Foundations of Large Language Models

Tong Xiao and Jingbo Zhu

March 24, 2025

NLP Lab, Northeastern University & NiuTrans Research

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

March 24, 2025

Notation

a	variable
$\mathbf{a}$	row vector or matrix
$f(a)$	function of a
$\max f(a)$	maximum value of $f(a)$
$\arg\max_a f(a)$	value of a that maximizes $f(a)$
$\mathbf{x}$	input token sequence to a model
x_j	input token at position j
$\mathbf{y}$	output token sequence produced by a model
y_i	output token at position i
θ	model parameters
$\Pr(a)$	probability of a
$\Pr(a\|b)$	conditional probability of a given b
$\Pr(\cdot\|b)$	probability distribution of a variable given b
$\Pr_\theta(a)$	probability of a as parameterized by θ
$\mathbf{h}_t$	hidden state at time step t in sequential models
$\mathbf{H}$	matrix of all hidden states over time in a sequence
$\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$	query, key, and value matrices in attention mechanisms
$\mathrm{Softmax}(\mathbf{A})$	Softmax function that normalizes the input vector or matrix $\mathbf{A}$
$\mathcal{L}$	loss function
$\mathcal{D}$	dataset used for training or fine-tuning a model
$\frac{\partial \mathcal{L}}{\partial \theta}$	gradient of the loss function $\mathcal{L}$ with respect to the parameters θ
$\mathrm{KL}(p \,\|\|\, q)$	KL divergence between distributions p and q

Contents

CHAPTER 1

Pre-training

The development of neural sequence models, such as **Transformers** [Vaswani et al., 2017], along with the improvements in large-scale self-supervised learning, has opened the door to universal language understanding and generation. This achievement is largely motivated by pre-training: we separate common components from many neural network-based systems, and then train them on huge amounts of unlabeled data using self-supervision. These pre-trained models serve as foundation models that can be easily adapted to different tasks via fine-tuning or prompting. As a result, the paradigm of NLP has been enormously changed. In many cases, large-scale supervised learning for specific tasks is no longer required, and instead, we only need to adapt pre-trained foundation models.

While pre-training has gained popularity in recent NLP research, this concept dates back decades to the early days of deep learning. For example, early attempts to pre-train deep learning systems include unsupervised learning for RNNs, deep feedforward networks, autoencoders, and others [Schmidhuber, 2015]. In the modern era of deep learning, we experienced a resurgence of pre-training, caused in part by the large-scale unsupervised learning of various word embedding models [Mikolov et al., 2013b; Pennington et al., 2014]. During the same period, pre-training also attracted significant interest in computer vision, where the backbone models were trained on relatively large labeled datasets such as ImageNet, and then applied to different downstream tasks [He et al., 2019; Zoph et al., 2020]. Large-scale research on pre-training in NLP began with the development of language models using self-supervised learning. This family of models covers several well-known examples like **BERT** [Devlin et al., 2019] and **GPT** [Brown et al., 2020], all with a similar idea that general language understanding and generation can be achieved by training the models to predict masked words in a huge amount of text. Despite the simple nature of this approach, the resulting models show remarkable capability in modeling linguistic structure, though they are not explicitly trained to achieve this. The generality of the pre-training tasks leads to systems that exhibit strong performance in a large variety of NLP problems, even outperforming previously well-developed supervised systems. More recently, pre-trained large language models have achieved a greater success, showing the exciting prospects for more general artificial intelligence [Bubeck et al., 2023].

This chapter discusses the concept of pre-training in the context of NLP. It begins with a general introduction to pre-training methods and their applications. BERT is then used as an example to illustrate how a sequence model is trained via a self-supervised task, called **masked language modeling**. This is followed by a discussion of methods for adapting pre-trained sequence models for various NLP tasks. Note that in this chapter, we will focus primarily on the pre-training paradigm in NLP, and therefore, we do not intend to cover details about generative large language models. A detailed discussion of these models will be left to subsequent chapters.

1.1 Pre-training NLP Models

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation. While these problems have different

forms, for simplicity, we describe them using a single model defined as follows:

$$\begin{aligned} \mathbf{o} &= g(x_0, x_1, ..., x_m; \theta) \\ &= g_\theta(x_0, x_1, ..., x_m) \end{aligned} \tag{1.1}$$

where $\{x_0, x_1, ..., x_m\}$ denotes a sequence of input tokens[1], x_0 denotes a special symbol ($\langle s \rangle$ or [CLS]) attached to the beginning of a sequence, $g(\cdot; \theta)$ (also written as $g_\theta(\cdot)$) denotes a neural network with parameters θ, and $\mathbf{o}$ denotes the output of the neural network. Different problems can vary based on the form of the output $\mathbf{o}$. For example, in token prediction problems (as in language modeling), $\mathbf{o}$ is a distribution over a vocabulary; in sequence encoding problems, $\mathbf{o}$ is a representation of the input sequence, often expressed as a real-valued vector sequence.

There are two fundamental issues here.

- Optimizing θ on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.
- Applying the pre-trained model $g_{\hat{\theta}}(\cdot)$ to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters $\hat{\theta}$ slightly using labeled data or prompt the model with task descriptions.

In this section, we discuss the basic ideas in addressing these issues.

1.1.1 Unsupervised, Supervised and Self-supervised Pre-training

In deep learning, pre-training refers to the process of optimizing a neural network before it is further trained/tuned and applied to the tasks of interest. This approach is based on an assumption that a model pre-trained on one task can be adapted to perform another task. As a result, we do not need to train a deep, complex neural network from scratch on tasks with limited labeled data. Instead, we can make use of tasks where supervision signals are easier to obtain. This reduces the reliance on task-specific labeled data, enabling the development of more general models that are not confined to particular problems.

During the resurgence of neural networks through deep learning, many early attempts to achieve pre-training were focused on **unsupervised learning**. In these methods, the parameters of a neural network are optimized using a criterion that is not directly related to specific tasks. For example, we can minimize the reconstruction cross-entropy of the input vector for each layer [Bengio et al., 2006]. Unsupervised pre-training is commonly employed as a preliminary step before supervised learning, offering several advantages, such as aiding in the discovery of better local minima and adding a regularization effect to the training process [Erhan et al., 2010]. These benefits make the subsequent supervised learning phase easier and more stable.

A second approach to pre-training is to pre-train a neural network on **supervised learning** tasks. For example, consider a sequence model designed to encode input sequences into some

[1] Here we assume that tokens are basic units of text that are separated through tokenization. Sometimes, we will use the terms *token* and *word* interchangeably, though they have closely related but slightly different meanings in NLP.

representations. In pre-training, this model is combined with a classification layer to form a classification system. This system is then trained on a pre-training task, such as classifying sentences based on sentiment (e.g., determining if a sentence conveys a positive or negative sentiment). Then, we adapt the sequence model to a downstream task. We build a new classification system based on this pre-trained sequence model and a new classification layer (e.g., determining if a sequence is subjective or objective). Typically, we need to fine-tune the parameters of the new model using task-specific labeled data, ensuring the model is optimally adjusted to perform well on this new type of data. The fine-tuned model is then employed to classify new sequences for this task. An advantage of supervised pre-training is that the training process, either in the pre-training or fine-tuning phase, is straightforward, as it follows the well-studied general paradigm of supervised learning in machine learning. However, as the complexity of the neural network increases, the demand for more labeled data also grows. This, in turn, makes the pre-training task more difficult, especially when large-scale labeled data is not available.

