

Foundations of Large Language Models

Tong Xiao and Jingbo Zhu

June 17, 2025

NLP Lab, Northeastern University & NiuTrans Research

This book is a selection of chapters from an introductory NLP resource
available at <https://github.com/NiuTrans/NLPBook>

Copyright © 2021-2025 Tong Xiao and Jingbo Zhu

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

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

June 17, 2025

Preface

Large language models originated from natural language processing, but they have undoubtedly become one of the most revolutionary technological advancements in the field of artificial intelligence in recent years. An important insight brought by large language models is that knowledge of the world and languages can be acquired through large-scale language modeling tasks, and in this way, we can create a universal model that handles diverse problems. This discovery has profoundly impacted the research methodologies in natural language processing and many related disciplines. We have shifted from training specialized systems from scratch using a large amount of labeled data to a new paradigm of using large-scale pre-training to obtain foundation models, which are then fine-tuned, aligned, and prompted.

This book aims to outline the basic concepts of large language models and introduce the related techniques. As the title suggests, the book focuses more on the foundational aspects of large language models rather than providing comprehensive coverage of all cutting-edge methods. The book consists of five chapters:

- Chapter 1 introduces the basics of pre-training. This is the foundation of large language models, and common pre-training methods and model architectures will be discussed here.
- Chapter 2 introduces generative models, which are the large language models we commonly refer to today. After presenting the basic process of building these models, we will also explore how to scale up model training and handle long texts.
- Chapter 3 introduces prompting methods for large language models. We will discuss various prompting strategies, along with more advanced methods such as chain-of-thought reasoning and automatic prompt design.
- Chapter 4 introduces alignment methods for large language models. We will focus on instruction fine-tuning and alignment based on human feedback.
- Chapter 5 introduces inference methods for large language models. We will discuss various decoding algorithms, acceleration methods, and the inference-time scaling issue.

If readers have some background in machine learning and natural language processing, along with a certain understanding of neural networks like Transformers, reading this book will be quite easy. However, even without this prior knowledge, it is still perfectly fine, as we have made the content of each chapter as self-contained as possible, ensuring that readers will not be burdened with too much reading difficulty.

The content presented here is part of a comprehensive introductory resource on neural networks and large language models in natural language processing. For readers who wish to learn more about background topics, such as sequence modeling and attention mechanisms, you can visit <https://github.com/NiuTrans/NLPBook> or <https://niutrans.github.io/NLPBook> for further information.

We would like to thank the students in our laboratory and all our friends who have shared with us their views on large language models and helped with corrections of errors in writing. In particular, we wish to thank Weiqiao Shan, Yongyu Mu, Chenglong Wang, Kaiyan Chang, Yuchun Fan, Hang Zhou, Chuanhao Lv, Xinyu Liu, Tao Zhou, Huiwen Bao, Tong Zheng, Junhao Ruan, Yingfeng Luo, Yuzhang Wu, and Yifu Huo.

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
$\text{Softmax}(\mathbf{A})$	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
$\text{KL}(p \parallel q)$	KL divergence between distributions p and q

Contents

1	Pre-training	1
1.1	Pre-training NLP Models	1
1.1.1	Unsupervised, Supervised and Self-supervised Pre-training	2
1.1.2	Adapting Pre-trained Models	3
1.2	Self-supervised Pre-training Tasks	7
1.2.1	Decoder-only Pre-training	7
1.2.2	Encoder-only Pre-training	8
1.2.3	Encoder-Decoder Pre-training	15
1.2.4	Comparison of Pre-training Tasks	20
1.3	Example: BERT	21
1.3.1	The Standard Model	21
1.3.2	More Training and Larger Models	27
1.3.3	More Efficient Models	27
1.3.4	Multi-lingual Models	28
1.4	Applying BERT Models	30
1.5	Summary	35
2	Generative Models	36
2.1	A Brief Introduction to LLMs	37
2.1.1	Decoder-only Transformers	38
2.1.2	Training LLMs	40
2.1.3	Fine-tuning LLMs	42
2.1.4	Aligning LLMs with the World	46
2.1.5	Prompting LLMs	51
2.2	Training at Scale	56
2.2.1	Data Preparation	56
2.2.2	Model Modifications	57
2.2.3	Distributed Training	60
2.2.4	Scaling Laws	63
2.3	Long Sequence Modeling	66
2.3.1	Optimization from HPC Perspectives	67
2.3.2	Efficient Architectures	68
2.3.3	Cache and Memory	70
2.3.4	Sharing across Heads and Layers	79

2.3.5	Position Extrapolation and Interpolation	81
2.3.6	Remarks	92
2.4	Summary	94
3	Prompting	96
3.1	General Prompt Design	97
3.1.1	Basics	97
3.1.2	In-context Learning	99
3.1.3	Prompt Engineering Strategies	101
3.1.4	More Examples	106
3.2	Advanced Prompting Methods	115
3.2.1	Chain of Thought	115
3.2.2	Problem Decomposition	117
3.2.3	Self-refinement	124
3.2.4	Ensembling	130
3.2.5	RAG and Tool Use	134
3.3	Learning to Prompt	138
3.3.1	Prompt Optimization	139
3.3.2	Soft Prompts	142
3.3.3	Prompt Length Reduction	152
3.4	Summary	153
4	Alignment	155
4.1	An Overview of LLM Alignment	155
4.2	Instruction Alignment	157
4.2.1	Supervised Fine-tuning	157
4.2.2	Fine-tuning Data Acquisition	161
4.2.3	Fine-tuning with Less Data	166
4.2.4	Instruction Generalization	167
4.2.5	Using Weak Models to Improve Strong Models	169
4.3	Human Preference Alignment: RLHF	172
4.3.1	Basics of Reinforcement Learning	173
4.3.2	Training Reward Models	179
4.3.3	Training LLMs	182
4.4	Improved Human Preference Alignment	187
4.4.1	Better Reward Modeling	187

4.4.2	Direct Preference Optimization	193
4.4.3	Automatic Preference Data Generation	196
4.4.4	Step-by-step Alignment	198
4.4.5	Inference-time Alignment	200
4.5	Summary	201
5	Inference	203
5.1	Prefilling and Decoding	204
5.1.1	Preliminaries	204
5.1.2	A Two-phase Framework	207
5.1.3	Decoding Algorithms	211
5.1.4	Evaluation Metrics for LLM Inference	221
5.2	Efficient Inference Techniques	222
5.2.1	More Caching	223
5.2.2	Batching	223
5.2.3	Parallelization	232
5.2.4	Remarks	233
5.3	Inference-time Scaling	234
5.3.1	Context Scaling	235
5.3.2	Search Scaling	236
5.3.3	Output Ensembling	237
5.3.4	Generating and Verifying Thinking Paths	238
5.4	Summary	245
	Bibliography	247

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 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:

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

where $\{x_0, x_1, \dots, x_m\}$ denotes a sequence of input tokens¹, 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

¹Here 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.

