompts to guide the LLMs in generating better responses, without demonstrating problem-solving steps or providing examples. Consider the following example. Suppose we want to use an LLM as an assistant that can help correct English sentences. A zero-shot learning prompt is given by

> SYSTEM You are a helpful assistant, and are great at grammar correction.
>
> USER You will be provided with a sentence in English. The task is to output the correct sentence.
>
> Input: She don't like going to the park.
> Output: ____

Here the gray words are used to indicate different fields of the prompt.

In one-shot learning, we extend this prompt by adding a demonstration of how to correct sentences, thereby allowing the LLM to learn from this newly-added experience.

SYSTEM	You are a helpful assistant, and are great at grammar correction.
DEMO	You will be provided with a sentence in English. The task is to output the correct sentence. Input: There is many reasons to celebrate. Output: There are many reasons to celebrate.
USER	You will be provided with a sentence in English. The task is to output the correct sentence. Input: She don't like going to the park. Output: ____

Furthermore, we can add more demonstrations to enable few-shot learning.

SYSTEM	You are a helpful assistant, and are great at grammar correction.
DEMO1	You will be provided with a sentence in English. The task is to output the correct sentence. Input: There is many reasons to celebrate. Output: There are many reasons to celebrate.
DEMO2	You will be provided with a sentence in English. The task is to output the correct sentence. Input: Me and my friend goes to the gym every day. Output: My friend and I go to the gym every day.
USER	You will be provided with a sentence in English. The task is to output the correct sentence. Input: She don't like going to the park. Output: ____

In few-shot learning, we essentially provide a pattern that maps some inputs to the corresponding outputs. The LLM attempts to follow this pattern in making predictions, provided that the prompt includes a sufficient number of demonstrations, although generally small. It is also possible to use simpler patterns to achieve this. For example, one can use the following few-shot learning prompt for translating words from Chinese to English.

DEMO	现在	→	now
	来	→	come
	去	→	go
	男孩	→	boy
USER	女孩	→	____

If the LLM is powerful enough, few-shot learning can enable it to address complex problems, such as mathematical reasoning. For example, consider the following task of summing two numbers and then dividing the sum by their product.

DEMO	12 5	→	$(12+5)/(12\times 5) = 0.283$
	3 1	→	$(3+1)/(3\times 1) = 1.33$
	−9 4	→	$(-9+4)/(-9\times 4) = 0.138$
	15 15	→	$(15+15)/(15\times 15) = 0.133$
USER	19 73	→	____

In many practical applications, the effectiveness of in-context learning relies heavily on the quality of prompts and the fundamental abilities of pre-trained LLMs. On one hand, we need a significant prompt engineering effort to develop appropriate prompts that help LLMs learn more effectively from demonstrations. On the other hand, stronger LLMs can make better use of in-context learning for performing new tasks. For example, suppose we wish to use an LLM to translate words from Inuktitut to English. If the LLM lacks pre-training on Inuktitut data, its understanding of Inuktitut will be weak, and it will be difficult for the model to perform well in translation regardless of how we prompt it. In this case, we need to continue training the LLM with more Inuktitut data, rather than trying to find better prompts.

It might be interesting to explore how in-context learning emerges during pre-training and why it works during inference. One simple understanding is that LLMs have gained some knowledge of problem-solving, but there are many possible predictions, which are hard to distinguish when the models confront new problems. Providing demonstrations can guide the LLMs to follow the "correct" paths. Furthermore, some researchers have tried to interpret in-context learning from several different perspectives, including Bayesian inference [Xie et al., 2022], gradient descent [Dai et al., 2023; Von Oswald et al., 2023], linear regression [Akyürek et al., 2023], meta learning [Garg et al., 2022], and so on.

9.1.3 Prompt Engineering Strategies

Designing prompts is highly empirical. In general, there are many ways to prompt an LLM for performing the same task, and we need to perform a number of trial-and-error runs to find a satisfactory prompt. To write good prompts more efficiently, one can follow certain strategies. Examples of common prompting principles include

- **Describing the task as clearly as possible**. When we apply an LLM to solve a problem, we need to provide a precise, specific, and clear description of the problem and instruct the LLM to perform as we expect. This is particularly important when we want the output of the LLM to meet certain expectations. For example, suppose we are curious about climate change. A simple prompt for asking the LLM to provide some information is

  > Tell me about climate change.
  >
  > ——

  Since this instruction is too general, the LLM may generate a response that addresses any aspect of climate change, which may not align with our specific interests. In this case, we can instead use prompts that are specific and detailed. One such example is

  > Provide a detailed explanation of the causes and effects of climate change, including the impact on global temperatures, weather patterns, and sea levels. Also, discuss possible solutions and actions being taken to mitigate these effects.
  >
  > ——

  Now suppose we intend to explain climate change to a 10-year-old child. We can adjust the above prompt further.

  > Explain the causes and effects of climate change to a 10-year-old child. Talk about how it affects the weather, sea levels, and temperatures. Also, mention some things people are doing to help. Try to explain in simple terms and do not exceed 500 words.
  >
  > ——

- **Guiding LLMs to think**. LLMs have exhibited surprisingly good capabilities to "think". A common example is that well-developed LLMs have achieved impressive performance in mathematical reasoning tasks, which are considered challenging. In prompt engineering, the "thinking" ability of LLMs needs to be activated through appropriate prompting, especially for problems that require significant reasoning efforts. In many cases, an LLM that is instructed to "think" can produce completely different results compared with the same LLM that is instructed to perform the task straightforwardly. For example, Kojima et al. [2022] found that simply appending "Let's think step by step" to the end of each prompt can improve the performance of LLMs on several reasoning tasks. LLMs can be prompted to "think" in a number of ways. One method is to instruct LLMs to

generate steps for reasoning about the problem before reaching the final answer. For example, consider a task of solving mathematical problems. See below for a simple prompt for this task.

> You are a mathematician. You will be provided with a math problem. Please solve the problem.
>
> ——

Since solving math problems requires a detailed reasoning process, LLMs would probably make mistakes if they attempted to work out the answer directly. So we can explicitly ask LLMs to follow a given reasoning process before coming to a conclusion.

> You are a mathematician. You will follow these detailed reasoning steps when solving math problems.
>
> Step 1: Problem Interpretation.
> The mathematician carefully listens to your query and understands the intricate details of the mathematical challenge you have presented.
>
> Step 2: Strategy Formulation.
> Drawing upon their extensive knowledge, the mathematician chooses the most effective strategy tailored to the type of math problem, whether it is algebra, calculus, or geometry.
>
> Step 3: Detailed Calculation.
> With precision and expertise, the mathematician performs the necessary calculations step by step, adhering to all mathematical principles.
>
> Step 4: Solution Review.
> Before providing the final answer, the mathematician meticulously checks the calculations for accuracy and offers a concise explanation or rationale for the solution.
>
> You will be provided with a math problem. Please solve the problem.
>
> $\{*\text{problem}*\}$
>
> ——

Another method to guide LLMs to "think" is through multiple rounds of interaction with LLMs. For example, as a first step, we can instruct LLMs to solve the problem directly

> You will be provided with a math problem. Please solve the problem.
>
> {∗problem∗}
>
> ——

Now we have an initial answer to the problem. As a second step, we prompt LLMs to evaluate the correctness of the answer and, if necessary, rework it to find a better solution.

> You will be provided with a math problem, along with a solution. Evaluate the correctness of this solution, and identify any errors if present. Then, work out your own solution.
>
> Problem: {∗problem∗}
>
> Solution: {∗solution∗}
>
> ——

The prompts presented here are closely related to a long line of research on reasoning problems in LLMs. It is impossible to provide a complete discussion of all related issues because this topic covers a large family of methods. But we will see a relatively more detailed discussion on how to improve prompting through more reasoning in Section 9.2.

- **Providing reference information**. As discussed in the previous section, we can include demonstrations in prompts and allow LLMs to in-context learn from these demonstrations how to perform the task. In fact, given the remarkable ability of language understanding of LLMs, we can add any type of text into the prompts and so these models can predict based on enriched contexts. In many applications, we have various information that is relevant to user queries. Instead of using LLMs to make unconstrained predictions, we often want LLMs to produce outputs that are confined to the relevant text. One such example is RAG, where the relevant text for the user query is provided by calling an IR system, and we prompt LLMs to generate responses based on this provided relevant text. The following prompt shows an example.

> You are an expert that can generate answers to input queries. You have now been provided with a query and the corresponding context information. Please generate an answer based on this context information. Note that you need to provide the answer in your own words, not just copy from the context provided.
>
> Context information: {∗IR-result∗}
>
> Query: {∗query∗}
>
> ——

If the context information is highly reliable, we can even restrict LLMs to answering using only the provided text. An example prompt is shown as follows

> You are an expert tasked with generating answers from input queries. You have been provided with a query and corresponding context information, organized in a table where each row represents a useful record. Please generate an answer using only this context information. Ensure that you provide the answer in your own words.
>
> Context information: {∗table∗}
>
> Query: {∗query∗}
>
> ——

When dealing with real-world problems, we often have prior knowledge and additional information about the problems that help produce better answers. Considering such information in prompting is generally helpful in improving the result.

- **Paying attention to prompt formats**. In general, the performance of LLMs is highly sensitive to the prompts we input. Sometimes a small modification to a prompt can lead to a big change in model output. An interesting example is that changing the order of sentences in a prompt may cause LLMs to generate different results. To make prompts easy to read and reduce ambiguity, it is common to format them in a way that ensures clarity. One example is that we define several fields for prompts and fill different information in each field. Another example is we can use code-style prompts for LLMs which can understand and generate both natural language and code. See the following for a code-style prompt that performs translation where one demonstration is presented.

> [English] = [I have an apple.]
> [German] = [Ich habe einen Apfel.]
> [English] = [I have an orange.]
> [German] = ____

LLMs can receive text in various formats. This allows us to use control characters, XML tags, and specific formatting to represent complex data. And it is useful to specify how the input and output should be formatted or structured. For example, we can delimit sections of text using quotes and prompt LLMs accordingly (e.g., adding a sentence like "the input text is delimited by double quotes" to the prompt).

Above, we have discussed only a few strategies for writing good prompts. There are, of course, many such methods, and one needs to develop their own through practice. Interested readers can refer to various online documents for more information, such as OpenAI's manual on the GPT series models[2].

9.1.4 More Examples

In this subsection, we consider more examples of prompting LLMs to perform various NLP tasks. The motivation here is not to give standard prompts for these tasks, but rather to use simple examples to illustrate how LLMs can be prompted to deal with NLP problems.

1. Text Classification

Text classification is perhaps one of the most common problems in NLP. Many tasks can be broadly categorized as assigning pre-defined labels to a given text. Here we consider the polarity classification problem in sentiment analysis. We choose polarity classification for illustration because it is one of the most popular and well-defined text classification tasks. In a general setup of polarity classification, we are required to categorize a given text into one of three categories: negative, positive, or neutral. Below is a simple prompt for doing this (for easy reading, we highlight the task description in the prompt).

> Analyze the polarity of the following text and classify it as positive, negative, or neutral.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> The polarity of the text can be classified as negative.

[2] See https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results.

To make the example complete, we show the response generated by the LLM (underlined text).

Although the answer is correct, the LLM gives this answer not in labels but in text describing the result. The problem is that LLMs are designed to generate text but not to assign labels to text and treat classification problems as text generation problems. As a result, we need another system to map the LLM's output to the label space (call it **label mapping**), that is, we extract "negative" from "The polarity of the text can be classified as negative". This is trivial in most cases because we can identify label words via simple heuristics. But occasionally, LLMs may not express the classification results using these label words. In this case, the problem becomes more complicated, as we need some way to map the generated text or words to predefined label words.

One method to induce output labels from LLMs is to reframe the problem as a cloze task. For example, the following shows a cloze-like prompt for polarity classification.

> Analyze the polarity of the following text and classify it as positive, negative, or neutral.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> The polarity of the text is negative

We can use LLMs to complete the text and fill the blank with the most appropriate word. Ideally, we wish the filled word would be $\mathrm{positive}$, $\mathrm{negative}$, or $\mathrm{neutral}$. However, LLMs are not guaranteed to generate these label words. One method to address this problem is to constrain the prediction to the set of label words and select the one with the highest probability. Then, the output label is given by

$$\mathrm{label} \quad = \quad \arg\max_{y \in Y} \Pr(y|\mathbf{x}) \tag{9.1}$$

where y denotes the word filled in the blank, and Y denotes the set of label words $\{\mathrm{positive}, \mathrm{negative}, \mathrm{neutral}\}$.

Another method of using LLMs to generate labels is to constrain the output with prompts. For example, we can prompt LLMs to predict within a controlled set of words. Here is an example.

> Analyze the polarity of the following text and classify it as positive, negative, or neutral.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> What is the polarity of the text?
>
> Just answer: positive, negative, or neutral.
>
> Negative

Sentiment analysis is a common NLP problem that has probably been well understood by LLMs through pre-training or fine-tuning. Thus we can prompt LLMs using simple instructions to perform the task. However, for new classification problems, it may be necessary to provide additional details about the task, such as the classification standards, so that the LLMs can perform correctly. To do this, we can add a more detailed description of the task and/or demonstrate classification examples in the prompts. To illustrate, consider the following example.

> Analyze the polarity of the following text and classify it as positive, negative, or neutral. Here's what each category represents:
>
> Positive: This indicates that the text conveys a positive emotion or attitude. For example, texts expressing happiness, satisfaction, excitement, or admiration are considered positive.
>
> Negative: This refers to a text that expresses a negative emotion or attitude. It encompasses feelings of sadness, anger, frustration, or criticism.
>
> Neutral: Neutral sentiment is used to describe texts that do not exhibit clear positive or negative emotions but instead convey informational, factual, or indifferent tones.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> What is the polarity of the text?
>
> Negative

While it seems straightforward to use LLMs for classification problems, there are still issues that have not been well addressed. For example, when dealing with a large number of categories, it remains challenging to effectively prompt LLMs. Note that if we face a very

difficult classification problem and have a certain amount of labeled data, fine-tuning LLMs or adopting "BERT + classifier"-like architectures is also desirable.

2. Information Extraction

Many NLP problems can be regarded as information extraction problems, involving the identification or extraction of specific pieces of information from unstructured text. This information can include named entities, relationships, events, and other relevant data points. The goal of information extraction is to transform raw data into a format that can be easily analyzed and used in various downstream applications.

As information extraction covers a wide range of problems, we cannot discuss them all here. Instead, we start with the task of named entity recognition — a task that has long been a concern in NLP. Named entity recognition is a process that detects and classifies key information in text into specific groups. These key pieces of information, known as named entities, typically include proper names and are categorized into distinct classes such as people, locations, organizations, dates, monetary values, and percentages. Consider first a simple example of extracting person names from a text[3].

> Identify all person names in the provided text.
>
> Text:
> Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."
>
> The person name identified in the provided text is:
>
> Tom Jenkins

Then, we can extract all named entities.

[3]The text is from https://edition.cnn.com/travel

Identify and classify all named entities in the provided text into categories such as person names, locations, dates, and organizations. List each entity with its type on one line.

Text:
Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

1. Tom Jenkins - Person Name
2. European Tourism Organisation - Organization
3. UK - Location
4. Europe - Location
5. 2024 - Date

Given these identified named entities, we can further process the text using other information extraction tasks. For example, we can identify the relationships between these named entities (call it **relation extraction**). The corresponding prompt is shown as follows.

Given a text and a list of named entities identified within it, analyze and describe the relationships between these entities. Explain how each entity is contextually related to others.

Text:
Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

Named Entity:
Tom Jenkins, European Tourism Organisation, UK, Europe, 2024

Tom Jenkins - European Tourism Organisation: Tom Jenkins is the CEO of the European Tourism Organisation. This indicates a professional association where Tom Jenkins likely has a significant role in directing the organization's activities, especially those related to tourism in Europe.

...

If LLMs have been fine-tuned with instruction following for information extraction tasks, it is relatively easy to perform various information extraction tasks. For example, the following is a prompt template for information extraction.