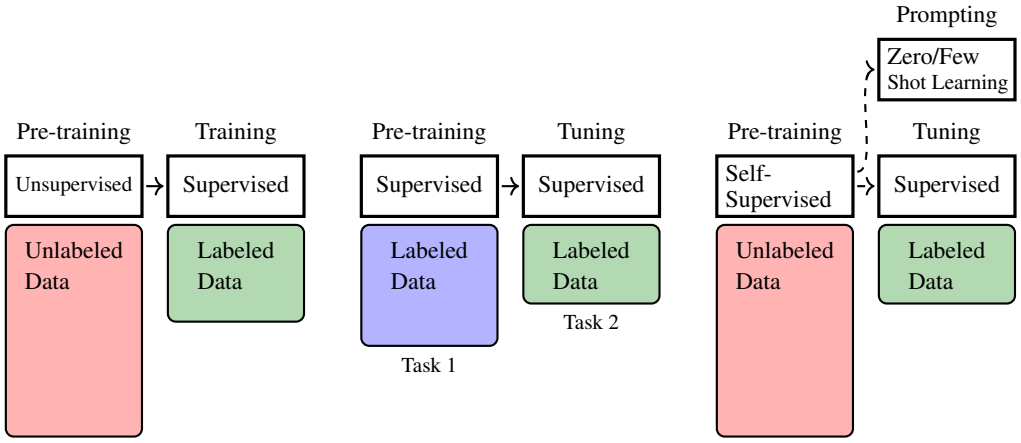
A third approach to pre-training is **self-supervised learning**. In this approach, a neural network is trained using the supervision signals generated by itself, rather than those provided by humans. This is generally done by constructing its own training tasks directly from unlabeled data, such as having the system create pseudo labels. While self-supervised learning has recently emerged as a very popular method in NLP, it is not a new concept. In machine learning, a related concept is **self-training** where a model is iteratively improved by learning from the pseudo labels assigned to a dataset. To do this, we need some seed data to build an initial model. This model then generates pseudo labels for unlabeled data, and these pseudo labels are subsequently used to iteratively refine and bootstrap the model itself. Such a method has been successfully used in several NLP areas, such as word sense disambiguation [Yarowsky, 1995] and document classification [Blum and Mitchell, 1998]. Unlike the standard self-training method, self-supervised pre-training in NLP does not rely on an initial model for annotating the data. Instead, all the supervision signals are created from the text, and the entire model is trained from scratch. A well-known example of this is training sequence models by successively predicting a masked word given its preceding or surrounding words in a text. This enables large-scale self-supervised learning for deep neural networks, leading to the success of pre-training in many understanding, writing, and reasoning tasks.

Figure 1.1 shows a comparison of the above three pre-training approaches. Self-supervised pre-training is so successful that most current state-of-the-art NLP models are based on this paradigm. Therefore, in this chapter and throughout this book, we will focus on self-supervised pre-training. We will show how sequence models are pre-trained via self-supervision and how the pre-trained models are applied.

1.1.2 Adapting Pre-trained Models

As mentioned above, two major types of models are widely used in NLP pre-training.

- **Sequence Encoding Models**. Given a sequence of words or tokens, a sequence encoding model represents this sequence as either a real-valued vector or a sequence of vectors, and obtains a representation of the sequence. This representation is typically used as input to another model, such as a sentence classification system.



(a) Unsupervised Pre-training (b) Supervised Pre-training (c) Self-supervised Pre-training

Fig. 1.1: Illustration of unsupervised, supervised, and self-supervised pre-training. In unsupervised pre-training, the pre-training is performed on large-scale unlabeled data. It can be viewed as a preliminary step to have a good starting point for the subsequent optimization process, though considerable effort is still required to further train the model with labeled data after pre-training. In supervised pre-training, the underlying assumption is that different (supervised) learning tasks are related. So we can first train the model on one task, and transfer the resulting model to another task with some training or tuning effort. In self-supervised pre-training, a model is pre-trained on large-scale unlabeled data via self-supervision. The model can be well trained in this way, and we can efficiently adapt it to new tasks through fine-tuning or prompting.

- **Sequence Generation Models**. In NLP, sequence generation generally refers to the problem of generating a sequence of tokens based on a given context. The term *context* has different meanings across applications. For example, it refers to the preceding tokens in language modeling, and refers to the source-language sequence in machine translation[2].

We need different techniques for applying these models to downstream tasks after pre-training. Here we are interested in the following two methods.

1.1.2.1 Fine-tuning of Pre-trained Models

For sequence encoding pre-training, a common method of adapting pre-trained models is fine-tuning. Let $\text{Encode}_\theta(\cdot)$ denote an encoder with parameters θ, for example, $\text{Encode}_\theta(\cdot)$ can be a standard Transformer encoder. Provided we have pre-trained this model in some way and obtained the optimal parameters $\hat{\theta}$, we can employ it to model any sequence and generate the corresponding representation, like this

$$\mathbf{H} \quad = \quad \text{Encode}_{\hat{\theta}}(\mathbf{x}) \tag{1.2}$$

where $\mathbf{x}$ is the input sequence $\{x_0, x_1, ..., x_m\}$, and $\mathbf{H}$ is the output representation which is a sequence of real-valued vectors $\{\mathbf{h}_0, \mathbf{h}_1, ..., \mathbf{h}_m\}$. Because the encoder does not work as a standalone NLP system, it is often integrated as a component into a bigger system. Consider, for example, a text classification problem in which we identify the polarity (i.e., positive, negative,

[2]More precisely, in auto-regressive decoding of machine translation, each target-language token is generated based on both its preceding tokens and source-language sequence.

and neutral) of a given text. We can build a text classification system by stacking a classifier on top of the encoder. Let $\mathrm{Classify}_{\omega}(\cdot)$ be a neural network with parameters ω. Then, the text classification model can be expressed in the form

$$\begin{aligned} \mathrm{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x}) &= \mathrm{Classify}_{\omega}(\mathbf{H}) \\ &= \mathrm{Classify}_{\omega}(\mathrm{Encode}_{\hat{\theta}}(\mathbf{x})) \end{aligned} \tag{1.3}$$

Here $\mathrm{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x})$ is a probability distribution over the label set $\{\mathrm{positive}, \mathrm{negative}, \mathrm{neutral}\}$, and the label with the highest probability in this distribution is selected as output. To keep the notation uncluttered, we will use $F_{\omega,\hat{\theta}}(\cdot)$ to denote $\mathrm{Classify}_{\omega}(\mathrm{Encode}_{\hat{\theta}}(\cdot))$.

Because the model parameters ω and $\hat{\theta}$ are not optimized for the classification task, we cannot directly use this model. Instead, we must use a modified version of the model that is adapted to the task. A typical way is to fine-tune the model by giving explicit labeling in downstream tasks. We can train $F_{\omega,\hat{\theta}}(\cdot)$ on a labeled dataset, treating it as a common supervised learning task. The outcome of the fine-tuning is the parameters $\tilde{\omega}$ and $\tilde{\theta}$ that are further optimized. Alternatively, we can freeze the encoder parameters $\hat{\theta}$ to maintain their pre-trained state, and focus solely on optimizing ω. This allows the classifier to be efficiently adapted to work in tandem with the pre-trained encoder.

Once we have obtained a fine-tuned model, we can use it to classify a new text. For example, suppose we have a comment posted on a travel website:

I love the food here. It's amazing!

We first tokenize this text into tokens[3], and then feed the token sequence $\mathbf{x}_{\mathrm{new}}$ into the fine-tuned model $F_{\tilde{\omega},\tilde{\theta}}(\cdot)$. The model generates a distribution over classes by

$$F_{\tilde{\omega},\tilde{\theta}}(\mathbf{x}_{\mathrm{new}}) = \begin{bmatrix} \mathrm{Pr}(\mathrm{positive}|\mathbf{x}_{\mathrm{new}}) & \mathrm{Pr}(\mathrm{negative}|\mathbf{x}_{\mathrm{new}}) & \mathrm{Pr}(\mathrm{neutral}|\mathbf{x}_{\mathrm{new}}) \end{bmatrix} \tag{1.4}$$

And we select the label of the entry with the maximum value as output. In this example it is $\mathrm{positive}$.

In general, the amount of labeled data used in fine-tuning is small compared to that of the pre-training data, and so fine-tuning is less computationally expensive. This makes the adaption of pre-trained models very efficient in practice: given a pre-trained model and a downstream task, we just need to collect some labeled data, and slightly adjust the model parameters on this data.

1.1.2.2 Prompting of Pre-trained Models

Unlike sequence encoding models, sequence generation models are often employed independently to address language generation problems, such as question answering and machine translation, without the need for additional modules. It is therefore straightforward to fine-tune these models as complete systems on downstream tasks. For example, we can fine-tune a pre-trained encoder-decoder multilingual model on some bilingual data to improve its performance on a specific translation task.

[3]The text can be tokenized in many different ways. One of the simplest is to segment the text into English words and punctuations $\{\mathrm{I}, \mathrm{love}, \mathrm{the}, \mathrm{food}, \mathrm{here}, ., \mathrm{It}, \mathrm{'s}, \mathrm{amazing}, !\}$

Among various sequence generation models, a notable example is the large language models trained on very large amounts of data. These language models are trained to simply predict the next token given its preceding tokens. Although token prediction is such a simple task that it has long been restricted to "language modeling" only, it has been found to enable the learning of the general knowledge of languages by repeating the task a large number of times. The result is that the pre-trained large language models exhibit remarkably good abilities in token prediction, making it possible to transform numerous NLP problems into simple text generation problems through prompting the large language models. For example, we can frame the above text classification problem as a text generation task

I love the food here. It's amazing! I'm _____

Here __ indicates the word or phrase we want to predict (call it the **completion**). If the predicted word is *happy*, or *glad*, or *satisfied* or a related positive word, we can classify the text as positive. This example shows a simple prompting method in which we concatenate the input text with *I'm* to form a prompt. Then, the completion helps decide which label is assigned to the original text.