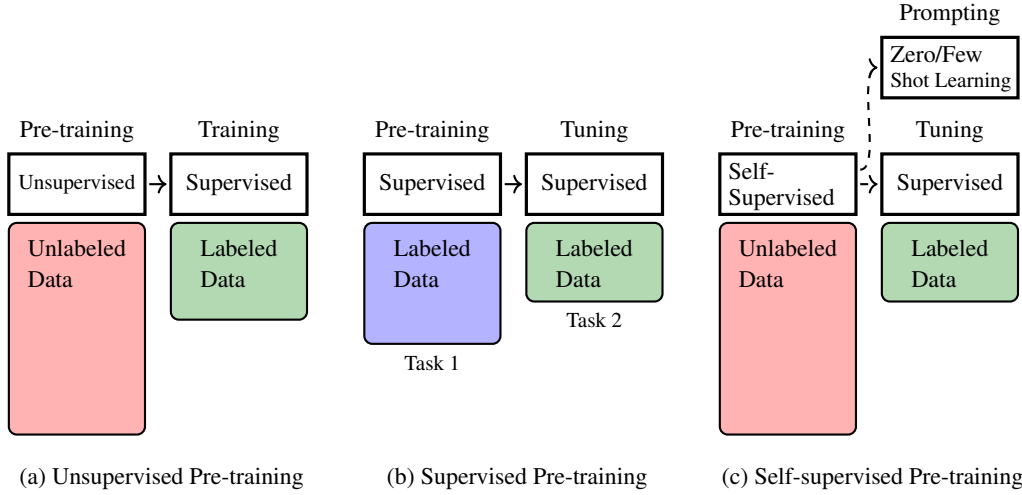


Fig. 1.1: Illustration of unsupervised, supervised, and self-supervised pre-training. In unsupervised pre-training, the pre-training is performed on large-scale unlabeled data. It can be viewed as a preliminary step to have a good starting point for the subsequent optimization process, though considerable 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 translation².

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

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

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

²More 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 $\text{Classify}_\omega(\cdot)$ be a neural network with parameters ω . Then, the text classification model can be expressed in the form

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

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

Because the model parameters 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 tokens³, and then feed the token sequence \mathbf{x}_{new} into the fine-tuned model $F_{\tilde{\omega},\tilde{\theta}}(\cdot)$. The model generates a distribution over classes by

$$F_{\tilde{\omega},\tilde{\theta}}(\mathbf{x}_{\text{new}}) = \left[\Pr(\text{positive}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{negative}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{neutral}|\mathbf{x}_{\text{new}}) \right] \quad (1.4)$$

And we select the label of the entry with the maximum value as output. In this example it is positive.

In general, the amount of labeled data used in fine-tuning is small compared to that of the pre-training data, and so fine-tuning is less computationally expensive. This makes the adaptation of pre-trained models very efficient in practice: given a pre-trained model and a downstream task, we just need to collect some labeled data, and slightly adjust the model parameters on this data. A more detailed discussion of fine-tuning can be found in Section 1.4.

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

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

as complete systems on downstream tasks. For example, we can fine-tune a pre-trained encoder-decoder multilingual model on some bilingual data to improve its performance on a specific translation task.

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

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

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

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

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 $\{x_0, \dots, x_i\}$, denoted by $\Pr_\theta(\cdot | x_0, \dots, x_i)$ (or \mathbf{p}_{i+1}^θ for short). Suppose we have the gold-standard distribution at the same position, denoted by $\mathbf{p}_{i+1}^{\text{gold}}$. For language modeling, we can think of $\mathbf{p}_{i+1}^{\text{gold}}$ as a one-hot representation of the correct predicted word. We then define a loss function $\mathcal{L}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}})$ to measure the difference between the model prediction and the true prediction. In NLP, the log-scale cross-entropy loss is typically used.

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

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

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

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}

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

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

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

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

1.2.2 Encoder-only Pre-training

As defined in Section 1.1.2.1, an encoder $\text{Encoder}_\theta(\cdot)$ is a function that reads a sequence of tokens $\mathbf{x} = x_0 \dots x_m$ and produces a sequence of vectors $\mathbf{H} = \mathbf{h}_0 \dots \mathbf{h}_m$ ⁴. Training this model is not straightforward, as we do not have gold-standard data for measuring how good the output of the real-valued function is. A typical approach to encoder pre-training is to combine the encoder with some output layers to receive supervision signals that are easier to obtain. Figure 1.2 shows a common architecture for pre-training Transformer encoders, where we add a Softmax layer on top of the Transformer encoder. Clearly, this architecture is the same as that of the decoder-based language model, and the output is a sequence of probability distributions

$$\begin{bmatrix} \mathbf{p}_1^{\mathbf{W}, \theta} \\ \vdots \\ \mathbf{p}_m^{\mathbf{W}, \theta} \end{bmatrix} = \text{Softmax}_{\mathbf{W}}(\text{Encoder}_\theta(\mathbf{x})) \quad (1.9)$$

⁴If we view \mathbf{h}_i as a row vector, \mathbf{H} can be written as

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \quad (1.8)$$

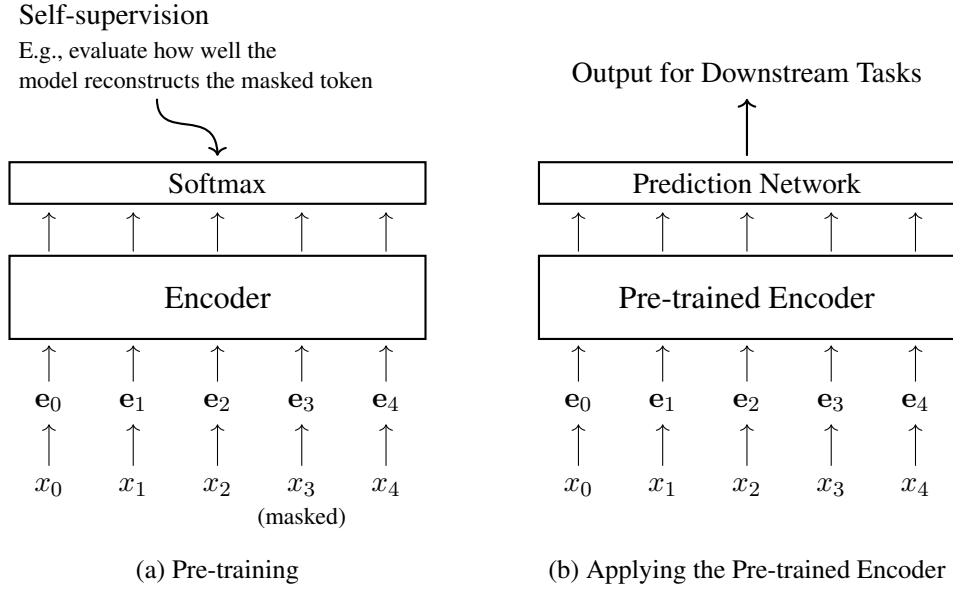


Fig. 1.2: Pre-training a Transformer encoder (left) and then applying the pre-trained encoder (right). In the pre-training phase, the encoder, together with a Softmax layer, is trained via self-supervision. In the application phase, the Softmax layer is removed, and the pre-trained encoder is combined with a prediction network to address specific problems. In general, for better adaptation to these tasks, the system is fine-tuned using labeled data.

Here $\mathbf{p}_i^{\mathbf{W}, \theta}$ is the output distribution $\Pr(\cdot | \mathbf{x})$ at position i . We use $\text{Softmax}_{\mathbf{W}}(\cdot)$ to denote that the Softmax layer is parameterized by \mathbf{W} , that is, $\text{Softmax}_{\mathbf{W}}(\mathbf{H}) = \text{Softmax}(\mathbf{H} \cdot \mathbf{W})$. For notation simplicity, we will sometimes drop the superscripts \mathbf{W} and θ affixed to each probability distribution.

The difference between this model and standard language models is that the output \mathbf{p}_i has different meanings in encoder pre-training and language modeling. In language modeling, \mathbf{p}_i is the probability distribution of predicting the next word. This follows an auto-regressive decoding process: a language model only observes the words up to position i and predicts the next. By contrast, in encoder pre-training, the entire sequence can be observed at once, and so it makes no sense to predict any of the tokens in this sequence.