> You will be provided with a text. Your task is to {∗task-description∗}
>
> Text: {∗text∗}
>
> ——

Here task-description is the instruction for the task. Examples of the instruction include

> *extract the keywords from the text.*
>
> *extract the key events from the text.*
>
> *detect the coreference and anaphoric links between entities from the text.*

and so on. Of course, we can provide a more detailed description of the task to prompt LLMs more accurately.

3. Text Generation

In NLP, the term *text generation* is used broadly to cover a wide range of different problems, many of which focus on generating text based on given text. These problems can be roughly categorized into two classes: 1) continual writing based on the input text (call it **text completion**), and 2) transformation of the input text into another text (call it **text transformation**). Here, we illustrate the prompts for the former and consider the prompts for the latter in the following discussion.

A simple example of text completion is continuing to write a story starting with a few given sentences.

> You are a writer who is great at writing stories. You are provided with a few sentences to start with. Please continue the story from the provided sentences.
>
> Start of the story:
> One sunny afternoon, Lily discovered a tiny, golden key hidden under a rock in her backyard. Curious, she picked it up and wondered what it might unlock. Just then, her cat, Whiskers, began to meow excitedly, leading her towards the old oak tree at the edge of the garden.
>
> Lily followed Whiskers, her heart pounding with excitement and a touch of nervousness. The old oak tree had always seemed mysterious to her, its gnarled branches casting long shadows even on the brightest days. As they reached the tree, Whiskers darted ahead and started pawing at something buried beneath a pile of autumn leaves ...

Likewise, we can create a conversation given the conversational context.

> You are provided with a conversation between a tourist and a taxi driver. Please complete the conversation.
>
> Tourist: Could you take me to the downtown museum, please?
>
> Taxi Driver: Of course! First time in the city?
>
> Tourist: Yes, any must-see places around?
>
> Taxi Driver: Definitely check out the botanical gardens after the museum. They're nearby and quite beautiful.
>
> Tourist: Great, thanks for the tip!
>
> Taxi Driver: No problem. Enjoy your visit!

Another application of text completion is generating text based on user requirements. The requirements can include style, tone, length, and any specific content that the text should contain. For example, users may need a formal report, a creative story, or a piece of programming code. Here is a prompt for generating a five-character regulated Chinese poem.

> You are a poet. Please write a traditional Chinese poem. Here is the requirement.
>
> Theme: Spring's rejuvenation
>
> Structure: Five-character regulated poem
>
> Emotion: Convey a sense of happiness and renewal through vivid spring imagery.
>
> 春意盎然
>
> 翠柳风中舞，
> 红花雨后新。
> 山明水又绿，
> 天宽鸟自频。

If the LLM is trained to generate language and code, we can prompt it to perform code completion tasks. Here is an example.

Please write a Python function to calculate the average of a list of numbers.

```
def calculate_average(numbers):
    if numbers:    # Check if the list is not empty
    return sum(numbers) / len(numbers)
    else:
    return 0  # Return 0 if the list is empty
```

4. Text Transformation

Typical text transformation tasks include machine translation, summarization, text style transfer, and so on. The prompts used to perform these tasks are not significantly different from the ones presented previously. Here, we present a few examples to showcase these prompts.

Translate the following text from English to Spanish.

Text:

The quick brown fox jumps over the lazy dog.

El rápido zorro marrón salta sobre el perro perezoso.

Summarize the following article in no more than 50 words:

Article:

In recent years, urban areas have been facing significant challenges related to sustainability and environmental management. Cities are experiencing higher rates of pollution, increased traffic congestion, and greater demands on infrastructure. This growth has led to numerous environmental issues, including elevated levels of air and water pollution, increased waste production, and strained public services ...

Urban areas are grappling with sustainability challenges, such as rising pollution, traffic congestion, and infrastructure demands ...

Rewrite this text in a formal tone.

Text:

Hey, what's up? Long time no see!

Hello, how have you been? It has been quite some time since we last met!

5. Question Answering

The question-answering format is inherently simple. For a given question, there is an answer that corresponds to it. For example, in open-domain question answering, we expect the system to return an answer in response to a user-submitted question. Prompt templates for general-purpose question answering can be:

{∗question∗}

____

Question: {∗question∗}

Answer: ____

Q: {∗question∗}

A: ____

Question answering is important in NLP because many problems can be framed as question-answering tasks. In particular, many recent reasoning tasks are defined in the form of question answering. For example, in the MMLU benchmark [Hendrycks et al., 2021], each example consists of a multiple-choice question, and LLMs are required to select the correct answer. See the following for an example prompt for answering a question in this dataset.

$(Z, *)$ is a group with $a * b = a + b + 1$ for all a, b in Z. The inverse of a is

(A) 0

(B) -2

(C) $a - 2$

(D) $(2 + a) * -1$

D

Another widely-used benchmark is the GSM8K dataset [Cobbe et al., 2021]. It consists of thousands of grade school math word problems. For each problem, we use LLMs to return a solution in natural language. For example, the following is a prompt used for solving a problem in GSM8K.

DEMO Q: Jerry's two daughters play softball on different teams. They each have 8 games this season. Each team practices 4 hours for every game they play. If each game lasts for 2 hours, how many hours will Jerry spend at the field watching his daughters play and practice altogether?

A: Jerry will spend 8 games * 2 hours per game = $\ll 8*2=16 \gg 16$ hours watching one daughter play her games. He will spend $16*2=\ll 16*2=32 \gg 32$ hours watching both daughters play their games. He will spend 8 games * 4 hours of practice = $\ll 8*4=32 \gg 32$ hours watching one daughter practice. He will spend $32*2=\ll 32*2=64 \gg 64$ hours watching both daughters practice. He will spend a total of 32 hours watching games + 64 hours watching practice $=\ll 32+64=96 \gg 96$ hours. #### 96

Q: Mary bought six apples from the store. From the apples she bought, for each that Mary ate, she planted two trees from the remaining ones. How many apples did Mary eat?

A: She planted eight trees. This means she used half of that amount of apples, which is 8 trees $/2$ trees/apple $=\ll 8/2=4 \gg 4$ apples. That means that she planted four of the six apples she bought, leaving only 6 apples $-$ 4 apples $=\ll 6-4=2 \gg 2$ apples to be eaten. #### 2

USER Q: Boris has 100 apples. Beck has 23 fewer apples than Boris. If Boris gives Beck 10 apples, how many fewer apples does Beck have than Boris now?

A: Boris starts with 100 apples and gives Beck 10 apples, leaving him with 100 apples -10 apples $=\ll 100-10=90 \gg 90$ apples. Initially, Beck has 23 fewer apples than Boris, giving him a starting count of $100-23=\ll 100-23=77 \gg 77$ apples. After receiving 10 apples from Boris, Beck's total comes to 77 apples $+10$ apples $=\ll 77+10=87 \gg 87$ apples. Consequently, Beck now has $90-87=\ll 90-87=3 \gg 3$ fewer apples than Boris. #### 3

Here a few-shot prompt is adopted. The LLM learns from these demonstrations of problem-solution pairs not only the way of problem-solving but also the way of formatting the output. For example, the final result of calculation follows the #### token, and $\ll \ldots \gg$ annotates the detailed calculation steps (called **calculation annotation**)[4].

[4]During prediction, a calculator is used when we see $\ll \ldots \gg$. More specifically, once the LLM encounters "=" in a $\ll \ldots \gg$, then the calculator calculates the expression on the left-hand side of "=". This method helps reduce the calculation errors made by LLMs.

9.2 Advanced Prompting Methods

So far in this chapter, we have introduced the basic concepts related to LLM prompting and presented a number of prompts for NLP tasks. We now consider several techniques for enhancing the effectiveness of prompting.

9.2.1 Chain of Thought

We have encountered the concept of **chain of thought** (**CoT**) several times in this chapter and previous ones [Wei et al., 2022; Chowdhery et al., 2022]. CoT methods provide a simple way to prompt LLMs to generate step-by-step reasoning for complex problems, thereby approaching tasks in a more human-like manner. Rather than coming to a conclusion directly, the CoT methods instruct LLMs to generate reasoning steps or to learn from demonstrations of detailed reasoning processes provided in the prompts. To illustrate CoT, we consider the problem of algebraic calculation, as commonly described in the literature. Suppose we are given an algebraic problem

Calculate the average of the numbers 2, 4, and 6.

We can consider it as the question and prompt an LLM to answer it.

> Q: Please calculate the average of the numbers 2, 4, and 9.
>
> A: The answer is 6.

It seems difficult for the LLM to directly give a correct answer. A simple improvement is to add demonstrations of similar problems in the prompt, and thus the LLM can learn from these demonstrations.

> Q: Please calculate the average of the numbers 1, 3, 5, and 7.
>
> A: The answer is 4.
>
> Q: Please calculate the average of the numbers 2, 4, and 9.
>
> A: The answer is 7.

The problem here is that, although we have shown a similar question-answer pair, it remains difficult for the LLM to reason out the correct answer. In CoT, not only can LLMs learn from the correspondence between questions and answers but they may gain more from detailed problem-solving steps that are used to derive the answers. To do this, we can incorporate some reasoning steps into the prompt to obtain a CoT prompt.

Q: Please calculate the mean square of the numbers 1, 3, 5, and 7.

A: Calculate the square of each number: $1^2 = 1, 3^2 = 9, 5^2 = 25$, and $7^2 = 49$. Sum the squares, $1 + 9 + 25 + 49 = 84$. There are 4 numbers in total. Divide the sum by the number of items, $84/4 = 21$. The answer is 21.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Calculate $2 + 4 + 9$, which equals 15. There are three numbers. Divide the total sum by the count, resulting in $15/3 = 5$. The answer is 5.

Here we highlight the reasoning steps in green. By providing a detailed reasoning process for a similar problem, the LLM learns to reason and thereby generates a problem-solving path that leads to the correct answer.

There are several benefits of using CoT prompting. First, CoT allows LLMs to decompose complex problems into smaller, sequential reasoning steps. This somewhat mirrors human problem-solving behaviors, making it particularly effective for tasks requiring detailed, multi-step reasoning. Second, CoT makes the reasoning process more transparent and interpretable. Since all reasoning steps are visible, we can understand and interpret how a conclusion was reached. Third, if users can follow the logic behind the reasoning process, they will be more likely to trust the predictions of an LLM. This is particularly important when applying LLMs in fields like medicine, education, and finance. Fourth, CoT is an in-context learning approach, and thus, it is applicable to most well-trained, off-the-shelf LLMs. Moreover, CoT provides efficient ways to adapt LLMs to different types of problems. It can even inspire more creative solutions by exploring various alternative reasoning paths, which might not be obvious when arriving at a conclusion directly.

The method described above requires providing one or more examples of CoT reasoning, typically called the few-shot CoT method. By contrast, the zero-shot CoT method does not require such examples. It instead prompts LLMs to reason step-by-step by incorporating specific instructions in prompts. For example, below is a zero-shot CoT prompt.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Let's think step-by-step.

We have three numbers: 2, 4, and 9. Add these numbers together, $2 + 4 + 9 = 15$. Determine how many numbers there are, which in this case is three. The average is calculated by dividing the total sum by the number of elements. Completing the division gives $15/3 = 5$. So the answer is 5.

Following the instruction "Let's think step by step", the LLM is prompted to generate detailed reasoning steps. As discussed in Kojima et al. [2022]'s work, prompting with such instructions may result in LLMs generating only the reasoning steps without a clear conclusion. In this case, a second round of prompting can be used to extract the answer from these reasoning

steps. For example, Kojima et al. [2022] create a second prompt which combines both the input and output in the first round of prompting. Using this combined input, the LLM can continue its reasoning process and then generate the correct answer. Furthermore, it is possible to prompt LLMs to reason using instructions other than "Let's think step by step", such as "Let's think logically" and "Please show me your thinking steps first".

While we have illustrated CoT methods using an algebraic reasoning problem, these methods can be applied to a variety of different problems. Typical problem-solving scenarios for CoT include mathematical reasoning, logical reasoning, commonsense reasoning, symbolic reasoning, code generation, and so on. See Figure 9.1 for more examples of applying CoT in various tasks.

CoT today is one of the most active fields of prompt engineering. This has not only led to improved performance for LLM prompting but has opened the door to a wide range of methods for studying and verifying reasoning capabilities of LLMs. Although we have focused on the basic idea of CoT in this section, it can be improved in several ways. For example, we can consider the reasoning process as a problem of searching through many possible paths, each of which may consist of multiple intermediate states (i.e., reasoning steps). In general, we wish the search space to be well-defined and sufficiently large, so that we are more likely to find the optimal result. For this reason, an area of current LLM research is aimed at designing better structures for representing reasoning processes, allowing LLMs to tackle more complex reasoning challenges. These structures include tree-based structures [Yao et al., 2024], graph-based structures [Besta et al., 2024], and so on. By using these compact representations of reasoning paths, LLMs can explore a wider range of decision-making paths, analogous to System 2 thinking[5]. Another line of research focuses on prompting LLMs with multi-round interactions. This involves decomposing complex problems into sub-problems, verifying and refining model outputs, employing model ensembling, and so on. Note that these methods and the issues involved are not limited to CoT. In fact, they are often used as more general approaches to improving LLMs, while CoT can be seen as a way to test the capabilities of LLMs. We will see discussions of some of these issues in the following subsections.

Before leaving our discussion of CoT, we should consider its practical limitations. One of them is the need for detailed, multi-step reasoning demonstrations in few-shot CoT scenarios, which may be difficult to obtain, either automatically or manually. Also, there is no standard method for breaking down complex problems into simpler problem-solving steps. This often heavily depends on the user's experience. In addition, errors in intermediate steps can also affect the accuracy of the final conclusion. For further discussion on the pros and cons of CoT, the interested reader can refer to recent surveys on this topic [Chu et al., 2023; Yu et al., 2023; Zhang et al., 2023a].

[5] System 1 and System 2 thinking, as described by Kahneman [2011], represent two different modes of cognitive processing. System 1 is fast, automatic, intuitive, and emotional. This mode of thinking operates effortlessly and quickly, and is often what guides our daily decisions, judgments, and impressions. System 2 is slow, deliberate, and analytical. It is activated when we need to perform complex computations.

Q: What home entertainment equipment requires cable?
Answer Choices: (a) radio shack (b) substation (c) television (d) cabinet
A: The answer must require cable. Of the above choices, only television requires cable. So the answer is (c).

Q: What type of person typically contracts illness?
Answer Choices: (a) hospital (b) head (c) sick person (d) elderly person (e) doctor's office
A: The answer must be a person susceptible to illness. Of the choices, "elderly person" is correct because they typically have weaker immune systems.
So the answer is (d).

(a) CSQA

Q: Do hamsters provide food for any animals?
A: Hamsters are prey animals. Prey are food for predators. Thus, hamsters provide food for some animals. So the answer is yes.

Q: Can you use Microsoft Office without internet?
A: Yes, you can use Microsoft Office applications , although features like cloud storage and online collaboration require internet. So the answer is yes.

(b) StrategyQA

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: [{
A: Let's think step by step.
0: empty stack
1: [; stack: [
2: { ; stack: [{
So the answer is }].

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < [[
A: Let's think step by step.
0: empty stack
1: < ; stack: <
2: [; stack: < [
3: [; stack: < [[
So the answer is]] >.

(c) Dyck languages

Q: Take the last letters of the words in "Elon Musk" and concatenate them.
A: The last letter of "Elon" is "n". The last letter of "Musk" is "k". Concatenating them is "nk". The answer is nk.

Q: Take the last letters of each word in "Renata Mariela Mona Kristin" and concatenate them.
A: The last letter of "Renata" is "a". The last letter of "Mariela" is "a". The last letter of "Mona" is "a". The last letter of "Kristin" is "n". Concatenating them is "aaan". The answer is aaan.

(d) Last Letter Concatenation

Figure 9.1: CoT in four different reasoning tasks, including CSQA, StrategyQA, Dyck languages, and Last Letter Concatenation. The CoT parts are highlighted in green.