Given the strong performance of language understanding and generation of large language models, a prompt can instruct the models to perform more complex tasks. Here is a prompt where we prompt the LLM to perform polarity classification with an instruction.

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: I love the food here. It's amazing!
>
> Polarity: _____

The first two sentences are a description of the task. Input and Polarity are indicators of the input and output, respectively. We expect the model to complete the text and at the same time give the correct polarity label. By using instruction-based prompts, we can adapt large language models to solve NLP problems without the need for additional training.

This example also demonstrates the zero-shot learning capability of large language models, which can perform tasks that were not observed during the training phase. Another method for enabling new capabilities in a neural network is few-shot learning. This is typically achieved through **in-context learning** (**ICT**). More specifically, we add some samples that demonstrate how an input corresponds to an output. These samples, known as **demonstrations**, are used to teach large language models how to perform the task. Below is an example involving demonstrations

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: The traffic is terrible during rush hours, making it difficult to reach the airport on time.
>
> Polarity: Negative
>
> Input: The weather here is wonderful.
>
> Polarity: Positive
>
> Input: I love the food here. It's amazing!
>
> Polarity: _____

Prompting and in-context learning play important roles in the recent rise of large language models. We will discuss these issues more deeply in Chapter 3. However, it is worth noting that while prompting is a powerful way to adapt large language models, some tuning efforts are still needed to ensure the models can follow instructions accurately. Additionally, the fine-tuning process is crucial for aligning the values of these models with human values. More detailed discussions of fine-tuning can be found in Chapter 4.

1.2 Self-supervised Pre-training Tasks

In this section, we consider self-supervised pre-training approaches for different neural architectures, including decoder-only, encoder-only, and encoder-decoder architectures. We restrict our discussion to Transformers since they form the basis of most pre-trained models in NLP. However, pre-training is a broad concept, and so we just give a brief introduction to basic approaches in order to make this section concise.

1.2.1 Decoder-only Pre-training

The decoder-only architecture has been widely used in developing language models [Radford et al., 2018]. For example, we can use a Transformer decoder as a language model by simply removing cross-attention sub-layers from it. Such a model predicts the distribution of tokens at a position given its preceding tokens, and the output is the token with the maximum probability. The standard way to train this model, as in the language modeling problem, is to minimize a loss function over a collection of token sequences. Let $\text{Decoder}_\theta(\cdot)$ denote a decoder with parameters θ. At each position i, the decoder generates a distribution of the next tokens based on its preceding tokens $\{x_0, ..., x_i\}$, denoted by $\Pr_\theta(\cdot|x_0, ..., x_i)$ (or $\mathbf{p}^\theta_{i+1}$ for short). Suppose we have the gold-standard distribution at the same position, denoted by $\mathbf{p}^{\text{gold}}_{i+1}$. For language modeling, we can think of $\mathbf{p}^{\text{gold}}_{i+1}$ as a one-hot representation of the correct predicted word. We then define a loss function $\mathcal{L}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1})$ to measure the difference between the model prediction and the true prediction. In NLP, the log-scale cross-entropy loss is typically used.

Given a sequence of m tokens $\{x_0, ..., x_m\}$, the loss on this sequence is the sum of the loss

over the positions $\{0, ..., m-1\}$, given by

$$\begin{aligned} \text{Loss}_\theta(x_0, ..., x_m) &= \sum_{i=0}^{m-1} \mathcal{L}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \\ &= \sum_{i=0}^{m-1} \text{LogCrossEntropy}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \end{aligned} \tag{1.5}$$

where $\text{LogCrossEntropy}(\cdot)$ is the log-scale cross-entropy, and $\mathbf{p}_{i+1}^{\text{gold}}$ is the one-hot representation of x_{i+1}.

This loss function can be extended to a set of sequences $\mathcal{D}$. In this case, the objective of pre-training is to find the best parameters that minimize the loss on $\mathcal{D}$

$$\hat{\theta} = \arg\min_\theta \sum_{\mathbf{x} \in \mathcal{D}} \text{Loss}_\theta(\mathbf{x}) \tag{1.6}$$

Note that this objective is mathematically equivalent to maximum likelihood estimation, and can be re-expressed as

$$\begin{aligned} \hat{\theta} &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \log \Pr_\theta(\mathbf{x}) \\ &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=0}^{i-1} \log \Pr_\theta(x_{i+1} | x_0, ..., x_i) \end{aligned} \tag{1.7}$$

With these optimized parameters $\hat{\theta}$, we can use the pre-trained language model $\text{Decoder}_{\hat{\theta}}(\cdot)$ to compute the probability $\Pr_{\hat{\theta}}(x_{i+1} | x_0, ..., x_i)$ at each position of a given sequence.

CHAPTER 2

Generative Models

One of the most significant advances in NLP in recent years might be the development of large language models (LLMs). This has helped create systems that can understand and generate natural languages like humans. These systems have even been found to be able to reason, which is considered a very challenging AI problem. With these achievements, NLP made big strides and entered a new era of research in which difficult problems are being solved, such as building conversational systems that can communicate with humans smoothly.

The concept of language modeling or probabilistic language modeling dates back to early experiments conducted by Shannon [1951]. In his work, a language model was designed to estimate the predictability of English — *how well can the next letter of a text be predicted when the preceding N letters are known*. Although Shannon's experiments were preliminary, the fundamental goals and methods of language modeling have remained largely unchanged over the decades since then. For quite a long period, particularly before 2010, the dominant approach to language modeling was the n-gram approach [Jurafsky and Martin, 2008]. In n-gram language modeling, we estimate the probability of a word given its preceding $n-1$ words, and thus the probability of a sequence can be approximated by the product of a series of n-gram probabilities. These probabilities are typically estimated by collecting smoothed relative counts of n-grams in text. While such an approach is straightforward and simple, it has been extensively used in NLP. For example, the success of modern statistical speech recognition and machine translation systems has largely depended on the utilization of n-gram language models [Jelinek, 1998; Koehn, 2010].

Applying neural networks to language modeling has long been attractive, but a real breakthrough appeared as deep learning techniques advanced. A widely cited study is Bengio et al. [2003]'s work where n-gram probabilities are modeled via a feed-forward network and learned by training the network in an end-to-end fashion. A by-product of this neural language model is the distributed representations of words, known as word embeddings. Rather than representing words as discrete variables, word embeddings map words into low-dimensional real-valued vectors, making it possible to compute the meanings of words and word n-grams in a continuous representation space. As a result, language models are no longer burdened with the curse of dimensionality, but can represent exponentially many n-grams via a compact and dense neural model.

The idea of learning word representations through neural language models inspired subsequent research in representation learning in NLP. However, this approach did not attract significant interest in developing NLP systems in the first few years after its proposal. Starting in about 2012, though, advances were made in learning word embeddings from large-scale text via simple word prediction tasks. Several methods, such as Word2Vec, were proposed to effectively learn such embeddings, which were then successfully applied in a variety of NLP systems [Mikolov et al., 2013a;b]. As a result of these advances, researchers began to think of learning representations of sequences using more powerful language models, such as LSTM-based models [Sutskever et al., 2014; Peters et al., 2018]. And further progress and interest in sequence representation exploded after Transformer was proposed. Alongside the rise of Transformer, the concept of language modeling was generalized to encompass models that learn to predict words in various ways. Many

powerful Transformer-based models were pre-trained using these word prediction tasks, and successfully applied to a variety of downstream tasks [Devlin et al., 2019].

Indeed, training language models on large-scale data has led NLP research to exciting times. While language modeling has long been seen as a foundational technique with no direct link to the goals of artificial intelligence that researchers had hoped for, it helps us see the emergence of intelligent systems that can learn a certain degree of general knowledge from repeatedly predicting words in text. Recent research demonstrates that a single, well-trained LLM can handle a large number of tasks and generalize to perform new tasks with a small adaptation effort [Bubeck et al., 2023]. This suggests a step towards more advanced forms of artificial intelligence, and inspires further exploration into developing more powerful language models as foundation models.