1.2.2.1 Masked Language Modeling

One of the most popular methods of encoder pre-training is **masked language modeling**, which forms the basis of the well-known BERT model [Devlin et al., 2019]. The idea of masked language modeling is to create prediction challenges by masking out some of the tokens in the input sequence and training a model to predict the masked tokens. In this sense, the conventional language modeling problem, which is sometimes called **causal language modeling**, is a special case of masked language modeling: at each position, we mask the tokens in the right-context, and predict the token at this position using its left-context. However, in causal language modeling we only make use of the left-context in word prediction, while the prediction may depend on tokens in the right-context. By contrast, in masked language modeling, all the unmasked tokens are used for word prediction, leading to a bidirectional model that makes predictions based on both left and right-contexts.

More formally, for an input sequence $\mathbf{x} = x_0 \dots x_m$, suppose that we mask the tokens at positions $\mathcal{A}(\mathbf{x}) = \{i_1, \dots, i_u\}$. Hence we obtain a masked token sequence $\bar{\mathbf{x}}$ where the token at each position in $\mathcal{A}(\mathbf{x})$ is replaced with a special symbol [MASK]. For example, for the following sequence

The early bird catches the worm

we may have a masked token sequence like this

The [MASK] bird catches the [MASK]

where we mask the tokens *early* and *worm* (i.e., $i_1 = 2$ and $i_2 = 6$).

Now we have two sequences \mathbf{x} and $\bar{\mathbf{x}}$. The model is then optimized so that we can correctly predict \mathbf{x} based on $\bar{\mathbf{x}}$. This can be thought of as an autoencoding-like process, and the training objective is to maximize the reconstruction probability $\Pr(\mathbf{x}|\bar{\mathbf{x}})$. Note that there is a simple position-wise alignment between \mathbf{x} and $\bar{\mathbf{x}}$. Because an unmasked token in $\bar{\mathbf{x}}$ is the same as the token in \mathbf{x} at the same position, there is no need to consider the prediction for this unmasked token. This leads to a simplified training objective which only maximizes the probabilities for masked tokens. We can express this objective in a maximum likelihood estimation fashion

$$(\widehat{\mathbf{W}}, \hat{\theta}) = \arg \max_{\mathbf{W}, \theta} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i^{\mathbf{W}, \theta}(x_i | \bar{\mathbf{x}}) \quad (1.10)$$

or alternatively express it using the cross-entropy loss

$$(\widehat{\mathbf{W}}, \hat{\theta}) = \arg \min_{\mathbf{W}, \theta} \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i \in \mathcal{A}(\mathbf{x})} \text{LogCrossEntropy}(\mathbf{p}_i^{\mathbf{W}, \theta}, \mathbf{p}_i^{\text{gold}}) \quad (1.11)$$

where $\Pr_k^{\mathbf{W}, \theta}(x_k | \bar{\mathbf{x}})$ is the probability of the true token x_k at position k given the corrupted input $\bar{\mathbf{x}}$, and $\mathbf{p}_k^{\mathbf{W}, \theta}$ is the probability distribution at position k given the corrupted input $\bar{\mathbf{x}}$. To illustrate, consider the above example where two tokens of the sequence “the early bird catches the worm” are masked. For this example, the objective is to maximize the sum of log-scale probabilities

$$\begin{aligned} \text{Loss} = & \log \Pr(x_2 = \text{early} | \bar{\mathbf{x}} = [\text{CLS}] \underbrace{\text{The [MASK] bird catches the [MASK]}}_{\bar{x}_2}) + \\ & \log \Pr(x_6 = \text{worm} | \bar{\mathbf{x}} = [\text{CLS}] \underbrace{\text{The [MASK] bird catches the [MASK]}}_{\bar{x}_6}) \end{aligned} \quad (1.12)$$

Once we obtain the optimized parameters $\widehat{\mathbf{W}}$ and $\hat{\theta}$, we can drop $\widehat{\mathbf{W}}$. Then, we can further fine-tune the pre-trained encoder $\text{Encoder}_{\hat{\theta}}(\cdot)$ or directly apply it to downstream tasks.

1.2.2.2 Permuted Language Modeling

While masked language modeling is simple and widely applied, it introduces new issues. One drawback is the use of a special token, [MASK], which is employed only during training but not

at test time. This leads to a discrepancy between training and inference. Moreover, the auto-encoding process overlooks the dependencies between masked tokens. For example, in the above example, the prediction of x_2 (i.e., the first masked token) is made independently of x_6 (i.e., the second masked token), though x_6 should be considered in the context of x_2 .

These issues can be addressed using the **permuted language modeling** approach to pre-training [Yang et al., 2019]. Similar to causal language modeling, permuted language modeling involves making sequential predictions of tokens. However, unlike causal modeling where predictions follow the natural sequence of the text (like left-to-right or right-to-left), permuted language modeling allows for predictions in any order. The approach is straightforward: we determine an order for token predictions and then train the model in a standard language modeling manner, as described in Section 1.2.1. Note that in this approach, the actual order of tokens in the text remains unchanged, and only the order in which we predict these tokens differs from standard language modeling. For example, consider a sequence of 5 tokens $x_0x_1x_2x_3x_4$. Let \mathbf{e}_i represent the embedding of x_i (i.e., combination of the token embedding and positional embedding). In standard language modeling, we would generate this sequence in the order of $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$. The probability of the sequence can be modeled via a generation process.

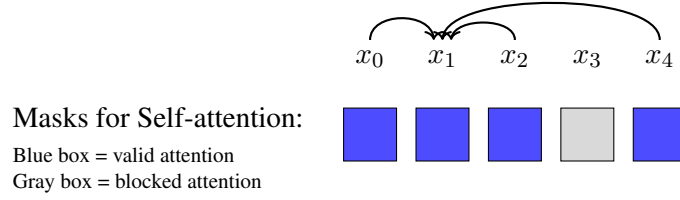
$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdot \Pr(x_3|x_0, x_1, x_2) \cdot \\ &\quad \Pr(x_4|x_0, x_1, x_2, x_3) \\ &= \Pr(x_0) \cdot \Pr(x_1|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_1) \cdot \Pr(x_3|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_4|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3) \end{aligned} \quad (1.13)$$

Now, let us consider a different order for token prediction: $x_0 \rightarrow x_4 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3$. The sequence generation process can then be expressed as follows:

$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_4|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_4) \cdot \Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_3|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2, \mathbf{e}_1) \end{aligned} \quad (1.14)$$

This new prediction order allows for the generation of some tokens to be conditioned on a broader context, rather than being limited to just the preceding tokens as in standard language models. For example, in generating x_3 , the model considers both its left-context (i.e., $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2$) and right-context (i.e., \mathbf{e}_4). The embeddings $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_4$ incorporate the positional information of x_0, x_1, x_2, x_4 , preserving the original order of the tokens. As a result, this approach is somewhat akin to masked language modeling: we mask out x_3 and use its surrounding tokens x_0, x_1, x_2, x_4 to predict this token.

The implementation of permuted language models is relatively easy for Transformers. Because the self-attention model is insensitive to the order of inputs, we do not need to explicitly reorder the sequence to have a factorization like Eq. (1.14). Instead, permutation can be done by setting appropriate masks for self-attention. For example, consider the case of computing $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2)$. We can place x_0, x_1, x_2, x_3, x_4 in order and block the attention from x_3 to x_1 in self-attention, as illustrated below



For a more illustrative example, we compare the self-attention masking results of causal language modeling, masked language modeling and permuted language modeling in Figure 1.3.