9.2.2 Problem Decomposition

We have seen that LLMs can benefit from solving a complex problem by breaking it down into simpler problem-solving tasks. Such an approach can be seen as an example of a broader paradigm known as **problem decomposition**, which has been extensively explored and dis-

cussed in psychology and computer science. From the psychological perspective, complex problem-solving refers to a process of addressing a problem using knowledge that helps overcome the barriers of the problem[6]. There are generally no standard or clear paths to a solution for a complex problem. However, it is often advantageous to employ strategies that decompose the problem, thereby making it easier to tackle the corresponding sub-problems with less effort. For example, consider writing a blog about the risks of AI. If we simply prompt an LLM with the instruction "Please write a blog about the risks of AI", the LLM may generate a blog with arbitrary structures and writing styles. A better method, instead, could be to outline the blog and provide more detailed information about each section. Consider the following prompt

> You are a blog writer. Please follow the provided outline below to write a blog about the risks of AI.
>
> - Introduction
>   Introduce AI, its relevance, and the importance of understanding its risks for youth.
> - Privacy Concerns
>   Discuss how AI might compromise personal privacy through interactions online.
> - Misinformation
>   Explore AI's role in spreading misinformation and influencing young people's decisions.
> - Cyberbullying
>   Highlight how AI tools can be utilized in cyberbullying and the impact on mental health.
> - Tips for Safe AI Use
>   Offer guidelines for responsible AI usage and promote critical thinking.
> - Conclusion
>   Recap main points and encourage proactive engagement with AI ethics.
>
> ——

Here we give the title and major points for each section. Then, the LLM can use this structure to break down the writing task by filling in content for these sections. Note that the way to structure the blog can be provided by humans or even generated automatically. For example, we can use the LLM to first generate the outline, and then ask it to follow this outline to complete the writing.

In computer science, decomposing complex problems is a commonly used strategy in software and hardware system design. A well-known example is the divide-and-conquer paradigm, which is often used to design algorithms for computation problems that can be reduced to simpler, more manageable problems. For example, consider a problem of determining whether

[6] A relatively formal definition can be found in Frensch and Funke [2014]'s book: *complex problem-solving occurs to overcome barriers between a given state and a desired goal state by means of behavioral and/or cognitive, multi-step activities*.

a document discusses the risks of AI. We can instruct the LLM with the following prompt.

> You are provided with a text. Please determine whether it discusses the risks of AI.
>
> {∗document∗}
>
> ____

If the document is long, the computation will be expensive. Alternatively, we can divide the document into relatively short segments and perform the same task on each segment. These segments can be processed in parallel to further reduce the computational cost. Next, we determine the relevancy of each segment to the topic of AI risks. The final output is then generated using another prompt.

> Your task is to determine whether a text discusses the risks of AI. This text has been divided into segments, and you have obtained the relevancy of each segment to the topic of AI risks. Based on this, please provide your final result.
>
> Segment 1: {∗relevancy-to-the-topic1∗}
>
> Segment 2: {∗relevancy-to-the-topic2∗}
>
> Segment 3: {∗relevancy-to-the-topic3∗}
>
> ...
>
> ____

Now let us return to a more general discussion of problem decomposition in prompting. While problem decomposition can be applied to various NLP problems, it has been more extensively discussed and tested in reasoning tasks recently. For complex reasoning tasks, we often need a multi-step reasoning path to reach a correct conclusion. We can use LLMs to achieve this in three different ways. First, LLMs can directly reach the conclusion. In other words, they can predict without explicit reasoning processes, and there is a hidden and uninterpretable reasoning mechanism. Second, LLMs are prompted to generate a multi-step reasoning path that leads to the conclusion, like CoT. However, we run LLMs just once, and all intermediate steps in reasoning are generated in a single prediction. Third, we break down the original problem into a number of sub-problems, which are either addressed in separate runs of LLMs or tackled using other systems. Here we focus our attention on the third approach, which is closely related to problem decomposition. Note, however, that a more comprehensive discussion could cover all these approaches, while the first two have been discussed to some extent in this chapter.

A general framework for problem decomposition involves two elements.

- **Sub-problem Generation**. This involves decomposing the input problem into a number of sub-problems.
- **Sub-problem Solving**. This involves solving each sub-problem and deriving intermediate and final conclusions through reasoning.

These two issues can be modeled in different ways, leading to various problem decomposition methods. One approach is to treat them as separate steps in a two-step process. For example, consider the blog writing task described at the beginning of this subsection. In the first step, we decompose the entire problem into sub-problems all at once (i.e., outline the blog). In the second step, we solve the sub-problems either sequentially or in another order (i.e., fill in content for each section as needed). The final output of this process combines the results from solving each sub-problem. While this method is simple and straightforward, it assumes that the problem is compositional, making it more suitable for tasks like writing and code generation.

However, many real-world problems require complex reasoning. One key characteristic of these problems is that the reasoning steps may not be fixed. The reasoning path can vary for different problems, and each step of reasoning may depend on the outcomes of prior steps. In such cases, it is undesirable to use fixed sub-problem generation in advance. Instead, sub-problems should be generated dynamically based on the input problem, and, if possible, generated on the fly during the reasoning process. This makes problem decomposition more challenging compared with designing divide-and-conquer algorithms. Ideally, we would like to jointly design both the systems for sub-problem generation and sub-problem solving. But a more practical and widely used approach is to adopt separate models for these tasks. A straightforward way to achieve this is to adapt an LLM for these tasks by either prompting or tuning the model.

Here we consider a method based on the above idea, called **least-to-most prompting** [Zhou et al., 2023a]. The motivation for this method arises from the challenges of solving difficult reasoning problems — those that cannot be addressed by simply generalizing from a few examples. For these problems, a more effective problem-solving strategy is to follow a progressive sequence of sub-problems that systematically lead to the conclusion. More specifically, in the least-to-most prompting method, sub-problem generation is performed by prompting an LLM with instructions and/or demonstrations. For example, below is a 2-shot prompt for sub-problem generation in least-to-most prompting.

TASK Your task is to decompose a problem into several sub-problems. You will be given a few examples to illustrate how to achieve this.

DEMO Q: In a community, 5% of the population are infants, 15% are children, 40% are adults, and 40% are seniors. Which group makes up the largest portion of the population?

A: To answer the question "Which group makes up the largest portion of the population?", we need to know: "How many percent are infants?", "How many percent are children?", "How many percent are adults?", "How many percent are seniors?".

Q: Alice, Bob, and Charlie brought beads for their group project in their craft class. Alice has twice as many beads as Bob, and Bob has five times as many beads as Charlie. If Charlie has 6 beads, how many beads can they use for their craft project?

A: To answer the question "How many beads can they use for their craft project?", we need to know: "How many beads does Bob have?", "How many beads does Alice have?".

USER Q: The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius. What was the duration of the environmental study?

A: To answer the question "What was the duration of the environmental study?", we need to know: "When did the environmental study start?", "When did the environmental study end?".

By learning from the examples, the LLM can generate two sub-problems for answering the new problem "What was the duration of the environmental study?" (highlighted in blue and orange). Given these sub-problems, we solve them sequentially. For each sub-problem, we take all previously-generated QA pairs as context, and then produce the answer. For the example above, we need to answer the first sub-problem by prompting the LLM, like this

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

Once we have the answer to the first sub-problem, we proceed to the second one. This time, we include both the first sub-problem and its corresponding answer in the input.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

Finally, we use the LLM to solve the original problem given the answers to all the sub-problems.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

FINAL Q: What was the duration of the environmental study?

A: The duration of the environmental study was 5 years.

The least-to-most method offers a basic approach to prompting LLMs to generate and solve sub-problems separately. We can improve it in several ways. One simple improvement is to apply various advanced prompting techniques, which do not require changes to the problem decomposition framework. For example, we can incorporate CoT into the prompting to enhance the reasoning performance of sub-problem generation and solving.

Another improvement is to explore methods for better decomposing problems and organizing problem-solving paths. To describe these approaches, we will use the symbol p_0 to denote the input problem, and use the symbols $\{p_1, ..., p_n\}$ to denote the sub-problems corresponding to p_0. For least-to-most prompting, we decompose p_0 into $\{p_1, ..., p_n\}$, given by

$$\{p_1, ..., p_n\} \quad = \quad G(p_0) \tag{9.2}$$

where $G(\cdot)$ denotes the function of sub-problem generation. Then, we solve the sub-problems $\{p_1, ..., p_n\}$ sequentially, resulting in a sequence of answers $\{a_1, ..., a_n\}$. For answering the i-th sub-problem p_i, we include both the original problem p_0 and all previously-seen problem-

answer pairs in the context for prediction. The answer a_i is given by

$$a_i = S_i(p_i, \{p_0, p_{<i}, a_{<i}\}) \tag{9.3}$$

where $p_{<i} = \{p_1, ..., p_{i-1}\}$ and $a_{<i} = \{a_1, ..., a_{i-1}\}$. $S_i(\cdot)$ denotes the function that solves the sub-problem p_i given the context $\{p_0, p_{<i}, a_{<i}\}$. The last step is to generate the answer to the original problem p_0, which can be expressed in a similar manner to Eq. (9.3).

$$a_0 = S_0(p_0, \{p_{\leq n}, a_{\leq n}\}) \tag{9.4}$$

One way to refine this model is to modify the $G(\cdot)$ function so that the model can dynamically generate answers. Instead of generating all sub-problems at one time, we can generate each of them during problem-solving [Dua et al., 2022]. To do this, we can replace Eq. (9.2) with

$$p_i = G_i(p_0, \{p_{<i}, a_{<i}\}) \tag{9.5}$$

Hence we obtain a sub-problem generation model that operates in a step-by-step manner. At each step i, we first generate the sub-problem p_i by prompting an LLM with the original problem p_0 and the problem-solving history $\{p_{<i}, a_{<i}\}$. We then generate the answer a_i for this sub-problem using the same or a different LLM, based on the same contextual information (see Eq. (9.3)). This method effectively expands the reasoning capacity of LLMs by allowing them to dynamically generate and solve sub-problems in intermediate reasoning steps. As a result, the reasoning paths are not fixed in advance, and the models can choose and adapt their reasoning strategies during problem-solving.

Another way to improve the above model is to focus on developing better sub-problem solvers. In our previous discussion, we restricted $S_i(\cdot)$ to LLMs that are prompted to solve the sub-problem p_i. In fact, we can expand this function to any system that is capable of addressing the sub-problem. For example, $S_i(\cdot)$ could make calls to IR systems, thereby allowing us to access a broader range of data for problem-solving. Another example is using $S_i(\cdot)$ as a calculator to accurately compute results in mathematical problem-solving. If the sub-problem p_i is complex and requires multiple intermediate problem-solving steps, it is also possible to further decompose p_i into smaller sub-problems. For example, $S_i(\cdot)$ can be defined as a recursive program that generates and solves sub-problems. This incorporates recursion into problem-solving and allows us to address problems by iteratively decomposing them. As a result, we can define a hierarchical structure for problem-solving [Khot et al., 2023].

If we generalize the above formulation a bit further, we can consider it as a reinforcement learning problem. A typical method is to model a problem-solving process as a decision making process. In each step of this process, an action is taken based on the current state. These actions can include all functions for sub-problem generation and solving (i.e., $G_i(\cdot)$ and $S_i(\cdot)$). Thus, the action sequence corresponds to a problem-solving path. Since the discussion of reinforcement learning problems is beyond the scope of this chapter, we skip the precise description of this learning task. Nevertheless, developing an agent or controller to determine

when and how to generate and solve a sub-problem is also a natural choice.

In NLP, problem decomposition is related to a long line of research on multi-hop question answering [Mavi et al., 2024]. This task requires the system to gather and combine information from multiple pieces of text to provide an accurate answer to a complex question. For example, to answer the question "What is the capital of the country where Albert Einstein was born?", we need to know "Where Albert Einstein was born?" and "What's the capital of Germany?". Earlier work in this area and related ones has investigated the issue of problem decomposition, though the methods might not be based on LLMs. For example, a popular method is to develop an additional neural model to generate simpler questions that address different aspects of the original question [Andreas et al., 2016; Talmor and Berant, 2018; Min et al., 2019]. This question generator can create questions in a batch or sequential manner.

Broadly speaking, problem decomposition is also related to the compositionality issue in NLP [Drozdov et al., 2022; Press et al., 2023]. For example, in semantic parsing, we map natural language sentences into structured meaning representations by breaking them down into constituent parts and understanding the sentences based on the meanings of these parts and the rules used to combine them. In early studies of this field, highly compositional sentences were considered easier for testing systems, as it is relatively straightforward to decompose such sentences and compose the meanings of their parts. However, the task becomes much more difficult when more generalization is required for modeling compositionality in new data. In this case, we want systems to have improved abilities of **compositional generalization**. In more recent research on LLMs, this issue has been frequently discussed in compositional reasoning tasks, such as SCAN[7], as it is considered an important aspect of testing the language understanding and reasoning abilities of LLMs. This also presents new tasks for developing and examining problem decomposition methods.

In LLMs, one interesting application of problem decomposition is tool use. In some cases, it is necessary to integrate external tools into LLMs to access accurate data not available during training or fine-tuning. For example, LLMs can integrate with APIs to fetch real-time data such as weather updates, stock market prices, or news feeds, enabling them to provide up-to-date responses to user queries. When using tools, LLM predictions might include markers that indicate where and how to call external APIs. This requires decomposing the problem into sub-problems, with some handled by the LLMs and others by external tools. More detailed discussions on this topic will be presented in Section 9.2.5.

9.2.3 Self-refinement

In many cases, predictions of LLMs can be inaccurate or incorrect. Given that current LLMs can perform tasks like refinement and correction, it makes sense to explore methods for these models to self-refine their outputs. Self-refinement is a common phenomenon in human psychological activities and daily behavior. For example, when designing a product, a designer

[7]The SCAN tasks (Simplified versions of the CommAI Navigation tasks) are designed to evaluate the ability of LLMs to perform compositional generalization [Lake and Baroni, 2018]. They involve translating natural language commands into a sequence of actions. For example, a command "jump opposite left and walk thrice" can be translated into the action sequence "LTURN LTURN JUMP WALK WALK WALK".

might first create a basic prototype, then refine the design after evaluation and testing to enhance user experience and functionality. The refinement can be iterated several times until the design is satisfactory. The idea of predict-then-refine can also be found in NLP. One early example is Brill's tagger [Brill, 1992], where an initial POS tagging result of a sentence can be iteratively refined using a rule-based system. In the era of deep learning, a good deal of work on sequence-to-sequence problems, such as grammar correction and text rewriting, can also be seen as examples on this theme.

We can prompt LLMs to do self-refinement. Consider a simple example where we first use an LLM to translate a text from Chinese to English and then use the same model to refine the translation. Here is how we can prompt the model to translate:

> Translate the following text from Chinese to English:
>
> Chinese:
> 一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今5800年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。
>
> English:
> A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggest that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Then, we can prompt it to refine the translation:

You are provided with a Chinese text along with its English translation. Please refine the translation to improve its accuracy and fluency if necessary.

Chinese:
一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今5800年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。

English:
A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggests that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Refinement:
A series of archaeological discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggest that, starting from around 5800 years ago, various regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Here modifications to the input translation are highlighted in green. This process can be iterative: at each step, we use the translation from the previous step as input, and prompt the LLM to produce a potentially better translation.

In the above example, the refinement of the initial model output was achieved by prompting the LLM with a simple instruction like "Please refine it!". However, the refinement is based solely on the instruction-following ability of the LLM, and there is no guidance or supervision on how and where to improve the model output. A more effective approach would be to consider feedback on specific aspects that need refinement. For example, we can prompt the LLM with "Please correct all the grammatical errors in the translation", so that the model can focus more on grammatical error correction during refinement.

A general framework of self-refinement with LLMs involves three steps [Madaan et al., 2024].

- **Prediction**. We use an LLM to produce the initial model output.
- **Feedback Collection**. We obtain feedback on the model output.
- **Refinement**. We use the LLM to refine the model output based on the feedback.