In this chapter, we consider the basic concepts of generative LLMs. For simplicity, we use the terms *large language models* or *LLMs* to refer to generative models like GPT, though this term can broadly cover other types of models like BERT. We begin by giving a general introduction to LLMs, including the key steps of building such models. We then discuss two scaling issues of LLMs: how LLMs are trained at scale, and how LLMs can be improved to handle very long texts. Finally, we give a summary of these discussions.

2.1 A Brief Introduction to LLMs

In this section we give an introduction to the basic ideas of LLMs as required for the rest of this chapter and the following chapters. We will use terms *word* and *token* interchangeably. Both of them refer to the basic units used in language modeling, though their original meanings are different.

Before presenting details, let us first consider how language models work. The goal of language modeling is to predict the probability of a sequence of tokens occurring. Let $\{x_0, x_1, ..., x_m\}$ be a sequence of tokens, where x_0 is the start symbol $\langle s \rangle$ (or $\langle \text{SOS} \rangle$)[1]. The probability of this sequence can be defined using the chain rule

$$\begin{aligned} \Pr(x_0, ..., x_m) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdots \Pr(x_m|x_0, ..., x_{m-1}) \\ &= \prod_{i=0}^{m} \Pr(x_i|x_0, ..., x_{i-1}) \end{aligned} \tag{2.1}$$

or alternatively in a logarithmic form

$$\log \Pr(x_0, ..., x_m) = \sum_{i=0}^{m} \log \Pr(x_i|x_0, ..., x_{i-1}) \tag{2.2}$$

Here $\Pr(x_i|x_0, ..., x_{i-1})$ is the probability of the token x_i given all its previous tokens $\{x_0, ..., x_{i-1}\}$ [2]. In the era of deep learning, a typical approach to language modeling is to estimate this

[1] The start symbol can also be [CLS] following BERT models.

[2] We assume that when $i = 0$, $\Pr(x_i|x_0, ..., x_{i-1}) = \Pr(x_0) = 1$. Hence $\Pr(x_0, ..., x_m) = \Pr(x_0) \Pr(x_1, ..., x_m|x_0) = \Pr(x_1, ..., x_m|x_0)$.

Context	Predict	Decision Rule	Sequence Probability
$\langle s\rangle\ a$	b	$\arg\max_{x_2\in V}\Pr(x_2\|\langle s\rangle\ a)$	$\Pr(\langle s\rangle)\cdot\Pr(a\|\langle s\rangle)\cdot\ \Pr(b\|\langle s\rangle\ a)$
$\langle s\rangle\ a\ b$	c	$\arg\max_{x_3\in V}\Pr(x_3\|\langle s\rangle\ a\ b)$	$\Pr(\langle s\rangle)\cdot\Pr(a\|\langle s\rangle)\cdot\Pr(b\|\langle s\rangle\ a)\cdot$ $\Pr(c\|\langle s\rangle\ a\ b)$
$\langle s\rangle\ a\ b\ c$	d	$\arg\max_{x_4\in V}\Pr(x_4\|\langle s\rangle\ a\ b\ c)$	$\Pr(\langle s\rangle)\cdot\Pr(a\|\langle s\rangle)\cdot\Pr(b\|\langle s\rangle\ a)\cdot$ $\Pr(c\|\langle s\rangle\ a\ b)\cdot\ \Pr(d\|\langle s\rangle\ a\ b\ c)$

Table 2.1: Illustration of generating the three tokens $b\ c\ d$ given the prefix $\langle s\rangle\ a$ via a language model. In each step, the model picks a token x_i from V so that $\Pr(x_i|x_0,...,x_{i-1})$ is maximized. This token is then appended to the end of the context sequence. In the next step, we repeat the same process, but based on the new context.

probability using a deep neural network. Neural networks trained to accomplish this task receive a sequence of tokens $x_0,...,x_{i-1}$ and produce a distribution over the vocabulary $\mathcal{V}$ (denoted by $\Pr(\cdot|x_0,...,x_{i-1})$). The probability $\Pr(x_i|x_0,...,x_{i-1})$ is the value of the i-th entry of $\Pr(\cdot|x_0,...,x_{i-1})$.

When applying a trained language model, a common task is to find the most likely token given its previous context tokens. This token prediction task can be described as

$$\hat{x}_i \quad = \quad \arg\max_{x_i\in\mathcal{V}} \Pr(x_i|x_0,...,x_{i-1}) \tag{2.3}$$

We can perform word prediction multiple times to generate a continuous text: each time we predict the best token $\hat{x}_i$, and then add this predicted token to the context for predicting the next token $\hat{x}_{i+1}$. This results in a left-to-right generation process implementing Eqs. (2.1) and (2.2). To illustrate, consider the generation of the following three words given the prefix '$\langle s\rangle\ a$', as shown in Table 2.1. Now we discuss how LLMs are constructed, trained, and applied.

2.1.1 Decoder-only Transformers

As is standard practice, the input of a language model is a sequence of tokens (denoted by $\{x_0,...,x_{m-1}\}$). For each step, an output token is generated, shifting the sequence one position forward for the next prediction. To do this, the language model outputs a distribution $\Pr(\cdot|x_0,...,x_{i-1})$ at each position i, and the token x_i is selected according to this distribution. This model is trained by maximizing the log likelihood $\sum_{i=1}^{m}\log\Pr(x_i|x_0,...,x_{i-1})$[3].

Here, we focus on the decoder-only Transformer architecture, as it is one of the most popular model architectures used in LLMs. The input sequence of tokens is represented by a sequence of d_e-dimensional vectors $\{\mathbf{e}_0,...,\mathbf{e}_{m-1}\}$. $\mathbf{e}_i$ is the sum of the token embedding of x_i and the positional embedding of i. The major body of the model is a stack of Transformer blocks (or layers). Each Transformer block has two stacked sub-layers, one for self-attention modeling and one for FFN modeling. These sub-layers can be defined using the post-norm architecture

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input})+\text{input}) \tag{2.4}$$

[3]Note that $\sum_{i=1}^{m}\log\Pr(x_i|x_0,...,x_{i-1}) = \sum_{i=0}^{m}\log\Pr(x_i|x_0,...,x_{i-1})$ since $\log\Pr(x_0)=0$.

or the pre-norm architecture

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input})) + \text{input} \tag{2.5}$$

where input and output denote the input and output, both being an $m \times d$ matrix. The i-th rows of input and output can be seen as contextual representations of the i-th token in the sequence.

$F(\cdot)$ is the core function of a sub-layer. For FFN sub-layers, $F(\cdot)$ is a multi-layer FFN. For self-attention sub-layers, $F(\cdot)$ is a multi-head self-attention function. In general, self-attention is expressed in a form of QKV attention

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \quad = \quad \text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\text{T}}}{\sqrt{d}} + \mathbf{Mask})\mathbf{V} \tag{2.6}$$

where $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V} \in \mathbb{R}^{m \times d}$ are the queries, keys, and values, respectively. It is important to note that only previous tokens are considered when predicting a token. So a masking variable $\mathbf{Mask} \in \mathbb{R}^{m \times m}$ is incorporated into self-attention to achieve this. The entry (i, k) of $\mathbf{Mask}$ has a value of 0 if $i \leq k$, and a value of $-\inf$ otherwise.

Given a representation $\mathbf{H} \in \mathbb{R}^{m \times d}$, the multi-head self-attention function can be defined as

$$F(\mathbf{H}) \quad = \quad \text{Merge}(\text{head}_1, ..., \text{head}_\tau)\mathbf{W}^{\text{head}} \tag{2.7}$$

where $\text{Merge}(\cdot)$ representees a concatenation of its inputs, and $\mathbf{W}^{\text{head}} \in \mathbb{R}^{d \times d}$ represents a parameter matrix. head_j is the output of QKV attention on a sub-space of representation

$$\text{head}_j \quad = \quad \text{Att}_{\text{qkv}}(\mathbf{Q}^{[j]}, \mathbf{K}^{[j]}, \mathbf{V}^{[j]}) \tag{2.8}$$

$\mathbf{Q}^{[j]}$,$\mathbf{K}^{[j]}$,and $\mathbf{V}^{[j]}$ are the queries, keys, and values projected onto the j-th sub-space via linear transformations

$$\mathbf{Q}^{[j]} \quad = \quad \mathbf{H}\mathbf{W}_j^q \tag{2.9}$$

$$\mathbf{K}^{[j]} \quad = \quad \mathbf{H}\mathbf{W}_j^k \tag{2.10}$$

$$\mathbf{V}^{[j]} \quad = \quad \mathbf{H}\mathbf{W}_j^v \tag{2.11}$$

where $\mathbf{W}_j^q$, $\mathbf{W}_j^k$, and $\mathbf{W}_j^v \in \mathbb{R}^{d \times \frac{d}{\tau}}$ are the parameter matrices of the transformations.