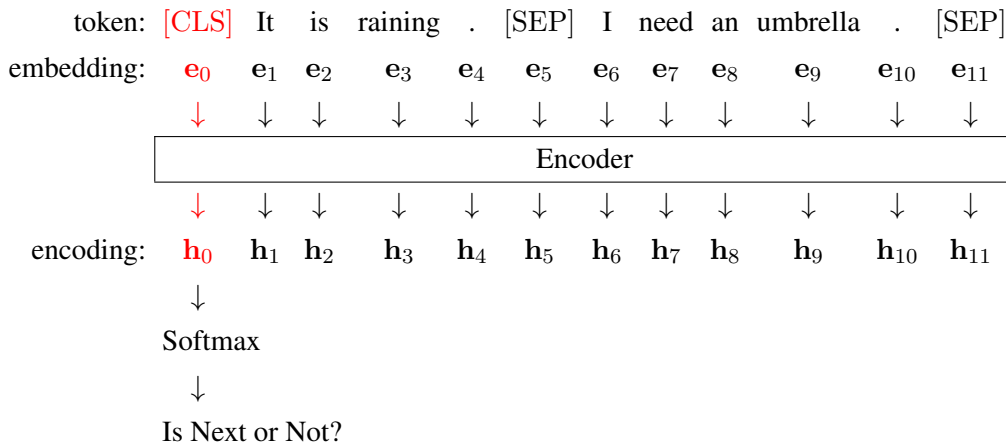
1.2.2.3 Pre-training Encoders as Classifiers

Another commonly-used idea to train an encoder is to consider classification tasks. In self-supervised learning, this is typically done by creating new classification challenges from the unlabeled text. There are many different ways to design the classification tasks. Here we present two popular tasks.

A simple method, called **next sentence prediction (NSP)**, is presented in BERT's original paper [Devlin et al., 2019]. The assumption of NSP is that a good text encoder should capture the relationship between two sentences. To model such a relationship, in NSP we can use the output of encoding two consecutive sentences Sent_A and Sent_B to determine whether Sent_B is the next sentence following Sent_A . For example, suppose $\text{Sent}_A = \text{'It is raining.'}$ and $\text{Sent}_B = \text{'I need an umbrella.'}$. The input sequence of the encoder could be

[CLS] It is raining . [SEP] I need an umbrella . [SEP]

where [CLS] is the start symbol (i.e., x_0) which is commonly used in encoder pre-training, and [SEP] is a separator that separates the two sentences. The processing of this sequence follows a standard procedure of Transformer encoding: we first represent each token x_i as its corresponding embedding e_i , and then feed the embedding sequence $\{e_0, \dots, e_m\}$ into the encoder to obtain the output sequence $\{h_0, \dots, h_m\}$. Since h_0 is generally considered as the representation of the entire sequence, we add a Softmax layer on top of it to construct a binary classification system. This process is illustrated as follows



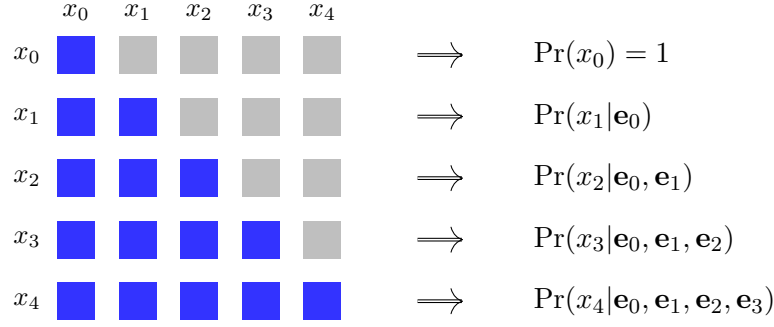
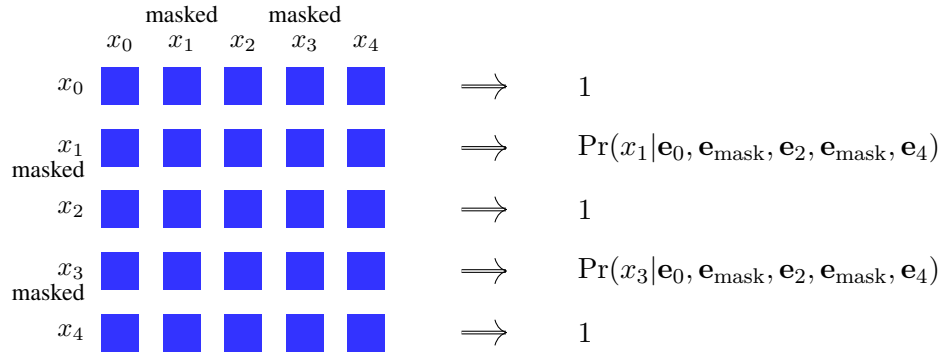
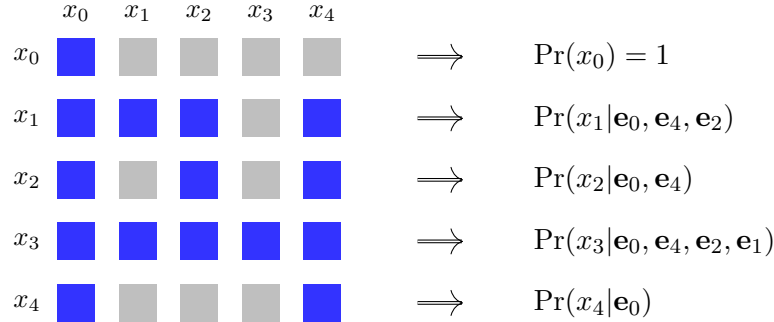
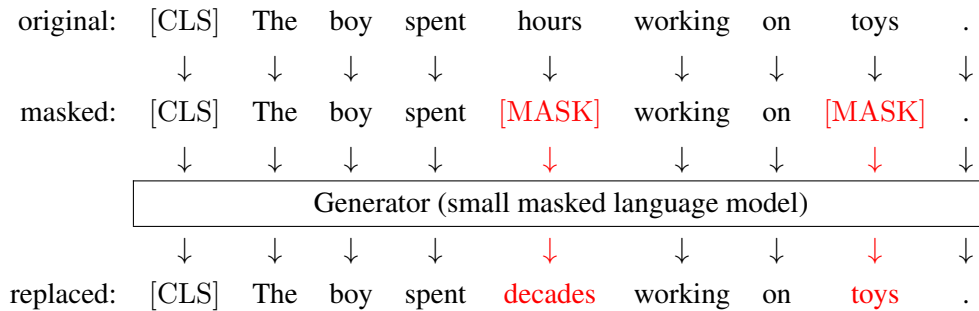
(a) Causal Language Modeling (order: $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$)(b) Masked Language Modeling (order: $x_0, [\text{MASK}], x_2, [\text{MASK}], x_4 \rightarrow x_1, x_3$)(c) Permuted Language Modeling (order: $x_0 \rightarrow x_4 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3$)

Fig. 1.3: Comparison of self-attention masking results of causal language modeling, masked language modeling and permuted language modeling. The gray cell denotes the token at position j does not attend to the token at position i . The blue cell (i, j) denotes that the token at position j attends to the token at position i . \mathbf{e}_{mask} represents the embedding of the symbol [MASK], which is a combination of the token embedding and the positional embedding.

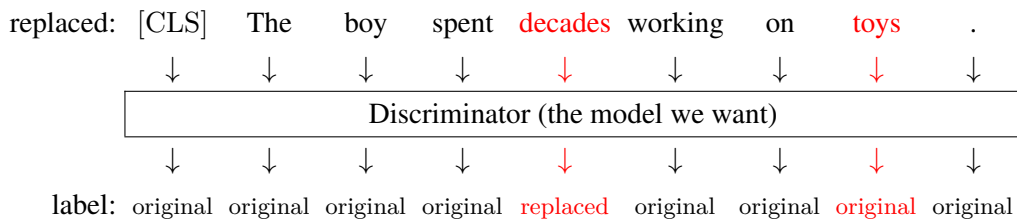
In order to generate training samples, we need two sentences each time, one for Sent_A and the other for Sent_B . A simple way to do this is to utilize the natural sequence of two consecutive sentences in the text. For example, we obtain a positive sample by using actual consecutive sentences, and a negative sample by using randomly sampled sentences. Consequently, training this model is the same as training a classifier. Typically, NSP is used as an additional training loss

function for pre-training based on masked language modeling.