The last two steps can be repeated multiple times, which leads to an iterative self-refinement process. In this framework, receiving accurate and detailed feedback is critical as it guides the subsequent refinement step. In general, feedback can be obtained in various ways [Fernandes et al., 2023; Pan et al., 2024]. We can manually examine the model's predictions to identify

issues. Also, we can use a feedback model to automatically generate feedback. For instance, a reward model trained on some labeled data can help evaluate the model output and return a score indicating its quality.

Since the focus of this chapter is on LLM prompting, we consider using LLMs to provide feedback. To illustrate, let us consider an example of generating good responses to user questions:

- First, we prompt an LLM to answer the input question, as usual.

> Generate a response to the question: "What are some major environmental concerns today?"
>
> Response:
>
> Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

- Then, using the following prompt, we prompt the same LLM again to get feedback on the generated response.

> You will be provided with a question and its corresponding response. Evaluate the accuracy and depth of this response, and identify any factual inaccuracies or areas where additional details could enhance understanding.
>
> Question:
>
> What are some major environmental concerns today?
>
> Response:
>
> Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.
>
> Feedback:
>
> The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:
>
> 1. Include specific examples of environmental issues.
> 2. Add information about major pollutants.
> 3. Specify the regions most affected.

- Given the feedback the LLM produces, we prompt it to refine the response.

You will be provided with a question, its corresponding response, and some feedback on this response.

Question:

What are some major environmental concerns today?

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

Feedback:

The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:

1. Include specific examples of environmental issues.
2. Add information about major pollutants.
3. Specify the regions most affected.

Now refine the response based on the provided feedback.

Refined Response:

Key environmental concerns include climate change, evidenced by rising global temperatures and severe weather patterns; significant air and water pollution in urban and industrial areas; rampant deforestation in regions like the Amazon, causing biodiversity loss; and ocean degradation, highlighted by coral reef bleaching and widespread overfishing.

Ideally, if a strong LLM is adopted, we would like to have it perform all three steps without extra training. On the other hand, if we have enough labeled data for the task of interest, we can enhance the performance of the LLM using supervised learning. For example, we can fine-tune the LLM to better adapt it to refinement tasks, or alternatively, use task-specific models, which may not necessarily be based on LLMs [Welleck et al., 2023; Schick et al., 2023]. In a broader sense, improving LLMs for self-refinement tasks can be seen as an alignment issue. For example, it has been found that some self-correction abilities can be activated through RLHF [Ganguli et al., 2023]. However, discussing these issues is beyond the scope of this chapter. Further discussion can be found in Chapter 10.

In LLMs, self-refinement is related to several concepts that reveal the psychological aspects of these models, such as the ability to self-reflect. A view is that if LLMs are capable of self-reflection, their predictions can become more accurate and even possess self-correcting capabilities. This self-reflection can be activated in various ways, for example, by prompting these LLMs to engage in more in-depth and careful thinking, or by providing examples from

which the models can learn and reflect. To illustrate, we consider here the **deliberate-then-generate** (**DTG**) method presented in Li et al. [2023a]'s work, where LLMs are prompted to deliberate. In DTG, we are given an initial model output which may contain errors. LLMs are then prompted to identify the error types of this model output and provide an improved output. Below is a template of DTG prompting for Chinese-to-English translation tasks.

> Given the Chinese sentence: {∗source∗}
>
> The English translation is: {∗target∗}
>
> Please first detect the type of error, and then refine the translation.
>
> Error Type:
>
>

We aim to first predict the error type (red), and then produce a refined translation (blue). This process of deliberation is guided by the instruction "Please first detect the type of error, and then refine the translation". It encourages LLMs to initially engage in thoughtful analysis and then give better results. Since error type prediction and refinement are performed in a single run of LLMs, this method incorporates both steps of feedback and refinement into one process.

In the above prompts, we assume that the LLM we use is able to review the input translation and correctly identify its error types. However, this raises new difficulties as the model may not be good at finding errors in translations. This will in turn result in extra fine-tuning or prompting engineering efforts. So a simpler method is to reduce the burden of error identification and use LLMs for deliberation only. To do this, we can replace the input translation with a random translation and assign a default error type. An example of such a prompt is shown below.

> Given the Chinese sentence:
>
> 一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。
>
> The English translation is:
>
> A variety of innovative techniques have redefined the importance of modern art in contemporary cultural studies.
>
> Please first detect the type of error, and then refine the translation.
>
> Error Type: Incorrect Translation
>
>

In this example, the input translation is not generated by LLMs but is instead randomly sampled from the dataset. So it is simply an incorrect translation for the source sentence, and we can set the error type accordingly. The LLMs then generate a new translation by taking both the source sentence and the incorrect translation as input. The design of this prompt

can also be considered as activating the learning capabilities of LLMs through "negative evidence" [Marcus, 1993], thereby enabling them to reflect and produce better outcomes through contrastive analysis. Nevertheless, this method does not rely on any feedback and can enhance the performance of a single LLM prediction via simple prompting.

Note that while DTG is non-iterative, iterative learning and refinement are commonly used in NLP. An advantage of these iterative approaches is that they mimic human learning and problem-solving, where continuous feedback and adjustments lead to progressively improved outcomes. Iterative methods can be applied to a range of LLM prompting problems. For example, in problem decomposition, one can incorporate new sub-problems and their solutions into the context at each step, and thus LLMs can progressively approach the solution of the original problem. On the other hand, iterative methods raise several issues that are absent in non-iterative methods, for example, errors in earlier steps may negatively impact subsequent problem-solving, and determining when to stop iterating often requires additional engineering effort.

9.2.4 Ensembling

Model ensembling for text generation has been extensively discussed in the NLP literature. The idea is to combine the predictions of two or more models to generate a better prediction. This technique can be directly applicable to LLMs. For example, we can collect a set of LLMs and run each of them on the same input. The final output is a combined prediction from these models.

For LLM prompting, it is also possible to improve performance by combining predictions based on different prompts. Suppose we have an LLM and a collection of prompts that address the same task. We can run this LLM with each of the prompts and then combine the predictions. For example, below are three different prompt templates for text simplification.

> Make this text simpler.
> {∗text∗}
> ——

> Condense and simplify this text.
> {∗text∗}
> ——

> Rewrite for easy reading.
> {∗text∗}
> ——

Each of these prompts will lead to a different prediction, and we can consider all three predictions to generate the final one.

Formally, let $\{\mathbf{x}_1, ..., \mathbf{x}_K\}$ be K prompts for performing the same task. Given an LLM $\Pr(\cdot|\cdot)$, we can find the best prediction for each $\mathbf{x}_i$ using $\hat{\mathbf{y}}_i = \arg\max_{\mathbf{y}_i} \Pr(\mathbf{y}_i|\mathbf{x}_i)$. These predictions can be combined to form a "new" prediction:

$$\hat{\mathbf{y}} = \text{Combine}(\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_K) \tag{9.6}$$

Here $\text{Combine}(\cdot)$ is the combination model, which can be designed in several different ways. For example, we can select the best prediction by voting or by identifying the one that overlaps the most with others. Another method for model combination is to perform model averaging during token prediction. Let $\hat{y}_j$ be the predicted token at the j-th step for model combination. The probability of predicting $\hat{y}_j$ is given by

$$\hat{y}_j = \arg\max_{y_j} \sum_{k=1}^{K} \log \Pr(y_j|\mathbf{x}_k, \hat{y}_1, ..., \hat{y}_{j-1}) \tag{9.7}$$

The interested reader can refer to Chapter 5 for more details of these methods.

In ensembling for LLM prompting, it is generally advantageous to use diverse prompts so that the combination can capture a broader range of potential responses. This practice is common in ensemble learning, as diversity helps average out biases and errors that may be specific to any single model or configuration. From the Bayesian viewpoint, we can treat the prompt $\mathbf{x}$ as a latent variable, given the problem of interest, p. This allows the predictive distribution of $\mathbf{y}$ given p to be written as the distribution $\Pr(\mathbf{y}|\mathbf{x})$ marginalized over all possible prompts

$$\Pr(\mathbf{y}|p) = \int \Pr(\mathbf{y}|\mathbf{x}) \Pr(\mathbf{x}|p) d\mathbf{x} \tag{9.8}$$

The integral computes the total probability of $\mathbf{y}$ by considering all possible values of $\mathbf{x}$, weighted by their likelihoods given p. Here $\Pr(\mathbf{y}|\mathbf{x})$ is given by the LLM, and $\Pr(\mathbf{x}|p)$ is the prior distribution of prompts for the problem. This is a good model because the integral effectively accounts for the uncertainty in the choice of $\mathbf{x}$, ensuring that the final predictive distribution $\Pr(\mathbf{y}|p)$ is robust and encompasses all potential variations and biases in the prompts. However, computing this integral directly can be computationally infeasible due to the potentially infinite space of $\mathbf{x}$. One approach to addressing this issue is to employ methods like Monte Carlo sampling, which approximate the integral using a manageable, finite number of prompts.

While the Bayesian treatment is mathematically well-defined, it is common practice in NLP to assume a non-informative or uniform prior and focus instead on constructing a set of diverse prompts. Consequently, the output can be computed using a straightforward combination model, as described in Eq. (9.6). The issue of creating high-quality, diverse prompts has been studied in CoT and other in-context learning areas. Most of the research focuses on

incorporating a variety of demonstration examples across different prompts. Here, we list some of these methods.

- Given a problem, we manually create a number of demonstrations and use different ones for different prompts.
- Given a problem, we use LLMs to automatically generate demonstrations and prompts.
- Given a prompt, we create different prompts by changing the order of demonstrations in the prompt.
- Given a prompt, we use LLMs to generate a number of similar prompts.
- Given a prompt, we transform it into other forms, e.g., translating it into other languages.

Of course, in practice, we can combine these methods to achieve greater diversity. An underlying assumption here is that diverse prompts can lead to diverse model outputs. This is particularly the case when the problem we deal with is relatively new and difficult. For stronger and more robust LLMs, the variance in output for similar prompts might not be large. In this case, the benefit of involving multiple prompts can be modest.

In addition to providing diverse prompts for LLMs, another approach is to make use of the inherent variance in the outputs of LLMs. One simple way to generate multiple outputs is to sample outputs from the hypothesis space. This is straightforward for LLMs because their search algorithms are naturally suited to producing multiple results. For example, if beam search is used, we can easily collect all the complete hypotheses in the beam to generate multiple output candidates. Also, we can modify the search algorithms to implement more powerful sampling methods and explore a wider range of high-quality hypotheses.

The next question is how to combine diverse model predictions. This leads us back to the model combination issue as described in Eq. (9.6). Here we consider the **self-consistency** method, which outputs not the prediction with the highest probability, but rather the one that best aligns with other predictions [Wang et al., 2022; 2023]. First, an LLM is prompted with CoT as usual and generates multiple reasoning paths by sampling. Then, the frequency of each answer is counted across these reasoning paths. The final output is the answer that has the highest count. Below shows three different predictions by prompting an LLM with CoT.

Prediction 1 (Correct):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Each coin flip is an independent event with a 50% chance of landing heads (H) and 50% tails (T). Consider the possible outcomes for three flips: HHH, HHT, HTH, HTT, THH, THT, TTH, TTT. Out of these, only HHH, HHT, HTH, and THH are relevant. We are only interested in scenarios with exactly one head: HTT, THT, and TTH. There are 3 such outcomes out of 8 total, giving a probability of $3/8$.

Prediction 2 (Correct):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Using a binomial probability formula, where p (success or heads) is 0.5, and each of the 3 trials (flips) is independent, we look for exactly one success (head). The formula for exactly k successes in n trials is given by the binomial coefficient formula $\binom{n}{k}p^k(1-p)^{n-k}$. Plugging in the values, we get $\binom{3}{1}(0.5)^1(0.5)^{3-1} = 3 \cdot 0.5 \cdot 0.25 = 0.375$.

Prediction 3 (Wrong):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Since each coin flip has a 50% chance of landing heads and a 50% chance of landing tails, and we are looking for the case where only one of the three flips results in heads, we might initially think this is simply a matter of picking which flip is the heads. There are three flips, so one might incorrectly assume that the chance of picking one specific outcome like this would be 1 out of 3. Thus, they might conclude that the probability of exactly one head is $1/3 = 33.3\%$.

Predictions 1 and 2 correctly identify the three cases where exactly one head is flipped, both obtaining a probability of 37.5%. The reasoning in Prediction 3 fails to account for the total number of outcomes possible with three coin flips, thus giving a wrong answer of 33.3%. Therefore, we select 37.5% as the final answer because it is the consensus.

Self-consistency provides a criterion for determining the best prediction in a pool of candidates. Since the prompt and the model are fixed in this method, it is not strictly a prompt ensembling method. Instead, it can be seen as an instance of output ensembling methods, also known as hypothesis selection methods, which have long been explored in NLP, particularly for text generation problems [Xiao et al., 2013]. In these methods, multiple outputs are generated by varying model architectures or parameters. Each output is then assigned a score by some criterion, and the outputs are re-ranked based on these scores. There are various ways to define the scoring function, such as measuring the agreement between an output and others, and using a stronger model to rescore each output[8]. Figure 9.2 shows a comparison of different

[8] An interpretation of self-consistency is to view it as a minimum Bayes risk search process. It searches for the best output by minimizing the Bayes risk. More specifically, a risk function $R(\mathbf{y}, \mathbf{y}_r)$ is defined on each pair of outputs (denoted by $(\mathbf{y}, \mathbf{y}_r)$), representing the cost of replacing $\mathbf{y}$ with $\mathbf{y}_r$. Given a set of outputs Ω, the risk of an

ensembling methods for LLMs.

Now, let us briefly review the methods we have discussed so far in this section, such as problem decomposition and self-refinement. It is apparent that these methods enhance decision-making by introducing more "choices" into the reasoning process. To some extent, they all involve evaluating and providing feedback on the results of LLMs. For example, in self-refinement, we need to offer suggestions for improving the prediction of LLMs, and in output ensembling, we select the optimal output from a pool of candidates. In this sense, these methods fall under the broader category of predict-then-verify approaches, where predictions are initially made, then verified and refined. The fundamental problem here involves verifying and evaluating the reasoning results or intermediate steps. This issue is somewhat related to the problem of training reward models in RLHF, although RLHF addresses a different aspect. In fact, the development of verifiers has been explored and implemented in reasoning with LLMs. Most work, rather than developing heuristic-based inference-time algorithms, focuses on learning verifiers in a supervised manner. A straightforward method is to train verifiers as binary classifiers, such as classifying an answer as correct or incorrect, although these verifiers are typically used as scoring models. Given a reasoning path for a problem, the verifiers can be used to score either the entire path (called outcome-based approaches) [Cobbe et al., 2021], or each individual reasoning step (called process-based approaches) [Uesato et al., 2022; Lightman et al., 2024].

9.2.5 RAG and Tool Use

RAG is generally employed when standard LLMs, which rely solely on pre-trained knowledge, lack accuracy and depth in the generated text. By drawing from external databases and documents, RAG can significantly improve the quality of responses, ensuring they are both contextually relevant and factually correct. Such an approach is particularly useful in scenarios that require high factual accuracy and up-to-date information, such as complex question answering.

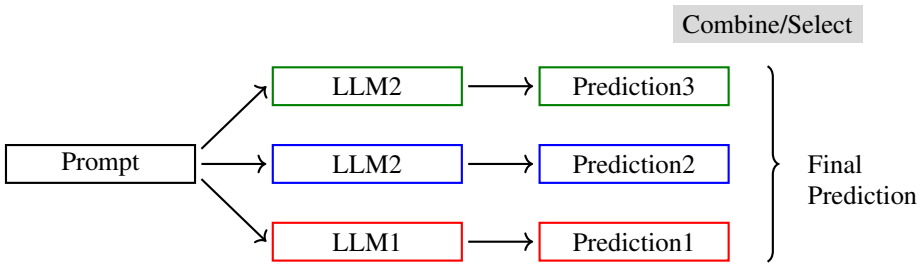
The concept of RAG has been mentioned several times in the previous sections and chapters. For completeness, we outline the key steps involved in RAG here.