Suppose we have L Transformer blocks. A Softmax layer is built on top of the output of the last block. The Softmax layer outputs a sequence of m distributions over the vocabulary, like this

$$\begin{bmatrix} \Pr(\cdot|x_0, ..., x_{m-1}) \\ \vdots \\ \Pr(\cdot|x_0, x_1) \\ \Pr(\cdot|x_0) \end{bmatrix} \quad = \quad \text{Softmax}(\mathbf{H}^L\mathbf{W}^o) \tag{2.12}$$

where $\mathbf{H}^L$ is the output of the last Transformer block, and $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$ is the parameter matrix.

Figure 2.1 shows the Transformer architecture for language modeling. Applying this language

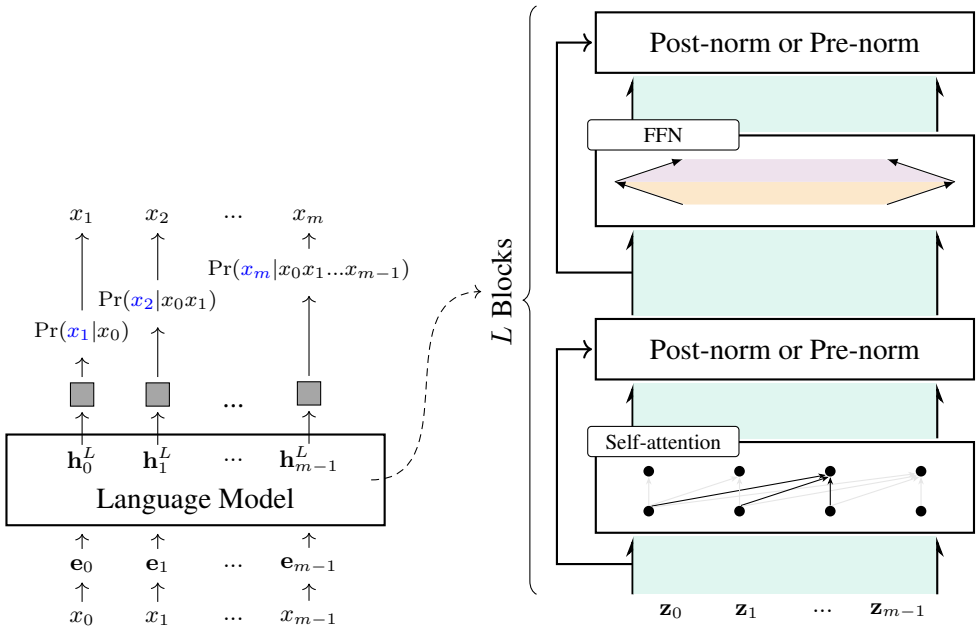


Fig. 2.1: The Transformer-decoder architecture for language modeling. The central components are L stacked Transformer blocks, each comprising a self-attention sub-layer and an FFN sub-layer. To prevent the model from accessing the right-context, a masking variable is incorporated into self-attention. The output layer uses a Softmax function to generate a probability distribution for the next token, given the sequence of previous tokens. During inference, the model takes the previously predicted token to predict the next one, repeating this process until the end of the sequence is reached. $\{\mathbf{z}_0, ..., \mathbf{z}_{m-1}\}$ denote the inputs of a Transformer block, and $\{\mathbf{h}_0^L, ..., \mathbf{h}_{m-1}^L\}$ denote the outputs of the last Transformer block.

model follows an autoregressive process. Each time the language model takes a token x_{i-1} as input and predicts a token x_i that maximizes the probability $\Pr(x_i|x_0, ..., x_{i-1})$. It is important to note that, despite different implementation details, many LLMs share the same architecture described above. These models are called large because both their depth and width are significant. Table 2.2 shows the model sizes for a few LLMs, as well as their model setups.

2.1.2 Training LLMs

Now suppose that we are given a training set $\mathcal{D}$ comprising K sequences. The log-likelihood of each sequence $\mathbf{x} = x_0...x_m$ in $\mathcal{D}$ can be calculated using a language model

$$\mathcal{L}_\theta(\mathbf{x}) \quad = \quad \sum_{i=1}^{m} \log \Pr_\theta(x_i|x_0, ..., x_{i-1}) \tag{2.13}$$

Here the subscript θ affixed to $\mathcal{L}(\cdot)$ and $\Pr(\cdot)$ denotes the parameters of the language model. Then, the objective of maximum likelihood training is defined as

$$\hat{\theta} \quad = \quad \arg\max_\theta \sum_{\mathbf{x}\in\mathcal{D}} \mathcal{L}_\theta(\mathbf{x}) \tag{2.14}$$

Training Transformer-based language models with the above objective is commonly viewed as a standard optimization process for neural networks. This can be achieved using gradient descent algorithms, which are widely supported by off-the-shelf deep learning toolkits. Somewhat

LLM	# of Parameters	Depth L	Width d	# of Heads (Q/KV)
GPT-1 [Radford et al., 2018]	0.117B	12	768	12/12
GPT-2 [Radford et al., 2019]	1.5B	48	1,600	25/25
GPT-3 [Brown et al., 2020]	175B	96	12,288	96/96
LLaMA2 [Touvron et al., 2023b]	7B	32	4,096	32/32
	13B	40	5,120	40/40
	70B	80	8,192	64/64
LLaMA3/3.1 [Dubey et al., 2024]	8B	32	4,096	32/8
	70B	80	8,192	64/8
	405B	126	16,384	128/8
Gemma2 [Team et al., 2024]	2B	26	2,304	8/4
	9B	42	3,584	16/8
	37B	46	4,608	32/16
Qwen2.5 [Yang et al., 2024]	0.5B	24	896	14/2
	7B	28	3,584	28/4
	72B	80	8,192	64/8
DeepSeek-V3 [Liu et al., 2024]	671B	61	7,168	128/128
Falcon [Penedo et al., 2023]	7B	32	4,544	71/71
	40B	60	8,192	128/128
	180B	80	14,848	232/232
Mistral [Jiang et al., 2023]	7B	32	4,096	32/32

Table 2.2: Comparison of some LLMs in terms of model size, model depth, model width, and number of heads (a/b means a heads for queries and b heads for both keys and values).

surprisingly, better results were continuously yielded as language models were evolved into more computationally intensive models and trained on larger datasets [Kaplan et al., 2020]. These successes have led NLP researchers to continue increasing both the training data and model size in order to build more powerful language models.

However, as language models become larger, we confront new training challenges, which significantly change the problem compared to training relatively small models. One of these challenges arises from the need for large-scale distributed systems to manage the data, model parameters, training routines, and so on. Developing and maintaining such systems requires a significant amount of work in both software and hardware engineering, as well as expertise in deep learning. A related issue is that when the training is scaled up, we need more computing resources to ensure the training process can be completed in an acceptable time. For example, it generally requires hundreds or thousands of GPUs to train an LLM with tens of billions of parameters from scratch. This requirement drastically increases the cost of training such models, especially considering that many training runs are needed as these models are developed. Also, from the perspective of deep learning, the training process can become unstable if the neural networks are very deep and/or the model size is very large. In response, we typically need to modify the model architecture to adapt LLMs to large-scale training.

2.1.3 Fine-tuning LLMs

Once we have pre-trained an LLM, we can then apply it to perform various NLP tasks. Traditionally language models are used as components of other systems, for example, they are widely applied to score translations in statistical machine translation systems. By contrast, in generative AI, LLMs are considered complete systems and are employed to address NLP problems by making use of their generation nature. A common approach is to describe the task we want to address in text and then prompt LLMs to generate text based on this description. This is a standard text generation task where we continue or complete the text starting from a given context.