A second example of training Transformer encoders as classifiers is to apply classification-based supervision signals to each output of an encoder. For example, [Clark et al. \[2019\]](#) in their ELECTRA model, propose training a Transformer encoder to identify whether each input token is identical to the original input or has been altered in some manner. The first step of this method is to generate a new sequence from a given sequence of tokens, where some of the tokens are altered. To do this, a small masked language model (call it the generator) is applied: we randomly mask some of the tokens, and train this model to predict the masked tokens. For each training sample, this masked language model outputs a token at each masked position, which might be different from the original token. At the same time, we train another Transformer encoder (call it the discriminator) to determine whether each predicted token is the same as the original token or altered. More specifically, we use the generator to generate a sequence where some of the tokens are replaced. Below is an illustration.



Then, we use the discriminator to label each of these tokens as original or replaced, as follows



For training, the generator is optimized as a masked language model with maximum likelihood estimation, and the discriminator is optimized as a classifier using a classification-based loss. In ELECTRA, the maximum likelihood-based loss and the classification-based loss are combined for jointly training both the generator and discriminator. An alternative approach is to use generative adversarial networks (GANs), that is, the generator is trained to fool the discriminator, and the discriminator is trained to distinguish the output of the generator from the true distribution. However, GAN-style training complicates the training task and is more difficult to scale up. Nevertheless, once training is complete, the generator is discarded, and the encoding part of the discriminator is applied as the pre-trained model for downstream tasks.

1.2.3 Encoder-Decoder Pre-training

In NLP, encoder-decoder architectures are often used to model sequence-to-sequence problems, such as machine translation and question answering. In addition to these typical sequence-to-sequence problems in NLP, encoder-decoder models can be extended to deal with many other problems. A simple idea is to consider text as both the input and output of a problem, and so we can directly apply encoder-decoder models. For example, given a text, we can ask a model to output a text describing the sentiment of the input text, such as *positive*, *negative*, and *neutral*.

Such an idea allows us to develop a single text-to-text system to address any NLP problem. We can formulate different problems into the same text-to-text format. We first train an encoder-decoder model to gain general-purpose knowledge of language via self-supervision. This model is then fine-tuned for specific downstream tasks using targeted text-to-text data.

1.2.3.1 Masked Encoder-Decoder Pre-training

In Raffel et al. [2020]’s **T5** model, many different tasks are framed as the same text-to-text task. Each sample in T5 follows the format

$$\text{Source Text} \rightarrow \text{Target Text}$$

Here \rightarrow separates the source text, which consists of a task description or instruction and the input given to the system, from the target text, which is the response to the input task. As an example, consider a task of translating from Chinese to English. A training sample can be expressed as

$$[\text{CLS}] \text{ Translate from Chinese to English: } \text{你好!} \rightarrow \langle s \rangle \text{ Hello!}$$

where $[\text{CLS}]$ and $\langle s \rangle$ are the start symbols on the source and target sides, respectively⁵.

Likewise, we can express other tasks in the same way. For example

$$\begin{aligned} &[\text{CLS}] \text{ Answer: } \text{when was Albert Einstein born?} \\ \rightarrow &\langle s \rangle \text{ He was born on March 14, 1879.} \end{aligned}$$

$$\begin{aligned} &[\text{CLS}] \text{ Simplify: } \text{the professor, who has published numerous papers in his field,} \\ &\text{will be giving a lecture on the topic next week.} \\ \rightarrow &\langle s \rangle \text{ The experienced professor will give a lecture next week.} \end{aligned}$$

$$\begin{aligned} &[\text{CLS}] \text{ Score the translation from English to Chinese. English: when in Rome, do as} \\ &\text{the Romans do. Chinese: 人在罗马就像罗马人一样做事。} \\ \rightarrow &\langle s \rangle 0.81 \end{aligned}$$

where instructions are highlighted in gray. An interesting case is that in the last example we

⁵We could use the same start symbol for different sequences. Here we use different symbols to distinguish the sequences on the encoder and decoder-sides.

reframe the scoring problem as the text generation problem. Our goal is to generate a text representing the number 0.81, rather than outputting it as a numerical value.

The approach described above provides a new framework of universal language understanding and generation. Both the task instructions and the problem inputs are provided to the system in text form. The system then follows the instructions to complete the task. This method puts different problems together, with the benefit of training a single model that can perform many tasks simultaneously.

In general, fine-tuning is necessary for adapting the pre-trained model to a specific downstream task. In this process, one can use different ways to instruct the model for the task, such as using a short name of the task as the prefix to the actual input sequence or providing a detailed description of the task. Since the task instructions are expressed in text form and involved as part of the input, the general knowledge of instruction can be gained through learning the language understanding models in the pre-training phase. This may help enable zero-shot learning. For example, pre-trained models can generalize to address new problems where the task instructions have never been encountered.

There have been several powerful methods of self-supervised learning for either Transformer encoders or decoders. Applying these methods to pre-train encoder-decoder models is relatively straightforward. One common choice is to train encoder-decoder models as language models. For example, the encoder receives a sequence prefix, while the decoder generates the remaining sequence. However, this differs from standard causal language modeling, where the entire sequence is autoregressively generated from the first token. In our case, the encoder processes the prefix at once, and then the decoder predicts subsequent tokens in the manner of causal language modeling. Put more precisely, this is a **prefix language modeling** problem: a language model predicts the subsequent sequence given a prefix, which serves as the context for prediction.

Consider the following example

$$\underbrace{[\text{CLS}] \text{ The puppies are frolicking}}_{\text{Prefix}} \rightarrow \underbrace{\langle s \rangle \text{ outside the house}}_{\text{Subsequent Sequence}}.$$

We can directly train an encoder-decoder model using examples like this. Then, the encoder learns to understand the prefix, and the decoder learns to continue writing based on this understanding. For large-scale pre-training, it is easy to create a large number of training examples from unlabeled text.

It is worth noting that for pre-trained encoder-decoder models to be effective in multi-lingual and cross-lingual tasks, such as machine translation, they should be trained with multi-lingual data. This typically requires that the vocabulary includes tokens from all the languages. By doing so, the models can learn shared representations across different languages, thereby enabling capabilities in both language understanding and generation in a multi-lingual and cross-lingual context.

A second approach to pre-training encoder-decoder models is masked language modeling. In this approach, as discussed in Section 1.2.2, tokens in a sequence are randomly replaced with a mask symbol, and the model is then trained to predict these masked tokens based on the entire masked sequence.

As an illustration, consider the task of masking and reconstructing the sentence

The puppies are frolicking outside the house .

By masking two tokens (say, *frolicking* and *the*), we have the BERT-style input and output of the model, as follows

[CLS] The puppies are [MASK] outside [MASK] house .
 → ⟨s⟩ _ _ _ frolicking _ the _ _

Here _ denotes the masked position at which we do not make token predictions. By varying the percentage of the tokens in the text, this approach can be generalized towards either BERT-style training or language modeling-style training [Song et al., 2019]. For example, if we mask out all the tokens, then the model is trained to generate the entire sequence

[CLS] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK]
 → ⟨s⟩ The puppies are frolicking outside the house .

In this case, we train the decoder as a language model.

Note that, in the context of the encoder-decoder architecture, we can use the encoder to read the masked sequence, and use the decoder to predict the original sequence. With this objective, we essentially have a denoising autoencoder: the encoder transforms a corrupted input into some hidden representation, and the decoder reconstructs the uncorrupted input from this hidden representation. Here is an example of input and output for denoising training.

[CLS] The puppies are [MASK] outside [MASK] house .
 → ⟨s⟩ The puppies are frolicking outside the house .