- We prepare a collection of texts which are treated as an additional source of knowledge we can access.
- We retrieve relevant texts for a given query.
- We input both the retrieved texts and the query into an LLM, which is then prompted to produce the final prediction.

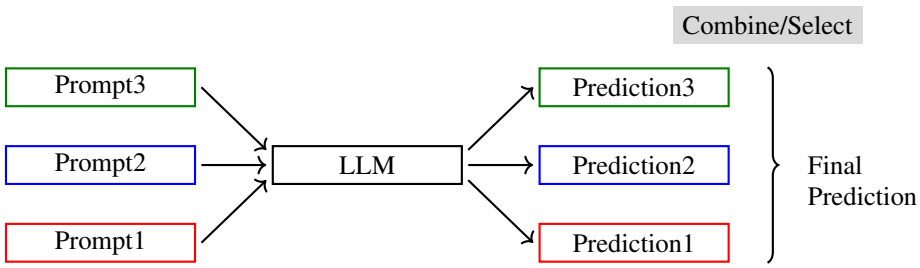
Steps 1 and 2 can be implemented by using an external information retrieval system. For example, we can store the collection of texts in a vector database and then retrieve the most similar texts through vector-based search techniques. Since information retrieval is not the

output $\mathbf{y} \in \Omega$ is given by

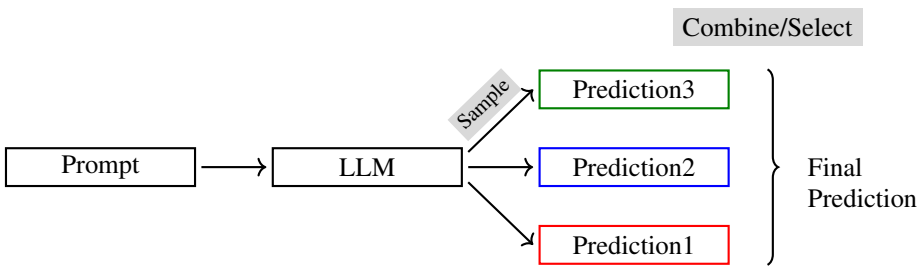
$$\begin{aligned} \text{Risk}(\mathbf{y}) &= \mathbb{E}_{\mathbf{y}_r \sim \Pr(\mathbf{y}_r|\mathbf{x})} R(\mathbf{y}, \mathbf{y}_r) \\ &= \sum_{\mathbf{y}_r \in \Omega} R(\mathbf{y}, \mathbf{y}_r) \cdot \Pr(\mathbf{y}_r|\mathbf{x}) \end{aligned} \tag{9.9}$$



(a) Model Ensembling



(b) Prompt Ensembling



(c) Output Ensembling

Figure 9.2: Ensembling methods for LLMs. In standard model ensembling (a), multiple LLMs varying in architectures or parameters are used. Each LLM receives the same prompt and produces a prediction. These predictions are combined to generate the final prediction. In prompt ensembling (b), we have one LLM and multiple prompts. The LLM produces a prediction for each prompt, and these predictions are combined as usual. In output ensembling (c), the LLM samples multiple predictions over the prediction space given a prompt. It can be seen as a method to boost the performance of the LLM itself. Note that these ensembling methods can be combined to increase the diversity of predictions. For example, we can use both prompt ensembling and output ensembling to obtain more diverse predictions.

focus of this chapter, we will assume that such systems are available off-the-shelf and use them directly.

Here we present how to prompt LLMs to make use of retrieved texts. To illustrate, consider

an example of using LLMs to answer the following question.

Where will the 2028 Olympics be held?

We can simply input this question into an online search engine. It will then return the relevant pieces of text found on the internet, for example,

> (Wikipedia)
> *The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28, is an upcoming international multi-sport event scheduled to take place from July 14-30, 2028, in the United States. ...*
>
> (The Sporting News)
> *In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after hosting the Summer Games in 1932 and 1984. It will also be the first time the United States has hosted an Olympic Games since the 2002 Winter Games in Salt Lake City. ...*
>
> ...

We can use these retrieved texts as additional context, and prompt an LLM to generate a response based on these texts. Below is an example RAG prompt.

> Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.
>
> Question:
> Where will the 2028 Olympics be held?
>
> Relevant Text 1:
> The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28 ...
>
> Relevant Text 2:
> In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after ...
>
> ...
>
> The 2028 Olympics will be held in Los Angeles.

This prompt assumes that the provided texts are relevant to the question and expects the LLM to generate a faithful response using these texts. However, the information retrieval system may sometimes provide irrelevant or incorrect texts, which may lead the LLM to produce an incorrect answer. One straightforward way to address this issue is to improve the accuracy of the information retrieval system. Nevertheless, as with most AI systems, errors may still occur. Therefore, it is also necessary to enhance the robustness of the LLM, so that it

can make reasonable predictions even when the input is inaccurate. Below is a new prompt that enables the LLM to be more faithful to the facts, and allows it to choose not to answer questions when the information provided is inaccurate.

> Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.
>
> Please note that your answers need to be as accurate as possible and faithful to the facts. If the information provided is insufficient for an accurate response, you may simply output "No answer!".
>
> Question:
>
> Where will the 2028 Olympics be held?
>
> Relevant Text 1:
>
> The 2024 Summer Olympics, officially the Games of the XXXIII Olympiad and branded as Paris 2024, were an international multi-sport event ...
>
> ...
>
> No answer!

In this example, the LLM refuses to answer because the provided information is insufficient and irrelevant to the question.

Both RAG and fine-tuning are common methods for adapting LLMs using task-specific data. Standard RAG is training-free and can be directly applied to LLMs. To further improve RAG, it is also possible to fine-tune LLMs, though this will require some training effort. For example, we can fine-tune LLMs using human-labelled data to supervise them in learning to refuse to answer. Note that, while the examples shown above seem simple, RAG is not trivial. From the prompt engineering perspective, different use cases may require different prompts, though our somewhat "greedy" goal is to develop a universal prompting strategy that can adapt to different tasks. In many cases, we need to control how much we depend on the retrieved context to make predictions. Sometimes, LLMs must derive responses strictly from the provided texts, while at other times, they may need to generate responses using their pre-trained knowledge if the provided texts are insufficient. There are many aspects of RAG, such as improvements to the retrieval systems, that cannot be covered in this chapter. Interested readers can refer to surveys of RAG techniques for more information [Li et al., 2022; Gao et al., 2023b].

One reason we discuss RAG here is that it can be broadly regarded as an instance of the general problem decomposition framework (see Section 9.2.2). RAG divides problem-solving into two steps. In the first step, we collect relevant and supporting information for a given query from various knowledge sources. In the second step, we use LLMs to generate responses based on the collected information. If we extend the concept of problem decomposition further, we will find that many tasks requiring the use of external systems or tools can be treated as

similar problems. One such example is tool use in LLMs. In many applications, LLMs need to employ external databases, APIs, and even simulation tools to generate accurate responses. For example, LLMs can access real-time data from financial markets to provide up-to-date investment advice or integrate with healthcare databases to offer personalized medical insights. This integration extends the capabilities of LLMs by allowing them to interact with, and in some contexts, influence or control external systems. Consequently, LLMs function more as **autonomous agents** rather than mere text generators [Franklin and Graesser, 1996].

The issue of tool use is broad and vast. Here we narrow our discussion to tasks that can be facilitated by calling external APIs to solve some of the sub-problems [Parisi et al., 2022; Gao et al., 2023a]. Consider again the example of asking an LLM to answer "Where will the 2028 Olympics be held?". Suppose the LLM can access a web search tool. We can then prompt the LLM to answer the question with web search, like this

> Your task is to answer the following question. You may use external tools, such as web search, to assist you.
>
> Question:
>
> Where will the 2028 Olympics be held?
>
> The information regarding this question is given as follows:
>
> `{tool: web-search, query: "2028 Olympics"}`
>
> So the answer is: Los Angeles

Here `{tool: web-search, query: "2028 Olympics"}` indicates a request to the web search system using the query "2028 Olympics". When the LLM sees this string, it executes a web search and uses the result to replace the string. Then, in subsequent steps of prediction, the LLM uses this web search result as context to produce the correct answer.

Consider another example where we ask the LLM to solve a mathematical problem.

Problem:

A swimming pool needs to be filled with water. The pool measures 10 meters in length, 4 meters in width, and 2 meters in depth. Calculate the volume of the pool in cubic meters and then determine how many liters of water are needed to fill it (considering 1 cubic meter equals 1000 liters).

Solution:

To solve this problem, the LLM needs to first calculate the volume of the pool by using the formula for the volume of a rectangular prism: Length$\times$ Width $\times$ Depth. Therefore, The volume is $10\,\text{m} \times 4\,\text{m} \times 2\,\text{m} =$ `{tool: calculator, expression: 10 * 4 * 2}` m^3. Next, to find out how many liters of water are needed, the LLM multiplies the volume in cubic meters by 1000 (since 1 cubic meter equals 1000 liters). Thus, 80 $\times 1000 =$ `{tool: calculator, expression: 80*1000}` liters.

Here the string `{tool: calculator, expression: 10 * 4 * 2}` triggers the invocation of a mathematical interpreter to calculate the result of the expression. Note that the result (i.e., 80) will replace `{tool: calculator, expression: 10 * 4 * 2}` and can be referred to in the following token predictions. For example, in the last step of problem-solving, 80 is used instead of `{tool: calculator, expression: 10 * 4 * 2}`.

A key difference between the tool use examples here and the previously discussed RAG examples is that in tool use, external functions can be called during inference. In contrast, in RAG, the retrieved texts are provided before the prediction process begins. However, from the language modeling perspective, they are actually doing the same thing: before generating the final result, we use external tools, either manually or automatically, to obtain sufficient and relevant context. A high-level interpretation of these approaches is that they both rely on an "agent" that can determine where and how to call external functions to generate the context necessary for prediction.

An issue with tool use is that the original LLMs are not trained to generate the necessary markers for tool use. Therefore, we need to fine-tune the LLMs to adapt them for these tasks [Schick et al., 2024]. As this chapter focuses on prompting, we will not present the details of this fine-tuning process. To put it simply, we first need to annotate data. For each fine-tuning example, we replace parts of the output that require the use of external tools with predefined commands or markers. Then, we use this labeled data to fine-tune the parameters of the LLM as usual. As a result, the LLM can gain the ability to generate commands for calling external tools. During inference, we can execute these tool use commands in the model outputs to get assistance from external tools.

9.3 Learning to Prompt

So far in this chapter, we have considered several basic prompting strategies and various refinements to them. However, all the prompts we have discussed were designed manually. This leads to a number of problems: First, designing high-quality prompts is inherently difficult and requires substantial manual effort. For example, extensive experimentation with different prompts is often needed to identify the most effective ones. Since different LLMs may respond better to certain types of prompts, developing universally effective prompts can be even more resource-intensive. Second, manual prompt design relies heavily on human expertise, which can limit the diversity of approaches and overlook potentially effective prompts that are not immediately obvious to humans. Third, prompts created by humans can be complex and redundant, leading to longer inputs for LLMs and higher computational costs.

In this section, we discuss techniques for automated prompting. These methods aim to automatically create, optimize, and represent prompts so that the downstream tasks can be addressed more effectively and efficiently. In particular, we consider three issues here.

- How can we automate the process of designing and optimizing prompts for LLMs?
- Are there other forms of representing prompts beyond strings, and how can we learn such representations?
- How can we make prompts more concise and compact, thereby reducing their complexity and length?

Note that there are many settings in which we can investigate these issues. For example, we might specify that prompts are developed specifically for a particular LLM, or that the development is independent of the LLM used. These settings can lead to different methods and application scenarios, but these methods may overlap in some ways. In the following discussion, we will cover several different scenarios and discuss the connections between various methods.

9.3.1 Prompt Optimization

Given that prompt design is difficult and labor-intensive, it is desirable to use machine learning models to discover the optimal prompt for a specific task (call it **automatic prompt design** or **prompt optimization**). This approach can broadly be regarded as an instance of **automated machine learning** (**AutoML**), which aims to reduce or eliminate the need for expert-driven manual design of machine learning models. Although our focus here is on the design of prompts, prompts themselves are discrete structures. Therefore, designing prompts is very similar to designing machine learning models, such as discrete model architectures. Perhaps one of the most related fields is **neural architecture search** (**NAS**), where the most optimal neural networks are identified by exploring a space of possible neural networks [Zoph and Le, 2016; Elsken et al., 2019]. If we consider prompt optimization as a search process, then we can describe a general prompt optimization framework involving the following components:

- **Prompt Search Space**. This defines all possible prompts that the algorithms can explore. For example, one can edit some seed prompts to generate a set of diverse candidate

prompts.

- **Performance Estimation**. Once a prompt is chosen, it needs to be evaluated. For example, a straightforward way is to input it to an LLM and measure its performance on a validation set.
- **Search Strategy**. The search process is generally the same as that used in many AI systems. At each step, the system explores a set of promising prompts in the search space and evaluates them. This process continues as more prompts are explored. The outcome of the search is the best-performing prompt observed until the search stops.

This is a very general framework, and different prompt optimization systems can vary in their design of each component. A widely-used approach is to use LLMs as the basis to develop these components. Initially, a few prompts are provided. Then, the following process is iterated until a stopping criterion is met: 1) the prompts are evaluated on a validation set; 2) a candidate pool is maintained by keeping only the most promising prompts; and 3) new prompts are created by employing LLMs to infer similar prompts from this candidate pool. One benefit of this approach is that it allows us to use off-the-shelf LLMs to perform the tasks mentioned above without the need for substantial system development. To achieve this, we can prompt or fine-tune LLMs to adapt them to these tasks. Here we consider Zhou et al. [2023b]'s method for illustrating LLM-based prompt optimization. It involves the following steps.

- **Initialization**. Let C represent the pool of the candidate prompts we intend to explore. The first step is to add initial prompts into C. We can do this in several ways. A simple method is to create such prompts by hand for a given task. However, in many cases where humans have limited knowledge about how to write effective prompts for the task, developing prompts becomes challenging. In these cases, it is desirable to use LLMs to generate prompts. For example, we can directly instruct LLMs to produce prompts, providing them with a description of the task.

> You are given a task to complete using LLMs. Please write a prompt to guide the LLMs.
>
> {∗task-description∗}
>
> ____

This method is straightforward, but it still requires a human-provided description of the task. An alternative method is to use LLMs to generate prompts given examples of the input and output of the task. Here is a prompt template.

> You are provided with several input-output pairs for a task. Please write an instruction for performing this task.
>
> Input: $\{*\text{input1}*\}$ Output: $\{*\text{output1}*\}$
> Input: $\{*\text{input2}*\}$ Output: $\{*\text{output2}*\}$
> ...
>
> ——

As such, LLMs can infer the corresponding instruction for the task from the provided inputs and outputs.

- **Evaluation**. Once we obtain the candidate pool C, we need to evaluate the prompts in C. One method is to feed each prompt into an LLM and assess the results on the downstream task. For example, we can evaluate the output of the LLM given an input using a pre-defined metric, or alternatively, use the log-likelihood of the output as a measure of the quality of the prompt.
- **Pruning**. If C contains a large number of prompts, it is reasonable to prune the unpromising prompts within it, thus reducing the computational burden in subsequent steps. This is a standard pruning problem. Given the evaluation score for each prompt, a simple method is to keep only a certain percentage of the prompts and discard the rest.
- **Expansion**. Expansion is a key operation in search algorithms used to explore different states in the search space. The expansion operation here can be defined as a function

$$C' = \text{Expand}(C, f) \tag{9.10}$$

where C' is the set of new prompts generated from C using the model f. If we consider f as an LLM, we can perform the expansion operation by instructing f to generate new and relevant prompts based on C. Below is an example.