More formally, let $\mathbf{x} = x_0...x_m$ denote a token sequence of context given by users, and $\mathbf{y} = y_1...y_n$ denote a token sequence following the context. Then, the inference of LLMs can be defined as a problem of finding the most likely sequence $\mathbf{y}$ based on $\mathbf{x}$:

$$\begin{aligned} \hat{\mathbf{y}} &= \arg\max_{\mathbf{y}} \log \Pr(\mathbf{y}|\mathbf{x}) \\ &= \arg\max_{\mathbf{y}} \sum_{i=1}^{n} \log \Pr(y_i|x_0, ..., x_m, y_1, ..., y_{i-1}) \end{aligned} \tag{2.15}$$

Here $\sum_{i=1}^{n} \log \Pr(y_i|x_0, ..., x_m, y_1, ..., y_{i-1})$ essentially expresses the same thing as the right-hand side of Eq. (2.2). It models the log probability of predicting tokens from position $m + 1$, rather than position 0. Throughout this chapter and subsequent ones, we will employ separate variables $\mathbf{x}$ and $\mathbf{y}$ to distinguish the input and output of an LLM, though they can be seen as subsequences from the same sequence. By adopting such notation, we see that the form of the above equation closely resembles those used in other text generation models in NLP, such as neural machine translation models.

To illustrate how LLMs are applied, consider the problem of determining the grammaticality for a given sentence. We can define a template like this

> {*sentence*}
> Question: Is this sentence grammatically correct?
> Answer: ____

Here __ represents the text we intend to generate. {*sentence*} is a placeholder variable that will be replaced by the actual sentence provided by the users. For example, suppose we have a sentence "*John seems happy today.*". We can replace the {*sentence*} in the template with this sentence to have an input to the language model

> John seems happy today.
> Question: Is this sentence grammatically correct?
> Answer: ____

To perform the task, the language model is given the context $\mathbf{x}$ ="John seems happy today .\n Question : Is this sentence grammatically correct?\n Answer :"[4]. It then generates the following

[4] \n is a special character used for line breaks.

text as the answer, based on the context. For example, the language model may output "Yes" (i.e., $\mathbf{y} =$ "Yes") if this text is the one with the maximum probability of prediction given this context.

Likewise, we can define more templates to address other tasks. For example, we can translate an English sentence into Chinese using the following template

```
{*sentence*}
Question: What is the Chinese translation of this English sentence?
Answer: ____
```

or using an instruction-like template

```
{*sentence*}
Translate this sentence from English into Chinese.
____
```

or using a code-like template.

```
[src-lang] = English [tgt-lang] = Chinese [input] = {*sentence*}
[output] = ____
```

The above templates provide a simple but effective method to "prompt" a single LLM to perform various tasks without adapting the structure of the model. However, this approach requires that the LLM can recognize and follow the instructions or questions. One way to do this is to incorporate training samples with instructions and their corresponding responses into the pre-training dataset. While this method is straightforward, building and training LLMs from scratch is computationally expensive. Moreover, making instruction-following data effective for pre-training requires a significant amount of such data, but collecting large-scale labeled data for all tasks of interest is very difficult.

A second method, which has been a de facto standard in recent research, is to adapt LLMs via fine-tuning. As such, the token prediction ability learned in the pre-training phase can be generalized to accomplish new tasks. The idea behind fine-tuning is that some general knowledge of language has been acquired in pre-training, but we need a mechanism to activate this knowledge for applying it to new tasks. To achieve this, we can slightly fine-tune the model parameters using instruction-following data. This approach is called **instruction fine-tuning**.

An instruction fine-tuning sample, which is represented by a sequence of tokens, can be seen as a tuple consisting of an input and the desired output. Here, the input includes instructions, system information (or system prefix), and any other user-provided information[5]. To illustrate, consider the following examples (blue text = input and underlined text = output).

[5] System information refers to a sequence of tokens added at the beginning of an input in order to guide the behavior of an LLM, such as, *you are a helpful assistant and should not output toxic content*.

Read the following sentence, and answer whether it is grammatically correct.
LLMs are powerful models but are expensive to build.

Yes

LLMs are powerful models but are expensive to build.
Does this sentence make sense grammatically?
Answer Option
- Yes
- No

Yes

Yes or No? Yes = the following sentence is grammatically correct. No = it contains grammatical errors.
LLMs are powerful models but are expensive to build.

Yes

All these samples describe the same binary classification task, but with different instructions. To increase the diversity in the fine-tuning data and make LLMs generalize, we can define more tasks using instructions.

Write a short story about three characters.
Character 1: A boy whose name is Andy.
Character 2: Andy's dog.
Character 3: Andy's friend, Jane.

In the heart of a bustling city, Andy, his loyal dog Rocket, and his friend Jane discovered a hidden garden beneath the glow of twilight. Surrounded by ancient trees and whispering winds, they shared dreams and secrets, forging an unbreakable bond. As the night deepened, they made a pact under the stars to always find magic in the mundane. Together, they realized that true adventure lay in the moments spent with friends, and every day was a treasure waiting to be discovered.

Show me a recipe for making ice cream.

Ingredients:
2 cups heavy cream; 1 cup milk; 3/4 cup sugar; 2 tsp vanilla extract;
Pinch of salt.
Steps:
1. Whisk all ingredients until sugar dissolves.
2. Chill mixture for 2 hours in the fridge.
3. Freeze in a shallow dish, stirring every 30 minutes, 3-4 times.
4. Freeze solid for 2 hours.
5. Serve your homemade ice cream.

If you buy 5 apples and each apple costs $1.20, how much do you spend in total?

$6.00

Write a Python program to calculate the sum of squares of the following numbers.
1 , 2 , 10 , -9 , 78

```
numbers = [1,2,10,-9 ,78]
sum_of_squares = sum(x**2 for x in numbers)
print(sum_of_squares)
```

To acquire instruction-following abilities, a certain amount of fine-tuning data is required. This data may include diverse instructions and possible responses. It has been found that scaling the number of fine-tuning tasks is beneficial for improving the performance of LLMs [Chung et al., 2022]. Note that although more fine-tuning data is favorable, the amount of this data is generally orders of magnitude smaller than that of the pre-training data. For example, LLMs can be fine-tuned with tens or hundreds of thousands of samples, or even fewer if these samples are of high quality [Zhou et al., 2023; Chen et al., 2023], whereas pre-training such models may require billions or trillions of tokens, resulting in significantly larger computational demands and longer training times [Touvron et al., 2023a].

It is also worth noting that we should not expect the fine-tuning data to cover all the downstream tasks to which we intend to apply LLMs. A common understanding of how the pre-training + fine-tuning approach works is that LLMs have gained knowledge for understanding instructions and generating responses in the pre-training phase. However, these abilities are not fully activated until we introduce some form of supervision. The general instruction-following behavior emerges as we fine-tune the models with a relatively small amount of labeled data. As a result, we can achieve some level of **zero-shot learning**: the fine-tuned models can handle new tasks that they have not been explicitly trained or fine-tuned for [Sanh et al., 2022; Wei et al., 2022a]. This zero-shot learning ability distinguishes generative LLMs from earlier pre-trained models like BERT, which are primarily fine-tuned for specific tasks.

Once we have prepared a collection of instruction-described data, the fine-tuning process is relatively simple. This process can be viewed as a standard training process as pre-training, but on a much smaller training dataset. Let $\mathcal{D}_{\text{tune}}$ be the fine-tuning dataset and $\hat{\theta}$ be the model parameters

optimized via pre-training. We can modify Eq. (2.14) to obtain the objective of fine-tuning

$$\tilde{\theta} \quad = \quad \arg\max_{\hat{\theta}^+} \sum_{\text{sample} \in \mathcal{D}_{\text{tune}}} \mathcal{L}_{\hat{\theta}^+}(\text{sample}) \tag{2.16}$$

Here $\tilde{\theta}$ denotes the optimal parameters. The use of notation $\hat{\theta}^+$ means that the fine-tuning starts with the pre-trained parameters $\hat{\theta}$.

For each $\text{sample} \in \mathcal{D}_{\text{tune}}$, we divide it into an input segment $\mathbf{x}_{\text{sample}}$ and an output segment $\mathbf{y}_{\text{sample}}$, that is,

$$\text{sample} \quad = \quad [\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}] \tag{2.17}$$

We then define the loss function to be

$$\mathcal{L}_{\hat{\theta}^+}(\text{sample}) \quad = \quad -\log \Pr_{\hat{\theta}^+}(\mathbf{y}_{\text{sample}} | \mathbf{x}_{\text{sample}}) \tag{2.18}$$

In other words, we compute the loss over the sub-sequence $\mathbf{y}_{\text{sample}}$, rather than the entire sequence. In a practical implementation of back-propagation for this equation, the sequence $[\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}]$ is constructed in the forward pass as usual. However, in the backward pass, error gradients are propagated back only through the parts of the network that correspond to $\mathbf{y}_{\text{sample}}$, leaving the rest of the network unchanged. As an example, consider a sequence

$$\underbrace{\langle s \rangle \text{ Square this number . 2 .}}_{\text{Context (Input)}} \quad \underbrace{\text{The result is 4 .}}_{\text{Prediction (Output)}}$$

The loss is calculated and back propagated only for The result is 4 ..