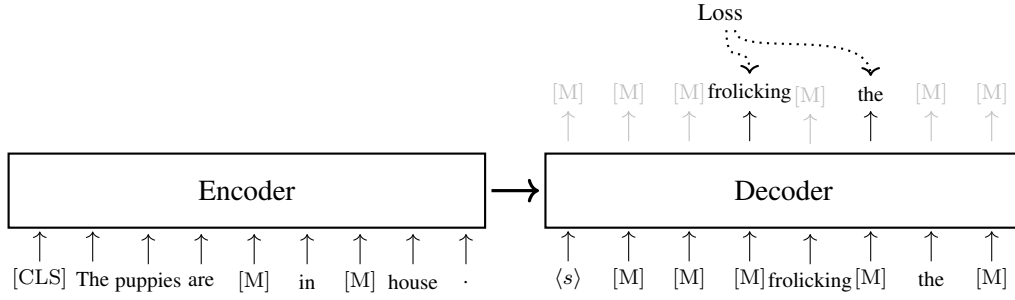
By learning to map from this corrupted sequence to its uncorrupted counterpart, the model gains the ability to understand on the encoder side and to generate on the decoder side. See Figure 1.4 for an illustration of how an encoder-decoder model is trained with BERT-style and denoising autoencoding objectives.

As we randomly select tokens for masking, we can certainly mask consecutive tokens [Joshi et al., 2020]. Here is an example.

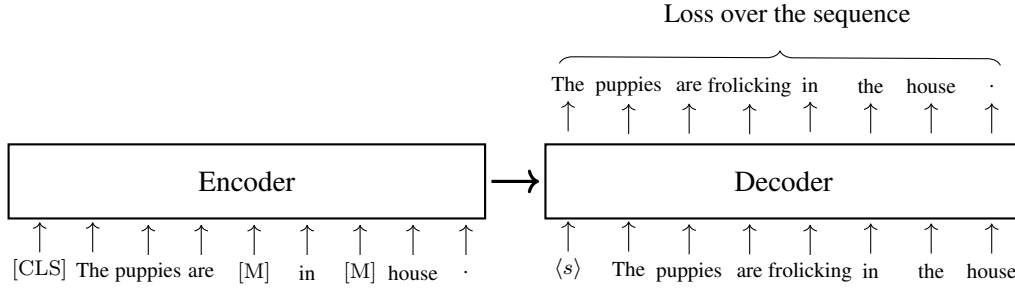
[CLS] The puppies are [MASK] outside [MASK] [MASK] .
 → ⟨s⟩ The puppies are frolicking outside the house .

Another way to consider consecutive masked tokens is to represent them as spans. Here we follow Raffel et al. [2020]’s work, and use [X], [Y] and [Z] to denote sentinel tokens that cover one or more consecutive masked tokens. Using this notation, we can re-express the above training example as

[CLS] The puppies are [X] outside [Y] .
 → ⟨s⟩ [X] frolicking [Y] the house [Z]



(a) Training an encoder-decoder model with BERT-style masked language modeling



(b) Training an encoder-decoder model with denoising autoencoding

Fig. 1.4: Training an encoder-decoder model using BERT-style and denoising autoencoding methods. In both methods, the input to the encoder is a corrupted token sequence where some tokens are masked and replaced with [MASK] (or [M] for short). The decoder predicts these masked tokens, but in different ways. In BERT-style training, the decoder only needs to compute the loss for the masked tokens, while the remaining tokens in the sequence can be simply treated as [MASK] tokens. In denoising autoencoding, the decoder predicts the sequence of all tokens in an autoregressive manner. As a result, the loss is obtained by accumulating the losses of all these tokens, as in standard language modeling.

The idea is that we represent the corrupted sequence as a sequence containing placeholder slots. The training task is to fill these slots with the correct tokens using the surrounding context. An advantage of this approach is that the sequences used in training would be shorter, making the training more efficient. Note that masked language modeling provides a very general framework for training encoder-decoder models. Various settings can be adjusted to have different training versions, such as altering the percentage of tokens masked and the maximum length of the masked spans.

1.2.3.2 Denoising Training

If we view the problem of training encoder-decoder models as a problem of training denoising autoencoders, there will typically be many different methods for introducing input corruption and reconstructing the input. For instance, beyond randomly masking tokens, we can also alter some of them or rearrange their order.

Suppose we have an encoder-decoder model that can map an input sequence \mathbf{x} to an output

sequence \mathbf{y}

$$\begin{aligned}\mathbf{y} &= \text{Decode}_{\omega}(\text{Encode}_{\theta}(\mathbf{x})) \\ &= \text{Model}_{\theta,\omega}(\mathbf{x})\end{aligned}\tag{1.15}$$

where θ and ω are the parameters of the encoder and the decoder, respectively. In denoising autoencoding problems, we add some noise to \mathbf{x} to obtain a noisy, corrupted input $\mathbf{x}_{\text{noise}}$. By feeding $\mathbf{x}_{\text{noise}}$ into the encoder, we wish the decoder to output the original input. The training objective can be defined as

$$(\hat{\theta}, \hat{\omega}) = \arg \min_{\theta, \omega} \text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x})\tag{1.16}$$

Here the loss function $\text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x})$ evaluates how well the model $\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}})$ reconstructs the original input \mathbf{x} . We can choose the cross-entropy loss as usual.

As the model architecture and the training approach have been developed, the remaining issue is the corruption of the input. [Lewis et al. \[2020\]](#), in their **BART** model, propose corrupting the input sequence in several different ways.

- **Token Masking.** This is the same masking method that we used in masked language modeling. The tokens in the input sequence are randomly selected and masked.
- **Token Deletion.** This method is similar to token masking. However, rather than replacing the selected tokens with a special symbol [MASK], these tokens are removed from the sequence. See the following example for a comparison of the token masking and token deletion methods.

Original (\mathbf{x}): The puppies are frolicking outside the house .
 Token Masking ($\mathbf{x}_{\text{noise}}$): The puppies are [MASK] outside [MASK] house .
 Token Deletion ($\mathbf{x}_{\text{noise}}$): The puppies are ~~frolicking~~ outside ~~the~~ house .

where the underlined tokens in the original sequence are masked or deleted.

- **Span Masking.** Non-overlapping spans are randomly sampled over the sequence. Each span is masked by [MASK]. We also consider spans of length 0, and, in such cases, [MASK] is simply inserted at a position in the sequence. For example, we can use span masking to corrupt the above sequence as

Original (\mathbf{x}): The 0 puppies are frolicking outside the house .
 Span Masking ($\mathbf{x}_{\text{noise}}$): The [MASK] puppies are [MASK] house .

Here the span *frolicking outside the* is replaced with a single [MASK]. 0 indicates a length-0 span, and so we insert an [MASK] between *The* and *puppies*. Span masking introduces new prediction challenges in which the model needs to know how many tokens are generated from a span. This problem is very similar to fertility modeling in machine translation [[Brown et al., 1993](#)].

If we consider a sequence consisting of multiple sentences, additional methods of corruption can be applied. In the BART model, there are two such methods.

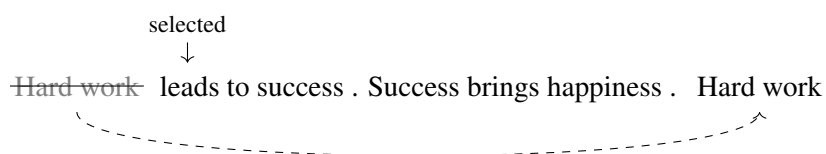
- **Sentence Reordering.** This method randomly permutes the sentences so that the model can learn to reorder sentences in a document. Consider, for example, two consecutive sentences

Hard work leads to success . Success brings happiness .

We can reorder the two sentences to have a corrupted input sequence

Success brings happiness . Hard work leads to success .