> Below is a prompt for an LLM. Please provide some new prompts to perform the same task.
>
> Input: $\{*\text{prompt}*\}$
>
> ——

Then, we replace C with C'. The steps of evaluation, pruning and expansion can be repeated, and so we can gradually explore a wider range of prompts.

In prompt optimization, the expansion step plays a key role, as it defines how we explore the search space, and our goal is to find optimal results with minimal effort. One improvement to this step is to treat the problem as a paraphrasing task. A simple method is to apply off-the-shelf paraphrasing systems, either based on LLMs or other models, to transform input prompts

into semantically equivalent forms [Jiang et al., 2020]. Alternatively, we can define specific edit operations, such as insertions and modifications, for each token. A given prompt can be edited into new prompts by applying these operations [Prasad et al., 2023]. Also, further evaluation and pruning can be applied to filter out low-quality prompts. In addition to framing prompt generation as a paraphrasing problem, we can improve the quality of prompts during expansion by learning from feedback [Pryzant et al., 2023]. This approach is somewhat related to the self-refinement issue discussed in Section 9.2.3. An LLM can be used to generate feedback on an input prompt, which is then revised based on this feedback. This feedback-and-revision cycle can be repeated multiple times until the result converges or the desired outcome is achieved.

Another approach to prompt optimization is to apply classic optimization techniques. For example, the problem can be framed as an evolutionary computation problem, where prompts are treated as candidates that evolve generation by generation as the optimization progresses [Guo et al., 2024]. Since many powerful optimization algorithms have been developed in related fields, they can be directly applied to this problem.

In practice, we might be tempted to use existing LLM APIs to implement the steps described above. Such an approach, however, would be strongly dependent on the inference and in-context learning abilities of the LLMs. If these LLMs are not strong and lack adaptation to the tasks, they may introduce errors into search, for example, generating incorrect prompts during expansion. In such cases, it is preferable to train models that are better suited to the tasks. One approach in this research direction appeals to reinforcement learning, which has been widely used in solving discrete decision making and optimization problems. For example, Deng et al. [2022] developed a prompt generator by integrating an FFN-based adaptor into an LLM. The prompt generator is trained as a typical policy network, but only the parameters of the adaptor are updated while the remaining parameters of the model are kept unchanged. During training, the reward is obtained by testing the generated prompts using another LLM, similar to the evaluation method as discussed above. Once the training is complete, the prompt generator is then employed to generate new prompts.

Note that, in our discussion here, prompts are simply seen as sequences of tokens, and the output of prompt optimization is such a sequence. However, in a strict sense, prompts have complex structures and include different fields such as user input, instruction, and demonstration. While our discussed approaches are mostly general, much work in prompt optimization has focused on learning better instructions for prompting. Specifically, the goal is to generate instructions that effectively guide LLMs based on a given task. Of course, the concept of prompt optimization can also be extended to learning other parts of prompts. For example, there has been substantial research interest in learning to select or generate demonstrations in CoT [Liu et al., 2022; Rubin et al., 2022; Zhang et al., 2023b]. One of the differences between learning instructions and learning demonstrations is that generating high-quality demonstrations using LLMs is relatively easy and the focus of learning demonstrations is typically on how to sample appropriate demonstrations from a pool of candidates. In contrast, the difficulty in learning instructions is partly because pre-trained LLMs are not suited to predict the quality of instructions, and testing these instructions on downstream

tasks is computationally expensive. This makes the optimization methods costly to apply, and exploring a wide variety of instructions poses significant challenges.

9.3.2 Soft Prompts

Although developing natural language prompts, either manually or automatically, is a straightforward and widely applied approach, it presents some problems. One problem is that natural language prompts can be complex and lengthy, resulting in significant computational burdens when processed via LLMs. In many applications, users may need to perform a task repeatedly, and inputting the same long prompt into the LLMs a large number of times is clearly inefficient. Another problem is that while prompts are typically represented as discrete token sequences (call them **hard prompts**) in regular LLM input, the LLMs encode them as low-dimensional real-valued vectors. This raises the question of whether there are more compact and efficient ways to represent prompts.

In this subsection, we introduce the concept of **soft prompts**, which can be viewed as hidden, distributed representations of prompts. When prompting LLMs, we are concerned with communicating tasks or questions to elicit the desired responses. We can define hard prompts as explicit, predefined text sequences that users input directly into LLMs to guide the responses. In contrast, we can think of soft prompts as implicit, adaptable prompting patterns embedded within LLMs. Unlike hard prompts, which are expressed in natural language and should be understandable for humans, soft prompts are encoded in a format that is more comprehensible to the model rather than to humans. To illustrate, consider a simple prompt

> Translate the sentence into Chinese.
>
> Consider it done!
>
> ——

Here, the instruction "Translate the sentence into Chinese" can be seen as a hard prompt, denoted by the token sequence $c_1...c_5$. By feeding these tokens into an LLM, they are transformed into a sequence of real-valued vectors $\mathbf{h}_1...\mathbf{h}_5$, each corresponding to a token. We can roughly think of $\mathbf{h}_1...\mathbf{h}_5$ as a soft prompt, as illustrated in Figure 9.3.

While the above example shows that soft prompts can be generated by transforming hard prompts, there is not necessarily a direct correspondence between them. In fact, we do not even need to interpret soft prompts using meaningful text. They are instead simply hidden states in LLMs and can be learned as standard parameters of the models through continuous optimization. Such a treatment allows us to explore prompting methods beyond text. As another benefit, soft prompts provide dense, low-dimensional, and learnable representations for encoding how we guide LLMs to generate specific outputs. The training and application of these representations require significantly lower computational costs than those required for processing long hard prompts. This approach would be of great practical value in LLM inference applications where the same prompt is repeatedly used.

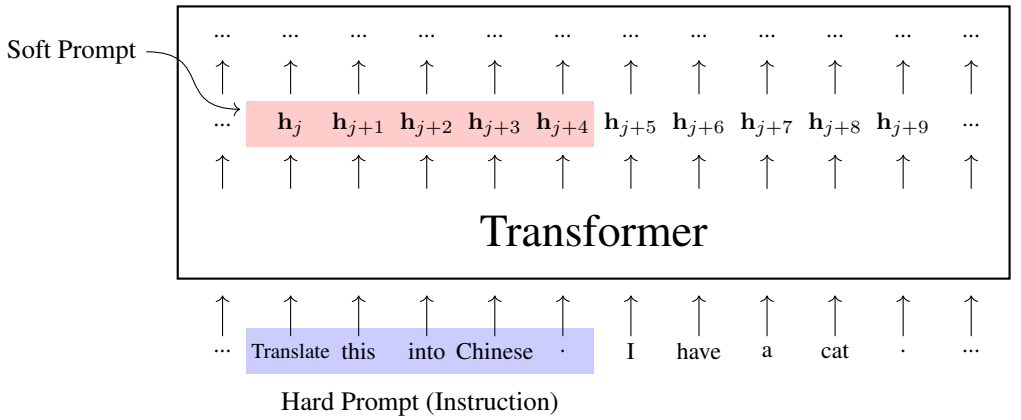


Figure 9.3: Illustration of hard and soft prompts. Here the hard prompt is the instruction we input to the LLM for performing the task. The LLM encodes this instruction as usual, and the intermediate representations corresponding to the instruction can be viewed as some sort of soft prompt.

1. Adapting LLMs with Less Prompting

One obvious way to adapt an LLM for a particular task is to simply fine-tune the model using labeled data. This leads to a variety of LLM alignment methods, such as supervised fine-tuning, which update the model parameters by aligning the responses to given prompts with supervision signals. Fine-tuned LLMs embed task-related information in model parameters, and thus these models can respond correctly when dealing with similar prompts with those in fine-tuning.

If we take this idea further, we can expect LLMs to absorb the knowledge about prompting of a task as much as possible during fine-tuning. Consequently, the prompting information is partially captured in the model parameters, and the fine-tuned LLMs can perform the task with less prompting. Here we consider a simple form of prompt, where only an instruction (denoted by $\mathbf{c}$) and a user input (denoted by $\mathbf{z}$) are included. A prompt can be expressed using the following tuple

$$\mathbf{x} \quad = \quad (\mathbf{c}, \mathbf{z}) \tag{9.11}$$

Given a set of prompt-response pairs $\mathcal{D} = \{(\mathbf{x}, \mathbf{y})\}$, the objective of fine-tuning is to minimize the total loss incurred over this set. A popular method is to minimize the negative log-likelihood (i.e., maximize the log-likelihood) with respect to the model parameters θ:

$$\begin{aligned} \hat{\theta} &= \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_\theta(\mathbf{y}|\mathbf{x}) \\ &= \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_\theta(\mathbf{y}|\mathbf{c},\mathbf{z}) \end{aligned} \tag{9.12}$$

where $\Pr_\theta(\cdot|\cdot)$ is the probability predicted by an LLM with the parameters θ[9].

In general, the instruction in each fine-tuning example should follow the guideline of prompt design, for example, a good instruction should be as clear as possible and provide a detailed description of the task. However, the method described in the above equation does not restrict the instruction to any particular form. This flexibility allows us to instruct LLMs in any way that we want. Consider an example where we intend to instruct LLMs to translate an English sentence into Chinese. Of course, as mentioned earlier in this chapter, we can prompt LLMs using the instruction

Translate the following sentence from English to Chinese.

If we want the instruction to be simpler, we may rephrase it into a simpler form

Translate this into Chinese.

Even, we can define the instruction as a single phrase

Translate!

With certain fine-tuning effort, we can adapt LLMs to follow any of these instructions. From an efficient prompting perspective, there are computational advantages in simplifying instructions in prompting. For example, we can use simple instructions like "Translate!" to perform tasks that would typically require more complex and detailed instructions. This can make subsequent prompting during inference much easier. On the other hand, fine-tuning LLMs with overly simplified instructions may be harmful to the generalization of the models. Since simplified instructions can lead to a loss of information, it is more likely that the LLMs will overfit the fine-tuning data and fail to generalize beyond those instructions. In scenarios involving both complex and simplified instructions for fine-tuning, this problem is more severe because the labeled data available for fine-tuning is usually limited, and accommodating a variety of instructions is costly.

An alternative way to adapt LLMs for simplified instructions is through knowledge distillation. As an example, we consider the context distillation method [Snell et al., 2022]. The goal of this method is to learn a student model that can make use of simplified instructions from a well-trained instruction-following teacher model. Figure 9.4 shows an illustration of this approach. Building the teacher model follows a standard fine-tuning process: we first collect a certain amount of data that includes instructions, user inputs, and correct responses, and then we continue to train a pre-trained model with this dataset. For building the student model, we need to construct a new dataset $\mathcal{D}'$ where each sample is a tuple consisting of an instruction, a corresponding simplified instruction, and a user input, denoted by $\mathbf{x}' = (\mathbf{c}, \mathbf{c}', \mathbf{z})$. Knowledge distillation is performed by minimizing a loss function defined on the outputs of the teacher

[9] In practice, we initialize θ with the parameters obtained from pre-training, and then adjust θ moderately to ensure that the results after fine-tuning do not deviate too much from the pre-trained results.

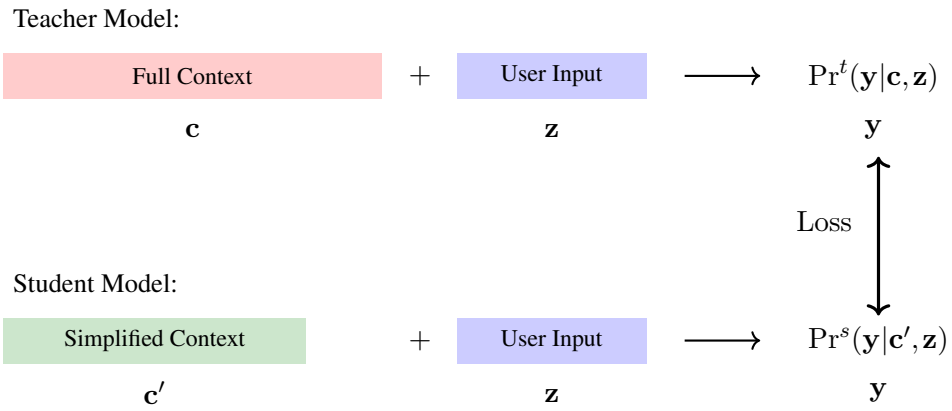


Figure 9.4: Illustration of context distillation [Snell et al., 2022]. The teacher model is a standard LLM, which takes both the context and the user input as model input and produces a prediction as model output. Then, we simplify the context (e.g., simplifying the instruction in prompting) and use the student model to make predictions based on the simplified context and the user input. The student model is trained by minimizing the loss between the predictions produced by the two models.

and student models

$$\hat{\theta} = \arg\min_{\theta} \sum_{\mathbf{x}' \in \mathcal{D}'} \text{Loss}(\Pr^t(\cdot|\cdot),\ \Pr^s_\theta(\cdot|\cdot),\ \mathbf{x}') \tag{9.13}$$

where $\Pr^t(\cdot|\cdot)$ denotes the pre-trained teacher model, and $\Pr^s_\theta(\cdot|\cdot)$ denotes the student model with the parameters θ. To keep the notation simple we will write $\text{Loss}(\Pr^t(\cdot|\cdot),\ \Pr^s_\theta(\cdot|\cdot),\ \mathbf{x})$ as Loss for short. A commonly-used loss is the sequence-level loss, which has the basic form:

$$\text{Loss} = \sum_{\mathbf{y}} \Pr^t(\mathbf{y}|\mathbf{c},\mathbf{z}) \log \Pr^s_\theta(\mathbf{y}|\mathbf{c}',\mathbf{z}) \tag{9.14}$$

But this function is computationally infeasible because it requires summing over an exponentially large number of outputs. A variant of this method is to train the student model using outputs generated by the teacher model. For each sample, we use the teacher model to produce an output $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \Pr^t(\mathbf{y}|\mathbf{c},\mathbf{z})$. Then we consider $\hat{\mathbf{y}}$ as the target for learning, and the loss function is given by

$$\text{Loss} = \log \Pr^s_\theta(\hat{\mathbf{y}}|\mathbf{c}',\mathbf{z}) \tag{9.15}$$

Alternatively, we can minimize the distances between the probability distributions outputted by the two models [Askell et al., 2021]. For example, the loss function can be defined as the KL divergence between the two output distributions

$$\text{Loss} = \text{KL}(\mathrm{P}^t \,||\, \mathrm{P}^s_\theta) \tag{9.16}$$

where

$$\mathrm{P}^t = \mathrm{Pr}^t(\cdot|\mathbf{c},\mathbf{z}) \tag{9.17}$$
$$\mathrm{P}^s_\theta = \mathrm{Pr}^s_\theta(\cdot|\mathbf{c}',\mathbf{z}) \tag{9.18}$$

Although we have restricted ourselves to knowledge distillation for instructions, the approaches discussed here are general. By learning from the outputs of the teacher model, the knowledge in prompting can be distilled into the parameters of the student model. Therefore, the distilled model can be considered as encoding some sort of soft prompt. This method can be applied to many other problems in prompt learning, such as compressing long contexts and learning soft prompts as specific components of LLMs.

2. Learning Soft Prompts for Parameter-efficient Fine-tuning

Updating all parameters is a common method for adapting LLMs to tasks of interest. Although fine-tuning is considered computationally cheaper than pre-training, it is still costly to apply in practice. This issue motivates the development of parameter-efficient fine-tuning methods, which aim to minimize the number of parameters that need to be updated.

One approach, known as **prefix fine-tuning**, is to append a series of trainable vectors, or prefixes, at the beginning of the input of each Transformer layer [Li and Liang, 2021]. These prefixes can be thought of as soft prompts that serve as additional context to guide the behavior of the model under specific tasks. During fine-tuning, we need only to learn the prefixes for embedding task-specific knowledge. Thus, this method is efficient because it only modifies a small part of the model rather than adjusting the entire set of model parameters.