Instruction fine-tuning also requires substantial engineering work. In order to achieve satisfactory results, one may experiment with different settings of the learning rate, batch size, number of fine-tuning steps, and so on. This typically requires many fine-tuning runs and evaluations. The cost and experimental effort of fine-tuning remain critical and should not be overlooked, though they are much lower than those of the pre-training phase.

While we focus on instruction fine-tuning for an illustrative example here, fine-tuning techniques play an important role in developing various LLMs and are more widely used. Examples include fine-tuning LLMs as chatbots using dialog data, and adapting these models to handle very long sequences. The wide application of fine-tuning has led researchers to improve these techniques, such as designing more efficient fine-tuning algorithms. While the research on fine-tuning is fruitful, in this section we just give a flavour of the key steps involved. We will see more detailed discussions on this topic in the following chapters.

2.1.4 Aligning LLMs with the World

Instruction fine-tuning provides a simple way to adapt LLMs to tasks that can be well defined. This problem can broadly be categorized as an **alignment** problem. Here, alignment is referred to as a process of guiding LLMs to behave in ways that align with human intentions. The guidance can come from labeled data, human feedback, or any other form of human preferences. For example,

we want LLMs not only to be accurate in following instructions, but also to be unbiased, truthful, and harmless. So we need to supervise the models towards human values and expectations. A common example is that when we ask an LLM how to build a weapon, it may provide a list of key steps to do so if it is not carefully aligned. However, a responsible model should recognize and avoid responding to requests for harmful or illegal information. Alignment in this case is crucial for ensuring that LLMs act responsibly and in accordance with ethical guidelines.

A related concept to alignment is AI safety. One ultimate goal of AI is to build intelligent systems that are safe and socially beneficial. To achieve this goal we should keep these systems robust, secure, and subjective, in any conditions of real-world use, even in conditions of misuse or adverse use. For LLMs, the safety can be increased by aligning them with appropriate human guidance, such as human labeled data and interactions with users during application.

Alignment is difficult as human values and expectations are diverse and shifting. Sometimes, it is hard to describe precisely what humans want, unless we see the response of LLMs to user requests. This makes alignment no longer a problem of tuning LLMs on predefined tasks, but a bigger problem of training them with the interactions with the real world.

As a result of the concerns with controlling AI systems, there has been a surge in research on the alignment issue for LLMs. Typically, two alignment steps are adopted after LLMs are pre-trained on large-scale unlabeled data.

- **Supervised Fine-tuning** (**SFT**). This involves continuing the training of pre-trained LLMs on new, task-oriented, labelled data. A commonly used SFT technique is instruction fine-tuning. As described in the previous subsection, by learning from instruction-response annotated data, LLMs can align with the intended behaviors for following instructions, thereby becoming capable of performing various instruction-described tasks. Supervised fine-tuning can be seen as following the pre-training + fine-tuning paradigm, and offers a relatively straightforward method to adapt LLMs.

- **Learning from Human Feedback**. After an LLM finishes pre-training and supervised fine-tuning, it can be used to respond to user requests if appropriately prompted. But this model may generate content that is unfactual, biased, or harmful. To make the LLM more aligned with the users, one simple approach is to directly learn from human feedback. For example, given some instructions and inputs provided by the users, experts are asked to evaluate how well the model responds in accordance with their preferences and interests. This feedback is then used to further train the LLM for better alignment.

A typical method for learning from human feedback is to consider it as a reinforcement learning (RL) problem, known as **reinforcement learning from human feedback** (**RLHF**) [Ouyang et al., 2022]. The RLHF method was initially proposed to address general sequential decision-making problems [Christiano et al., 2017], and was later successfully employed in the development of the GPT series models [Stiennon et al., 2020]. As a reinforcement learning approach, the goal of RLHF is to learn a policy by maximizing some reward from the environment. Specifically, two components are built in RLHF:

- **Agent**. An agent, also called an LM agent, is the LLM that we want to train. This agent operates by interacting with its environment: it receives a text from the environment and

outputs another text that is sent back to the environment. The policy of the agent is the function defined by the LLM, that is, $\Pr(\mathbf{y}|\mathbf{x})$.

- **Reward Model**. A reward model is a proxy of the environment. Each time the agent produces an output sequence, the reward model assigns this output sequence a numerical score (i.e., the reward). This score tells the agent how good the output sequence is.

In RLHF, we need to perform two learning tasks: 1) reward model learning, which involves training a reward model using human feedback on the output of the agent, and 2) policy learning, which involves optimizing a policy guided by the reward model using reinforcement learning algorithms. Here is a brief outline of the key steps involved in RLHF.

- Build an initial policy using pre-training and instruction fine-tuning.
- Use the policy to generate multiple outputs for each input, and then collect human feedback on these outputs (e.g., comparisons of the outputs).
- Learn a reward model from the human feedback.
- Fine-tune the policy with the supervision from the reward model.

2.1.5 Prompting LLMs

We have so far shown that LLMs can be used to perform various tasks by giving them appropriate prompts. There are no restrictions on these prompts, which can include any information we wish to ask or communicate with LLMs, such as natural language instructions and the context of conversations. Since this approach requires no additional training or tuning, adapting LLMs becomes highly efficient once they are developed. This somewhat influences the paradigms in NLP: we no longer need to develop specific systems for individual tasks but can instead use a single, well-trained LLM to perform different tasks by prompting it. An appealing aspect of LLM prompting arises as a result: users can easily have "customized" systems by designing their own prompts for LLMs. Given the important role played by prompting in LLMs, **prompting engineering** has become a very active area of research in NLP.

The term *prompt* is used in many different ways in the literature. In this chapter, this term refers to the entire input to LLMs, and so we use the terms *prompt* and *model input* interchangeably. Before discussing prompting further, let us first see a few examples where the prompts are more complex than those presented in the previous subsections. Note that this subsection is not aimed at writing high-quality prompts but rather at highlighting some interesting issues in prompting LLMs.

One of the popular ways to use LLMs is to assign them a "role" played in generating responses. For example, LLMs can act as a psychologist when answering questions.

Please explain what delayed gratification is?

Note, you are a researcher with a deep background in psychology. Your own field of study is developmental psychology in children, with a focus on aspects of control and intelligence. You are very familiar with the concept of delayed gratification and have applied it to your own research. You have published papers on delayed gratification and conducted related experiments.

Delayed gratification is the process of resisting an immediate reward in anticipation of receiving a more valuable reward in the future. It is a significant concept in developmental psychology, especially in relation to how children learn self-control and develop cognitive abilities. Research shows that the ability to delay gratification is associated with positive outcomes in academic performance, social competence, and emotional well-being. Studies and experiments in this area highlight the importance of cultivating this skill early in life to support long-term success and resilience.

Another example is the use of LLMs in detecting and correcting errors such as syntactic or semantic mistakes in text. For an LLM which is trained on both code and natural language data, we may use it for code debugging[6].

Fix the bugs in this C language program.

```
#include <stdio.h>
int main() {
    printg("Hello, World!")
    return 0;
}
```

1. The function name 'printg' is incorrect . It should be 'printf'.
2. There is a missing semicolon at the end of the 'printf' function call.