- **Document Rotation.** The goal of this task is to identify the start token of the sequence. First, a token is randomly selected from the sequence. Then, the sequence is rotated so that the selected token is the first token. For example, suppose we select the token *leads* from the above sequence. The rotated sequence is



where the subsequence *Hard work* before *leads* is appended to the end of the sequence.

For pre-training, we can apply multiple corruption methods to learn robust models, for example, we randomly choose one of them for each training sample. In practice, the outcome of encoder-decoder pre-training depends heavily on the input corruption methods used, and so we typically need to choose appropriate training objectives through careful experimentation.

1.2.4 Comparison of Pre-training Tasks

So far, we have discussed a number of pre-training tasks. Since the same training objective can apply to different architectures (e.g., using masked language modeling for both encoder-only and encoder-decoder pre-training), categorizing pre-training tasks based solely on model architecture does not seem ideal. Instead, we summarize these tasks based on the training objectives.

- **Language Modeling.** Typically, this approach refers to an auto-regressive generation procedure of sequences. At one time, it predicts the next token based on its previous context.
- **Masked Language Modeling.** Masked Language Modeling belongs to a general mask-predict framework. It randomly masks tokens in a sequence and predicts these tokens using the entire masked sequence.

- **Permuted Language Modeling.** Permuted language modeling follows a similar idea to masked language modeling, but considers the order of (masked) token prediction. It reorders the input sequence and predicts the tokens sequentially. Each prediction is based on some context tokens that are randomly selected.
- **Discriminative Training.** In discriminative training, supervision signals are created from classification tasks. Models for pre-training are integrated into classifiers and trained together with the remaining parts of the classifiers to enhance their classification performance.
- **Denoising Autoencoding.** This approach is applied to the pre-training of encoder-decoder models. The input is a corrupted sequence and the encoder-decoder models are trained to reconstruct the original sequence.

Table 1.1 illustrates these methods and their variants using examples. The use of these examples does not distinguish between models, but we mark the model architectures where the pre-training tasks can be applied. In each example, the input consists of a token sequence, and the output is either a token sequence or some probabilities. For generation tasks, such as language modeling, superscripts are used to indicate the generation order on the target side. If the superscripts are omitted, it indicates that the output sequence can be generated either autoregressively or simultaneously. On the source side, we assume that the sequence undergoes a standard Transformer encoding process, meaning that each token can see the entire sequence in self-attention. The only exception is in permuted language modeling, where an autoregressive generation process is implemented by setting attention masks on the encoder side. To simplify the discussion, we remove the token $\langle s \rangle$ from the target-side of each example.

While these pre-training tasks are different, it is possible to compare them in the same framework and experimental setup [Dong et al., 2019; Raffel et al., 2020; Lewis et al., 2020]. Note that we cannot list all the pre-training tasks here as there are many of them. For more discussions on pre-training tasks, the interested reader may refer to some surveys on this topic [Qiu et al., 2020; Han et al., 2021].

1.3 Example: BERT

In this section, we introduce BERT models, which are among the most popular and widely used pre-trained sequence encoding models in NLP.

1.3.1 The Standard Model

The standard BERT model, which is proposed in Devlin et al. [2019]’s work, is a Transformer encoder trained using both masked language modeling and next sentence prediction tasks. The loss used in training this model is a sum of the loss of the two tasks.

$$\text{Loss}_{\text{BERT}} = \text{Loss}_{\text{MLM}} + \text{Loss}_{\text{NSP}} \quad (1.17)$$

As is regular in training deep neural networks, we optimize the model parameters by minimizing this loss. To do this, a number of training samples are collected. During training, a batch of

Method	Enc	Dec	E-D	Input	Output
Causal LM		•	•		The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷
Prefix LM		•	•	[C] The kitten is	chasing ¹ the ² ball ³ . ⁴
Masked LM	•		•	[C] The kitten [M] chasing the [M] .	__ __ is __ __ ball __
MASS-style	•		•	[C] The kitten [M] [M] [M] ball .	__ __ is chasing the __ __
BERT-style	•		•	[C] The kitten [M] playing the [M] .	__ kitten is chasing __ ball __
Permuted LM	•			[C] The kitten is chasing the ball .	The ⁵ kitten ⁷ is ⁶ chasing ¹ the ⁴ ball ² . ³
Next Sentence Prediction	•			[C] The kitten is chasing the ball . Birds eat worms .	Pr(IsNext representation-of-[C])
Sentence Comparison	•			Encode a sentence as \mathbf{h}_a and another sentence as \mathbf{h}_b	Score($\mathbf{h}_a, \mathbf{h}_b$)
Token Classification	•			[C] The kitten is chasing the ball .	Pr(\cdot The) Pr(\cdot kitten) ... Pr(\cdot .)
Token Reordering			•	[C] . kitten the chasing The is ball	The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷
Token Deletion			•	[C] The kitten is chasing the ball .	The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷
Span Masking			•	[C] The kitten [M] is [M] .	The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷
Sentinel Masking			•	[C] The kitten [X] the [Y]	[X] ¹ is ² chasing ³ [Y] ⁴ ball ⁵ . ⁶
Sentence Reordering			•	[C] The ball rolls away swiftly . The kitten is chasing the ball .	The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷ The ⁸ ball ⁹ rolls ¹⁰ away ¹¹ swiftly ¹² . ¹³
Document Rotation			•	[C] chasing the ball . The ball rolls away swiftly . The kitten is	The ¹ kitten ² is ³ chasing ⁴ the ⁵ ball ⁶ . ⁷ The ⁸ ball ⁹ rolls ¹⁰ away ¹¹ swiftly ¹² . ¹³

Table 1.1: Comparison of pre-training tasks, including **language modeling**, **masked language modeling**, **permuted language modeling**, **discriminative training**, and **denoising autoencoding**. [C] = [CLS], [M] = [MASK], [X], [Y] = sentinel tokens. Enc, Dec and E-D indicate whether the approach can be applied to encoder-only, decoder-only, encoder-decoder models, respectively. For generation tasks, superscripts are used to represent the order of the tokens.

training samples is randomly selected from this collection at a time, and $\text{Loss}_{\text{BERT}}$ is accumulated over these training samples. Then, the model parameters are updated via gradient descent or its variants. This process is repeated many times until some stopping criterion is satisfied, such as when the training loss converges.

1.3.1.1 Loss Functions

In general, BERT models are used to represent a single sentence or a pair of sentences, and thus can handle various downstream language understanding problems. In this section we assume that the input representation is a sequence containing two sentences Sent_A and Sent_B , expressed as

$$[\text{CLS}] \text{ Sent}_A [\text{SEP}] \text{ Sent}_B [\text{SEP}]$$

Here we follow the notation in BERT’s paper and use [SEP] to denote the separator.

Given this sequence, we can obtain Loss_{MLM} and Loss_{NSP} separately. For masked language modeling, we predict a subset of the tokens in the sequence. Typically, a certain percentage of

the tokens are randomly selected, for example, in the standard BERT model, 15% of the tokens in each sequence are selected. Then the sequence is modified in three ways

- **Token Masking.** 80% of the selected tokens are masked and replaced with the symbol [MASK]. For example

Original: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
 Masked: [CLS] It is [MASK] . [SEP] I need [MASK] umbrella . [SEP]

where the selected tokens are underlined. Predicting masked tokens makes the model learn to represent tokens from their surrounding context.

- **Random Replacement.** 10% of the selected tokens are changed to a random token. For example

Original: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
 Random Token: [CLS] It is raining . [SEP] I need an **hat** . [SEP]

This helps the model learn to recover a token from a noisy input.

- **Unchanged.** 10% of the selected tokens are kept unchanged. For example,

Original: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
 Unchanged Token: [CLS] It is raining . [SEP] **I** need an umbrella . [SEP]

This is not a difficult prediction task, but can guide the model to use easier evidence for prediction.