Specifically, let the input of a layer at depth l be denoted by $\mathbf{H}^l = \mathbf{h}^l_0\mathbf{h}^l_1...\mathbf{h}^l_m$. The output of the layer can be expressed as

$$\mathbf{H}^{l+1} = \mathrm{Layer}(\mathbf{H}^l) \tag{9.19}$$

In prefix fine-tuning, we extend the sequence $\mathbf{h}^l_0\mathbf{h}^l_1...\mathbf{h}^l_m$ by adding a few vectors at the beginning, which we denote as $\mathbf{p}^l_0\mathbf{p}^l_1...\mathbf{p}^l_n$. Hence $\mathbf{H}^l$ can be written in the form

$$\mathbf{H}^l = \underbrace{\mathbf{p}^l_0\ \mathbf{p}^l_1\ ...\ \mathbf{p}^l_n}_{\text{trainable}}\ \underbrace{\mathbf{h}^l_0\ \mathbf{h}^l_1\ ...\ \mathbf{h}^l_m}_{\text{previous layer output}} \tag{9.20}$$

The output of the layer is the last $m+1$ representations.

$$\begin{aligned}\overline{\mathbf{H}}^{l+1} &= \mathrm{Layer}(\mathbf{H}^l)[-m-1:] \\ &= \mathbf{h}^{l+1}_0\mathbf{h}^{l+1}_1...\ \mathbf{h}^{l+1}_m\end{aligned} \tag{9.21}$$

where $[-m-1:]$ denotes the slicing operation that extracts the last $m+1$ elements of a sequence. Given $\overline{\mathbf{H}}^{l+1}$, the input of the next layer can be expressed in the same form of Eq.

(9.20):

$$\begin{aligned}\mathbf{H}^{l+1} &= \mathbf{p}_0^{l+1}\ \mathbf{p}_1^{l+1}\ ...\ \mathbf{p}_n^{l+1}\ \overline{\mathbf{H}}^{l+1} \\ &= \mathbf{p}_0^{l+1}\ \mathbf{p}_1^{l+1}\ ...\ \mathbf{p}_n^{l+1}\mathbf{h}_0^{l+1}\mathbf{h}_1^{l+1}...\ \mathbf{h}_m^{l+1}\end{aligned} \tag{9.22}$$

Here each $\mathbf{p}_i \in \mathbb{R}^d$ can be seen as a learnable parameter. During training, $\mathbf{p}_0^l\mathbf{p}_1^l...\mathbf{p}_n^l$ are trained as usual, and the parameters of the original Transformer model are kept fixed.

Figure 9.5 shows an illustration of prefix fine-tuning for a translation task. Here, only the prefix vectors $\mathbf{p}_0^l$ and $\mathbf{p}_1^l$ are updated by receiving the error gradients from the output (i.e., the Chinese translation). By adjusting these vectors for the translation task, the model adapts accordingly. This makes $\mathbf{p}_0^l$ and $\mathbf{p}_1^l$ serve as prompts which activate the LLM to perform the task without needing explicit input prompts like "Translate the following sentence from English to Chinese". At test time, we prepend the optimized $\mathbf{p}_0^l$ and $\mathbf{p}_1^l$ to the layer, and the LLM will then translate the input sentence. Note that prefix fine-tuning introduces additional $L \times n \times d$ parameters, where L is the number of layers, n is the number of prefixes, and d is the dimensionality of each prefix. However, this number is much smaller compared to the total number of parameters in the LLM, making the fine-tuning process highly efficient.

While prefix fine-tuning is simple, it still requires modifications to LLMs. Alternatively, separating soft prompts from the LLMs allows us to preserve the original model architecture, making it more efficient for deployment across different tasks without the need to adjust the core model. One such method is prompt tuning [Lester et al., 2021]. Like prefix fine-tuning, prompt tuning incorporates trainable vectors so that LLMs can adapt to given tasks by adjusting these vectors. However, prompt tuning differs in that it modifies only the embedding layer.

Recall that in LLMs each input token z_i is represented by an embedding $\mathbf{e}_i$. These embeddings are generally learned through a token embedding model and are then used as the real inputs to the LLMs, replacing the symbolically represented tokens. In prompt tuning, a number of pseudo embeddings $\mathbf{p}_0...\mathbf{p}_n$ are added at the beginning of the token embedding sequence. So the actual input to the LLMs can be expressed as

$$\underbrace{\mathbf{p}_0\ \mathbf{p}_1\ ...\ \mathbf{p}_n}_{\text{trainable}}\ \underbrace{\mathbf{e}_0\ \mathbf{e}_1\ ...\ \mathbf{e}_m}_{\text{token embeddings}}$$

Note that a pseudo embedding needs not to correspond to any token in natural language. Instead these embeddings can be seen as "soft prompt embeddings" that serve to condition the LLMs. By training soft prompt embeddings on task-specific data, they learn to interact adaptively with the token embeddings $\mathbf{e}_0...\mathbf{e}_m$ and guide the behavior of LLMs. Since prompt tuning does not change the underlying parameters of pre-trained LLMs, it is considered a lightweight and efficient method of fine-tuning, improving task-specific performance while maintaining their generalization capabilities. See Figure 9.6 for an illustration of prompt tuning.

Since $\mathbf{p}_0\ \mathbf{p}_1\ ...\ \mathbf{p}_n$ is itself a sequence, we can employ sequence models to better represent it. For example, a Transformer model can encode this sequence, and the resulting representation can then be used as the input to the LLM. In other words, we can develop an additional model for encoding soft prompts. Another way to improve prompting is by combining soft and

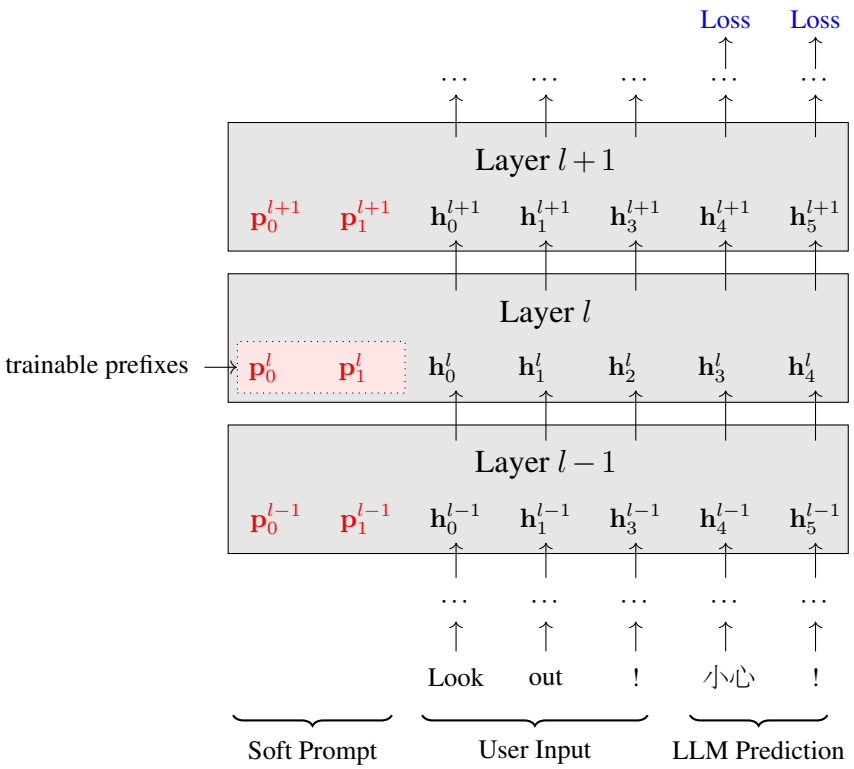
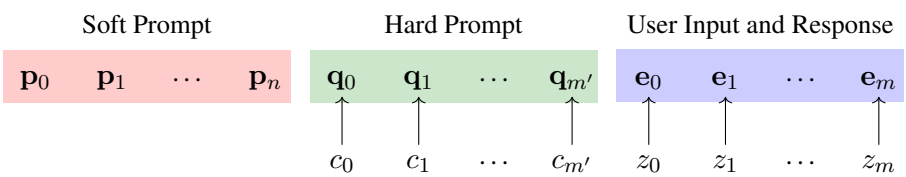


Figure 9.5: Illustration of prefix fine-tuning for a translation task (Look out! → 小心!). For each layer, we add two prefixes $\mathbf{p}_0^l$ and $\mathbf{p}_1^l$ at the beginning. The LLM is trained to minimize the loss on the predictions given the input. During this process, only the prefixes are optimized while the rest of the parameters remain fixed. Therefore, the model can adapt to the given task in a very efficient manner. At inference time, the LLM works with optimized prefixes, and can perform the task without the need of explicit hard prompts.

hard prompts, thereby taking advantage of both types [Liu et al., 2023b]. In the embedding sequence, we can arrange or intersperse these prompts. This would result in different prompt patterns. For example, a simple pattern that uses both two types of prompt is



where $c_0 ... c_{m'}$ denotes the hard prompt and $\mathbf{q}_0 ... \mathbf{q}_{m'}$ denotes the corresponding embedding sequence.

Here we have considered methods for inserting soft prompts in LLMs. But we skip the details of training these soft prompts and assume that the reader is familiar with the standard supervised learning process, that is, maximizing the likelihood of the correct model output

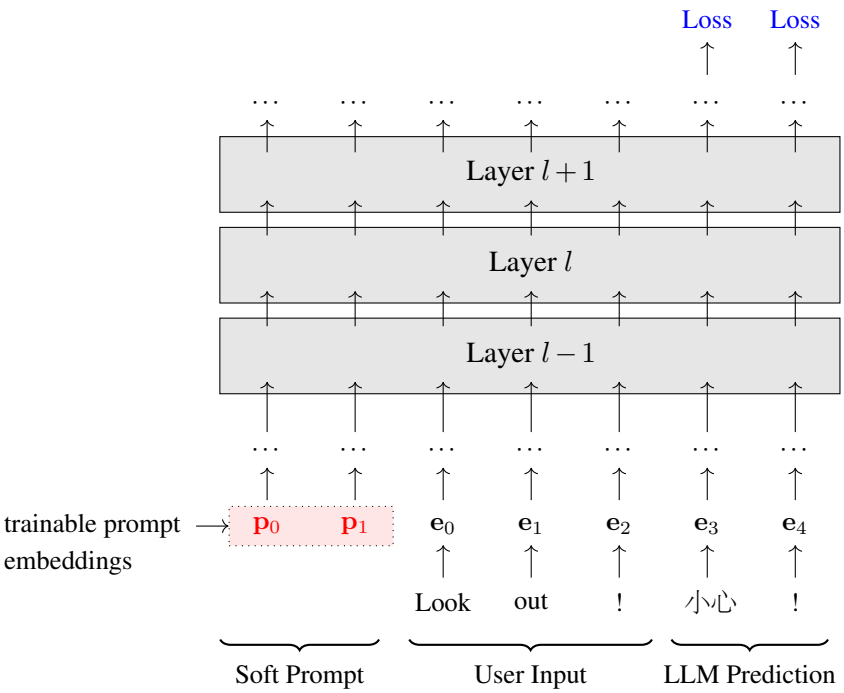
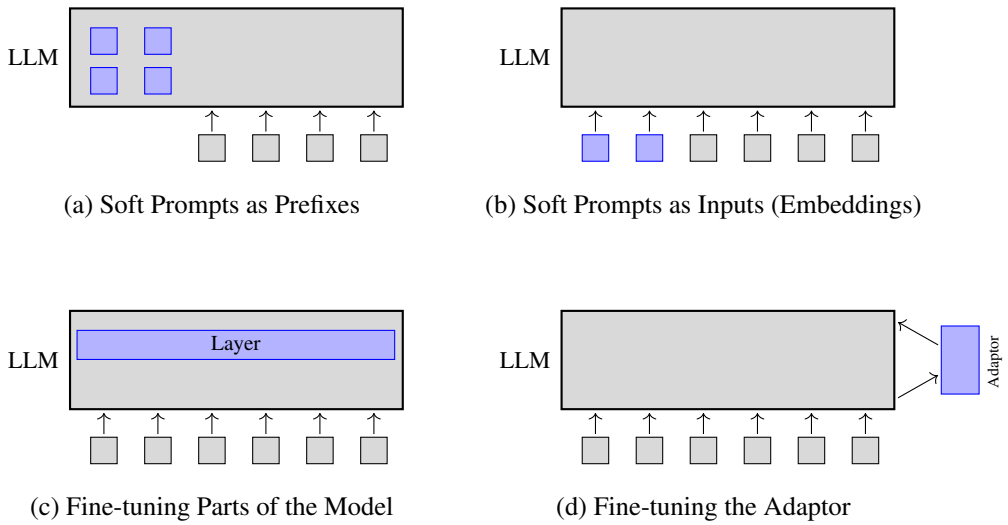


Figure 9.6: Illustration of prompt tuning for a translation task (Look out! → 小心!). Instead of using fixed textual prompts, soft prompts are learnable embeddings that are added at the beginning of the embedding sequence. During fine-tuning, only these prompt embeddings are optimized to efficiently adapt the LLM to the given task. Once optimized, the prompt embeddings are used to instruct the LLM to perform the task as new data arrives.

given the model input. In fact, learning soft prompts can be related to many issues in LLM fine-tuning. For example, if we consider it as a context compression problem, we can apply the knowledge distillation methods described previously. In Mu et al. [2024]'s work, prompts are compressed and represented as a few pseudo tokens, which are appended to each input sequence. The embeddings of these pseudo tokens are optimized to mimic the predictions of a standard-prompted model. In other words, the prompting knowledge is distilled from a teacher model into the pseudo tokens.

Broadly speaking, many parameter-efficient fine-tuning methods can be thought of as learning some sort of soft prompt [Lialin et al., 2023]. When we fine-tune a part of an LLM for a task, this process can essentially be seen as injecting task-related prompting information into that specific part of the model. Another widely-used approach to parameter-efficient fine-tuning is to add an adaptor layer between the existing model layers. This approach allows us to fine-tune only the adaptor layer on specific tasks without altering the underlying architecture or retraining the entire model. In this sense, adaptor layers can be viewed as soft prompts that encode prompting and task-related information and interact with the original LLM to help it adapt. To summarize, Figure 9.7 shows a comparison of different methods of using soft prompts in LLMs.



(a) Soft Prompts as Prefixes

(b) Soft Prompts as Inputs (Embeddings)

(c) Fine-tuning Parts of the Model

(d) Fine-tuning the Adaptor

Figure 9.7: Illustrations of using soft prompts in LLMs. Here tunable soft prompts are shown in blue, and components whose parameters are fixed during fine-tuning are shown in gray. In sub-figure (a), soft prompts are prefixes appended to each layer of the LLM. In sub-figure (b), soft prompts are used as input embeddings for the LLM. In sub-figures (c) and (d), soft prompts are broadly treated as components of the model that are fine-tuned for task adaptation.

3. Learning Soft Prompts with Compression

Another approach to learning soft prompts is from the perspective of compression. As a simple example, consider the problem of approximating a long context using a continuous representation [Wingate et al., 2022]. Suppose we have a user input $\mathbf{z}$ and its context $\mathbf{c}$ (such as long instructions and demonstrations). Now we want to develop a compressed representation of the context, denoted by σ, such that the prediction based on $\mathbf{z}$ and σ is as close as possible to the prediction based on $\mathbf{z}$ and $\mathbf{c}$. This goal can be expressed in the form

$$\hat{\sigma} \quad = \quad \arg\min_{\sigma} s(\hat{\mathbf{y}}, \hat{\mathbf{y}}_{\sigma}) \tag{9.23}$$

where $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{c}, \mathbf{z})$ and $\hat{\mathbf{y}}_{\sigma} = \arg\max_{\mathbf{y}_{\sigma}} \Pr(\mathbf{y}|\sigma, \mathbf{z})$ are the LLM predictions given the full context and the compressed context, respectively. The function $s(\cdot, \cdot)$ typically represents a loss or similarity measure, aiming to minimize the difference in predictions between the two context representations.