Here is the corrected version of the program:

```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

Also, LLMs can be fine-tuned to act as a conversation partner [7].

[6] In this example, the code is not tokenized for easier reading.

[7] To fine-tune an LLM for multi-turn dialogue, one needs to consider conversation history in the context for predicting the response in the current round of conversation. This makes the actual prompt used in response generation relatively longer than that used in single-turn dialogue.

Hi! I'm an assistant. How can I help you?

Who won the FIFA World Cup 2022?

Argentina won the FIFA World Cup 2022.

Where was it held?

The 2022 FIFA World Cup was held in Qatar.

How many times has Argentina won the World Cup?

Argentina has won the FIFA World Cup three times.

...

These examples and previous ones have shown that appropriate responses can be generated via prompts involving clear instructions and questions. However, when problem solving requires knowledge that is not explicitly specified, LLMs may make mistakes, even though the instructions are sufficiently clear and precise. A family of challenging tasks for LLMs involves arithmetic reasoning and commonsense reasoning. For example, we can ask an LLM to solve primary school math problems presented in natural language.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

The answer is 10.

The correct answer should be 7, so the model output is incorrect.

One approach to addressing such issues is to incorporate learning into prompts, called **in-context learning** or (**ICL**). The idea of ICL is to demonstrate the ways to solve problems in prompts, and condition predictions on these demonstrations. Here is an example where a similar problem and the corresponding answer are presented in the prompt (green = demonstrations).

Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?

The answer is 17.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?

The answer is 12.

But the LLM still made mistakes this time. A reason for this might be that solving math problems does not only involve problem-answer mappings but also, to a larger extent, the underlying logical inference in multiple steps. A method to improve the inference abilities of LLMs is **chain-of-thought prompting** (**COT prompting**) [Wei et al., 2022b]. In COT prompting, we decompose complex reasoning problems into multiple problem-solving intermediate steps. These

steps are demonstrated in prompts so that LLMs can be prompted to learn to mimic the underlying thought processes in solving the problems. COT prompting has been found to be very useful for achieving good results on challenging tasks, such as the GSM8K mathematical reasoning benchmark.

Consider the above demonstration example *Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now ?*. The corresponding logical reasoning steps can be structured:

1. Tom starts with 12 marbles.
2. He wins 7 more marbles in a game, so we add those to his initial count: 12 marbles + 7 marbles = 19 marbles.
3. The next day, Tom loses 5 marbles, so we subtract those from his current total: 19 marbles - 5 marbles = 14 marbles.
4. His brother gives him 3 more marbles, so we add those to his current total: 14 marbles + 3 marbles = 17 marbles.
5. After all these events, Tom has 17 marbles.

We can add these reasoning steps into the prompt, and get the correct answer.

> Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?
>
> Tom initially has 12 marbles. After winning a game, he gains 7 more, increasing his total to 19 marbles. He loses 5 marbles the next day, bringing his total down to 14 marbles. His brother gifts him 3 more marbles, increasing his total to 17 marbles. Therefore, Tom now has 17 marbles. So the answer is 17.
>
> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> Jack starts with 7 apples. He eats 2 apples for dinner, so we subtract 2 from 7, leaving him with 5 apples. His mom gives him 5 more apples, adding 5 to his current total. Now he has 5 + 5 = 10 apples. The next day, Jack gives 3 apples to his friend John. Subtracting 3 from 10, Jack is left with 7 apples. Therefore, Jack has 7 apples left. So the answer is 7.

This approach is also called **one-shot COT prompting**. Likewise, we can add more demonstrations to have a **few-shot COT prompting** approach.

Furthermore, a special kind of COT, **zero-shot COT**, introduces instructions that provoke LLMs to think, without presenting any intermediate reasoning steps in the prompts [Kojima et al.,

2022]. An interesting example is that by appending a simple sentence "Let's think step by step." to the end of a prompt, LLMs can generate reasoning steps to reach the final answer.

> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
> Let's think step by step.
>
> 1. Initial Quantity: Jack starts with 7 apples.
> 2. After Dinner: He eats 2 apples, so $7 - 2 = 5$ apples remain.
> 3. His Mom Gives More: His mom gives him 5 more apples, so $5 + 5 = 10$ apples are now with him.
> 4. Giving to John: The next day, Jack gives 3 apples to his friend John, so $10 - 3 = 7$ apples are left.
>
> In the end, Jack has 7 apples left.

Zero-shot, one-shot, and few-shot learning are common concepts in the area of in-context learning for LLMs and are not restricted to COT prompting. Broadly speaking, any prompting that involves only simple instructions without any demonstrations can be considered a form of zero-shot learning. This zero-shot learning ability emerges as LLMs are pre-trained and/or fine-tuned. Also, one-shot and few-shot learning methods are more often considered when LLMs do not acquire the corresponding zero-shot learning ability. These methods are therefore important for in-context learning when addressing new tasks. Examples include those for performing various NLP tasks by demonstrating task-formatted samples. See the following examples for sentiment sentence classification and phrase translation via few-shot learning.

> Given the following text snippets, classify their sentiment as Positive, Negative, or Neutral.
>
> Example 1: "I had an amazing day at the park!"
> Sentiment: Positive
>
> Example 2: "The service at the restaurant was terrible."
> Sentiment: Negative
>
> Example 3: "I think it's going to rain today."
> Sentiment: Neutral
>
> Text: "This movie was a fantastic journey through imagination."
> Sentiment: Positive

Translate the following Chinese phrases into English.

Example 1: “你好”
Translation: “Hello”

Example 2: “谢谢你”
Translation: “Thank you”

Phrase to translate: “早上好”
Translation: “Good Morning”

Above, we have presented examples to illustrate the fundamental in-context learning capabilities of prompting LLMs. This section, however, does not include more advanced prompting techniques in order to keep the content concise and compact. More discussions on prompting can be found in Chapter 3.

CHAPTER 3

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. $\rightarrow$ Label: Positive
>
> Example 2: I'm frustrated with the delays. $\rightarrow$ 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.

3.1 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.

3.1.1 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., 2023]. 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
{∗sentence1∗}
Here is another sentence
{∗sentence2∗}
Compute the semantic similarity between the two sentences
____

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

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

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

Q: {∗question∗}
A: ____

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

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

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

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.

3.1.2 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

---

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

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 × 5) = 0.283
        3  1  →  (3 + 1)/(3 × 1) = 1.33
       −9  4  →  (−9 + 4)/(−9 × 4) = 0.138
       15 15  →  (15 + 15)/(15 × 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 decent [Dai et al., 2023; Von Oswald et al., 2023], linear regression [Akyürek et al., 2023], meta learning [Garg et al., 2022], and so on.

3.1.3 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.
> $\{*\text{problem}*\}$
> ——

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.

- **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].

3.1.4 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.

3.1.4.1 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

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

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 positive.

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 "positive" from "The polarity of the text can be classified as positive". 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 positive

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

$$\mathrm{label} \quad = \quad \arg\max_{y \in Y} \Pr(y|\mathbf{x}) \tag{3.1}$$

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.
>
> Positive

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?
>
> Positive

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.

3.1.4.2 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.

> 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

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

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 $\{*\text{task-description}*\}$
>
> Text: $\{*\text{text}*\}$
>
> ——

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.

3.1.4.3 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
```

3.1.4.4 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 this 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!

3.1.4.5 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: ____

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 ≪ ... ≫ annotates the detailed calculation steps (called **calculation annotation**)[4].

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

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.

[Bengio et al., 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[Bengio et al., 2006] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

[Blum and Mitchell, 1998] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[Brown et al., 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[Chen et al., 2023] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpagasus: Training a better alpaca with fewer data. *arXiv preprint arXiv:2307.08701*, 2023.

[Christiano et al., 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[Chung et al., 2022] Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed Huai hsin Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[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.

[Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[Dubey et al., 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[Erhan et al., 2010] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208, 2010.

[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.

[He et al., 2019] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.

[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.

[Jelinek, 1998] Frederick Jelinek. *Statistical methods for speech recognition*. MIT Press, 1998.

[Jiang et al., 2023] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. *Speech and Language Processing (2nd ed.)*. Prentice Hall, 2008.

[Kaplan et al., 2020] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[Koehn, 2010] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

[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.

[Liu et al., 2024] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[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, 2023.

[Mikolov et al., 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, 2013a.

[Mikolov et al., 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 3111–3119, 2013b.

[Ouyang et al., 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[Penedo et al., 2023] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

[Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[Peters et al., 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[Radford et al., 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[Radford et al., 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 2019.

[Sanh et al., 2022] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *Proceedings of International Conference on Learning Representations*, 2022.

[Schmidhuber, 2015] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[Shannon, 1951] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[Stiennon et al., 2020] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[Sutskever et al., 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[Team et al., 2024] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[Touvron et al., 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

[Touvron et al., 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

[Vaswani et al., 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems*, volume 30, 2017.

[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.

[Wei et al., 2022] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *Proceedings of International Conference on Learning Representations*, 2022a.

[Wei et al., 2022] Jason Wei, 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, 2022b.

[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.

[Yang et al., 2024] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[Yarowsky, 1995] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.

[Zhou et al., 2023] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023.

[Zoph et al., 2020] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33:3833–3845, 2020.

Index