One general framework for achieving this is knowledge distillation, where $\hat{\mathbf{y}}$ and $\hat{\mathbf{y}}_{\sigma}$ can be seen as the predictions of the teacher model and the student model, respectively. This formalization links our discussion to the context distillation problem discussed earlier. The training objective can be obtained by analogy with Eqs. (9.15) and (9.16). For example, a simple training objective is given by

$$\hat{\sigma} \quad = \quad \arg\max_{\sigma} \log \Pr(\hat{\mathbf{y}}|\sigma, \mathrm{z}) \tag{9.24}$$

Alternatively, we can minimize the KL divergence between the output distributions, giving

$$\hat{\sigma} = \arg\min_{\sigma} \mathrm{KL}(\Pr(\cdot|\mathbf{c},\mathbf{z}) \,||\, \Pr(\cdot|\sigma,\mathbf{z})) \tag{9.25}$$

The difference with the models in Eqs. (9.15) and (9.16) is that here the compressed context is represented as real-valued vectors (call them **prompt embeddings**), rather than as normal tokens. By applying the above methods, we distill the context from the token sequence $\mathbf{c}$ into the embeddings σ. Note that the teacher model $\Pr(\cdot|\mathbf{c},\mathbf{z})$ and the student model $\Pr(\cdot|\sigma,\mathbf{z})$ may not share the same architecture or model settings. In practice, we generally wish for the teacher model to be stronger, while the student model should be smaller and more efficient.

While compressing full context into continuous representations is a straightforward approach to learning soft prompts, it requires a teacher model that can deal with long input sequences. In many cases, however, the context is so long that applying an LLM is too costly or infeasible. Modeling long input sequences can fall under the broad family of efficient methods for long-context LLMs. Many techniques have been developed to address this issue. For example, one can use a fixed-size KV cache to store the past information at each step during inference. Efficient Transformer architectures and long-context LLMs have been intensively discussed in this book. For more detailed discussions of these topics, interested readers can refer to Chapters 6 and 8.

There are also methods specifically designed to compress long context into soft prompts. Here we consider Chevalier et al. [2023]'s method as an example. The basic idea is that we learn soft prompts gradually by accumulating the fixed-size context representation over the context sequence. Given a long context, we first divide it into a number of segments $\mathbf{z}^1, ..., \mathbf{z}^K$. We then process these segments in sequence, each time generating a representation of the context we have processed so far, denoted by $\sigma^{<i+1}$. To do this, a few summary tokens $\langle \mathrm{g}_1 \rangle, ..., \langle \mathrm{g}_\kappa \rangle$ are introduced. At each step, we take a segment $\mathbf{z}^i = z^i_1 ... z^i_{m_i}$, along with the previous context representation $\sigma^{<i}$ and the summary tokens $\langle \mathrm{g}_1 \rangle, ..., \langle \mathrm{g}_\kappa \rangle$ as input, and use an LLM to produce the corresponding hidden representation sequence at the last Transformer layer. An example of this process is illustrated in Figure 9.8.

Here $\sigma^{<i}$ is essentially a memory. The model operates in an RNN fashion. Each time we take a segment and update this memory by encoding both the previous memory state and the segment. Therefore, the $\sigma^{<i}$ produced at the last segment is a representation of the entire context sequence. The Transformer model for learning these representations can be a standard LLM but we need to fine-tune it to adapt to this context representation task.

Note that here we simply consider *prompt* and *context* as similar terms, even though they are not the same. Although we are somewhat "misusing" the concept *prompt*, we can often view it as a type of context. From this perspective, the methods discussed here can be applied to general text compression problems.

9.3.3 Prompt Length Reduction

While soft prompts provide dense, hidden representations, they are not directly interpretable. The lack of interpretability can be a significant barrier for users trying to understand how their

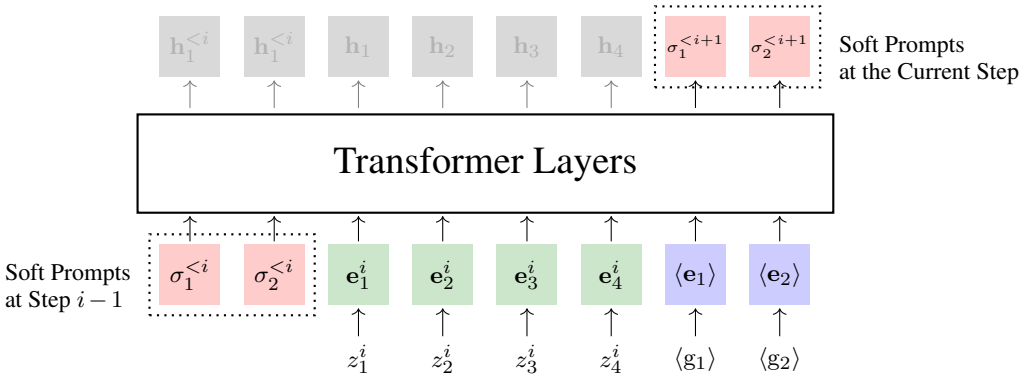


Figure 9.8: Illustration of compressing a context segment into soft prompts ($\kappa = 2$ and $m_i = 4$). The input to the LLM includes the soft prompts from the previous step ($\sigma_1^{<i}$ and $\sigma_2^{<i}$), the tokens of the segment (z_1, z_2, z_3, and z_4), and the summary tokens ($\langle \mathrm{g}_1 \rangle$ and $\langle \mathrm{g}_2 \rangle$). Given these, the LLM operates as usual. We then extract the outputs at the last Transformer layer that correspond to the summary tokens. These outputs can be viewed as the soft prompts that accumulated up to this segment.

inputs influence LLM outputs. Moreover, although soft prompts are efficient for fine-tuning and deployment, they are inflexible and do not allow for easy adjustments without extensive fine-tuning or modification. This inflexibility can limit their utility in dynamic environments where prompt changes are frequently needed.

One alternative way to develop efficient prompts is to simplify the text used for prompting. For example, below is a prompt for answering questions on healthcare and finance.

> The task involves developing a language model capable of understanding and responding to user inquiries across various domains, with a particular emphasis on healthcare and finance. Considering the broad range of potential queries, from the specifics of medical diagnoses to the nuances of financial regulations, the model must ensure a comprehensive understanding and accurate responses.
>
> Question:
>
> What are the best practices for using artificial intelligence in diagnosing cardiovascular diseases?
>
> ____

We can simplify the task description by deleting the unimportant parts.

> *The task involves developing a language model capable of understanding and responding to user inquiries ~~across various domains, with a particular emphasis~~ on healthcare and finance. ~~Considering the broad range of potential queries, from the specifics of medical diagnoses to the nuances of financial regulations,~~ The model must ensure a comprehensive understanding and accurate responses.*

We can also paraphrase it as a shorter text.

> *The task involves developing a language model focused on healthcare and finance, capable of understanding and accurately responding to a wide range of user inquiries.*

This problem can be viewed as a classic NLP issue — text simplification. So the methods used can be general and not restricted to the problem of simplifying prompts. There are many ways to achieve this. One simple method is to define some heuristics and identify redundant words that can be eliminated without losing essential information. For example, we can examine each token in a sequence in terms of its contribution to the overall meaning and remove those that provide minimal value [Li et al., 2023b; Jiang et al., 2023]. Another method involves framing the problem as a sequence-to-sequence task. With labeled data for text simplification, we can train an encoder-decoder model to transform each input text into its simplified form. In addition, given that many LLMs have been fine-tuned and aligned to perform text simplification tasks, it is straightforward to use these models to simplify prompts. For example, we can prompt an LLM to simplify a text under certain constraints, such as limiting the length of the simplified text.

9.4 Summary

In this chapter, we have discussed a variety of issues related to LLM prompting. Our discussion has focused mainly on two aspects:

- How to design basic prompts to guide the predictions of LLMs and refine these prompts for more effective and efficient problem-solving?
- How to automate the design and representation of prompts?

Solutions to these issues involve both general prompt designs and more advanced techniques, such as CoT and prompt learning, which have been explored extensively in recent research.

In NLP, prompting can be viewed as a technology that has evolved along with LLMs, and in a sense, it has opened the door to the practical application of these models in an impressive range of problem domains. In fact, if we expand the concept of prompts to some extent, it can be traced back to the early days of machine learning and NLP. For example, many NLP systems use hand-crafted features and templates to "prompt" specific tasks. Imagine developing a feature to indicate whether a text is formal or informal. We can feed this feature into a machine translation system to condition the translation on the type of the input text.

The widespread use of the modern concept of prompts began with the rise of large pre-trained models in the field of NLP. Initially, these models, such as BERT, were adapted to specific downstream tasks mainly through fine-tuning. However, researchers soon discovered that by designing specific "prompts" — adding certain words or sentences to the input — the models could be triggered to respond to specific tasks without extensive fine-tuning. This motivated the NLP community to develop and apply universal foundation models that can be prompted to address various tasks without changing the underlying architecture and the pre-training procedure.

Prompting approaches were first experimented with smaller models and later demonstrated impressive capabilities with large models like GPT-3, which could generate high-quality text in response to simple prompts across various tasks. As prompting technology evolved, prompt engineering emerged as a critical area of research. As discussed in this chapter, it broadly involves designing effective prompts to maximize model performance, encompassing both hand-crafted and automatically generated prompts. More recent research has explored how to enhance the effectiveness of prompting through techniques like few-shot learning, zero-shot learning, and CoT reasoning, enabling LLMs to work effectively across a wide range of scenarios. A general discussion of prompting can be very broad, and we cannot cover all details in this chapter. For more advanced techniques of prompting, the reader can refer to recent surveys. Topics include in-context learning [Li, 2023; Dong et al., 2022], CoT [Chu et al., 2023; Yu et al., 2023; Zhang et al., 2023a], efficient prompting [Chang et al., 2024], and general prompt engineering [Liu et al., 2023c; Chen et al., 2023].

Note that although we would ideally like to develop general prompting methods without adjusting model architectures and parameters, the results of prompting generally depend heavily on the quality and size of the given LLMs. For stronger models, such as commercialized online LLMs, simple prompts may be sufficient to instruct these models to perform tasks correctly. In this case, prompt engineering is relatively easy, though we still need certain efforts to make LLMs work properly. By contrast, if the LLMs are not powerful enough, we may need to carefully design the prompts to achieve the desired results. In many cases, fine-tuning is still necessary to adapt the models to sophisticated prompting strategies.

Bibliography

[Akyürek et al., 2023] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.

[Andreas et al., 2016] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.

[Askell et al., 2021] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.

[Besta et al., 2024] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.

[Brill, 1992] Eric Brill. A simple rule-based part of speech tagger. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.

[Chang et al., 2024] Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077*, 2024.

[Chen et al., 2023] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023.

[Chevalier et al., 2023] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3829–3846, 2023.

[Chowdhery et al., 2022] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M.

Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[Chu et al., 2023] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*, 2023.

[Cobbe et al., 2021] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[Dai et al., 2023] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4005–4019, 2023.

[Deng et al., 2022] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3369–3391, 2022.

[Dong et al., 2022] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[Drozdov et al., 2022] Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. Compositional semantic parsing with large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2022.

[Dua et al., 2022] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1251–1265, 2022.

[Elsken et al., 2019] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[Fernandes et al., 2023] Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. Bridging the gap: A survey on integrating (human) feedback for natural language generation. *Transactions of the Association for Computational Linguistics*, 11: 1643–1668, 2023.

[Franklin and Graesser, 1996] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International workshop on agent theories, architectures, and languages*, pages 21–35. Springer, 1996.

[Frensch and Funke, 2014] Peter A Frensch and Joachim Funke. *Complex problem solving: The European perspective*. Psychology Press, 2014.

[Ganguli et al., 2023] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamile Lukosiute, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, Dawn

Drain, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jackson Kernion, Jamie Kerr, Jared Mueller, Joshua Landau, Kamal Ndousse, Karina Nguyen, Liane Lovitt, Michael Sellitto, Nelson Elhage, Noemí Mercado, Nova DasSarma, Oliver Rausch, Robert Lasenby, Robin Larson, Sam Ringer, Sandipan Kundu, Saurav Kadavath, Scott Johnston, Shauna Kravec, Sheer El Showk, Tamera Lanham, Timothy Telleen-Lawton, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, Christopher Olah, Jack Clark, Samuel R. Bowman, and Jared Kaplan. The capacity for moral self-correction in large language models. *arXiv preprint arXiv:2302.07459*, 2023.

[Gao et al., 2023] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023a.

[Gao et al., 2023] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023b.

[Garg et al., 2022] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.

[Guo et al., 2024] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*, 2024.

[Hendrycks et al., 2021] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *Proceedings of International Conference on Learning Representations*, 2021.

[Jiang et al., 2023] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, 2023.

[Jiang et al., 2020] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8: 423–438, 2020.

[Kahneman, 2011] Daniel Kahneman. *Thinking, fast and slow*. macmillan, 2011.

[Khot et al., 2023] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.

[Kojima et al., 2022] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[Lake and Baroni, 2018] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.

[Lester et al., 2021] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, 2021.

[Li et al., 2023] Bei Li, Rui Wang, Junliang Guo, Kaitao Song, Xu Tan, Hany Hassan, Arul Menezes, Tong Xiao, Jiang Bian, and JingBo Zhu. Deliberate then generate: Enhanced prompting framework for text generation. *arXiv preprint arXiv:2305.19835*, 2023a.

[Li et al., 2022] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110*, 2022.

[Li and Liang, 2021] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.

[Li, 2023] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, pages 641–647, 2023.

[Li et al., 2023] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353, 2023b.

[Lialin et al., 2023] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023.

[Lightman et al., 2024] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

[Liu et al., 2022] Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, 2022.

[Liu et al., 2023] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023a.

[Liu et al., 2023] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023b.

[Liu et al., 2023] Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. *arXiv preprint arXiv:2311.12785*, 2023c.

[Madaan et al., 2024] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[Marcus, 1993] Gary F Marcus. Negative evidence in language acquisition. *Cognition*, 46(1):53–85, 1993.

[Mavi et al., 2024] Vaibhav Mavi, Anubhav Jangra, and Adam Jatowt. Multi-hop question answering. *Foundations and Trends® in Information Retrieval*, 17(5):457–586, 2024.

[Min et al., 2019] Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. Multi-hop

reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109, 2019.

[Mu et al., 2024] Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36, 2024.

[Pan et al., 2024] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024.

[Parisi et al., 2022] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[Prasad et al., 2023] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3845–3864, 2023.

[Press et al., 2023] Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, 2023.

[Pryzant et al., 2023] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[Rubin et al., 2022] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, 2022.

[Schick et al., 2023] Timo Schick, Jane A. Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.

[Schick et al., 2024] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

[Snell et al., 2022] Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. *arXiv preprint arXiv:2209.15189*, 2022.

[Talmor and Berant, 2018] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. *arXiv preprint arXiv:1803.06643*, 2018.

[Uesato et al., 2022] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

[Von Oswald et al., 2023] Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *Proceedings of International Conference on Machine Learning*, pages 35151–35174. PMLR, 2023.

[Wang et al., 2022] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747*, 2022.

[Wang et al., 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.

[Wei et al., 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[Welleck et al., 2023] Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.

[Wingate et al., 2022] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. Prompt compression and contrastive conditioning for controllability and toxicity reduction in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5621–5634, 2022.

[Xiao et al., 2013] Tong Xiao, Jingbo Zhu, and Tongran Liu. Bagging and boosting statistical machine translation systems. *Artificial Intelligence*, 195:496–527, 2013.

[Xie et al., 2022] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In *Proceedings of International Conference on Learning Representations*, 2022.

[Yao et al., 2024] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[Yu et al., 2023] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. Towards better chain-of-thought prompting strategies: A survey. *arXiv preprint arXiv:2310.04959*, 2023.

[Zhang et al., 2023] Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and Hai Zhao. Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents. *arXiv preprint arXiv:2311.11797*, 2023a.

[Zhang et al., 2023] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*, 2023b.

[Zhou et al., 2023] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023a.

[Zhou et al., 2023] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023b.

[Zoph and Le, 2016] Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *Proceedings of International Conference on Learning Representations*, 2016.