Let $\mathcal{A}(\mathbf{x})$ be the set of selected positions of a given token sequence \mathbf{x} , and $\bar{\mathbf{x}}$ be the modified sequence of \mathbf{x} . The loss function of masked language modeling can be defined as

$$\text{Loss}_{\text{MLM}} = - \sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i(x_i | \bar{\mathbf{x}}) \quad (1.18)$$

where $\Pr_i(x_i | \bar{\mathbf{x}})$ is the probability of predicting x_i at the position i given $\bar{\mathbf{x}}$. Figure 1.5 shows a running example of computing Loss_{MLM} .

For next sentence prediction, we follow the method described in Section 1.2.2.3. Each training sample is classified into a label set $\{\text{IsNext}, \text{NotNext}\}$, for example,

Sequence: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
 Label: IsNext

Sequence: [CLS] The cat sleeps on the windowsill . [SEP] Apples grow on trees . [SEP]
 Label: NotNext

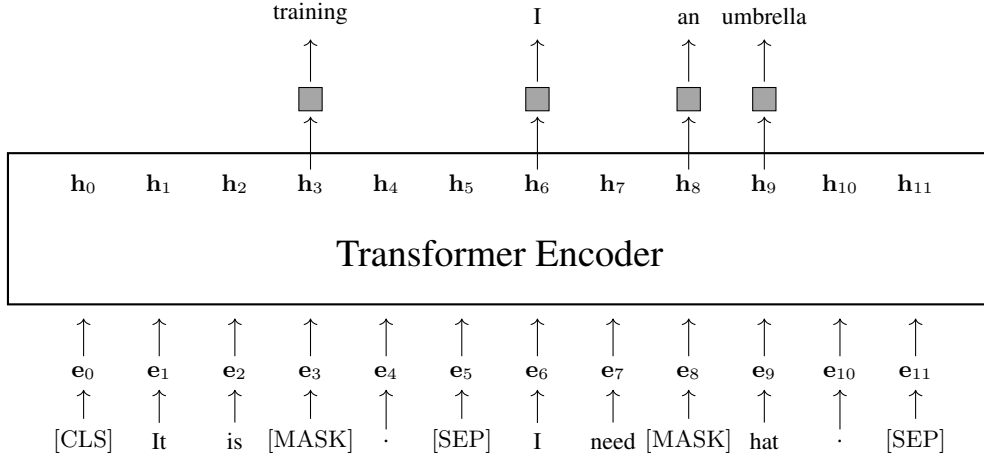
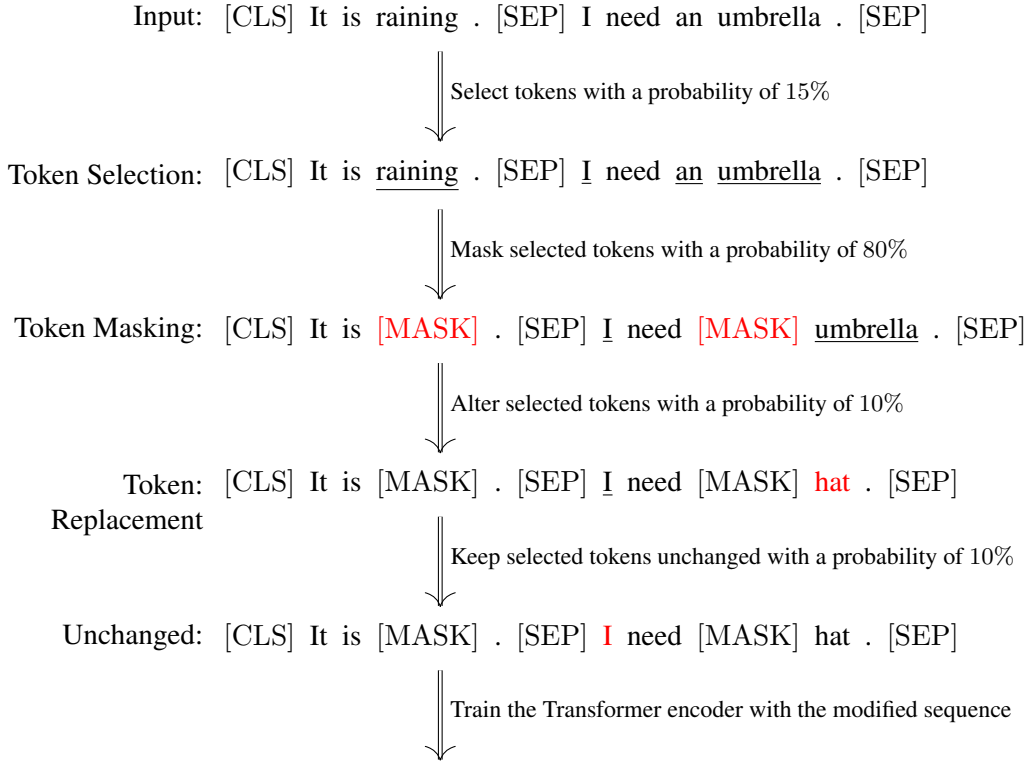


Fig. 1.5: A running example of BERT-style masked language modeling. First, 15% of the tokens are randomly selected. These selected tokens are then processed in one of three ways: replaced with a [MASK] token (80% of the time), replaced with a random token (10% of the time), or kept unchanged (10% of the time). The model is trained to predict these selected tokens based on the modified sequence. e_i represents the embedding of the token at the position i . Gray boxes represent the Softmax layers.

The output vector of the encoder for the first token [CLS] is viewed as the sequence representation, denoted by \mathbf{h}_{cls} (or \mathbf{h}_0). A classifier is built on top of \mathbf{h}_{cls} . Then, we can compute the probability of a label c given \mathbf{h}_{cls} , i.e., $\Pr(c|\mathbf{h}_{\text{cls}})$. There are many loss functions one can choose for classification problems. For example, in maximum likelihood training, we can define Loss_{NSP} as

$$\text{Loss}_{\text{NSP}} = -\log \Pr(c_{\text{gold}}|\mathbf{h}_{\text{cls}}) \quad (1.19)$$

where c_{gold} is the correct label for this sample.

1.3.1.2 Model Setup

As shown in Figure 1.6, BERT models are based on the standard Transformer encoder architecture. The input is a sequence of embeddings, each being the sum of the token embedding, the positional embedding, and the segment embedding.

$$\mathbf{e} = \mathbf{x} + \mathbf{e}_{\text{pos}} + \mathbf{e}_{\text{seg}} \quad (1.20)$$

Both the token embedding (\mathbf{x}) and positional embedding (\mathbf{e}_{pos}) are regular, as in Transformer models. The segment embedding (\mathbf{e}_{seg}) is a new type of embedding that indicates whether a token belongs to Sent_A or Sent_B . This can be illustrated by the following example.

Token	[CLS]	It	is	raining	.	[SEP]	I	need	an	umbrella	.	[SEP]
\mathbf{x}	\mathbf{x}_0	\mathbf{x}_1	\mathbf{x}_2	\mathbf{x}_3	\mathbf{x}_4	\mathbf{x}_5	\mathbf{x}_6	\mathbf{x}_7	\mathbf{x}_8	\mathbf{x}_9	\mathbf{x}_{10}	\mathbf{x}_{11}
\mathbf{e}_{pos}	PE(0)	PE(1)	PE(2)	PE(3)	PE(4)	PE(5)	PE(6)	PE(7)	PE(8)	PE(9)	PE(10)	PE(11)
\mathbf{e}_{seg}	\mathbf{e}_A	\mathbf{e}_A	\mathbf{e}_A	\mathbf{e}_A	\mathbf{e}_A	\mathbf{e}_A	\mathbf{e}_B	\mathbf{e}_B	\mathbf{e}_B	\mathbf{e}_B	\mathbf{e}_B	\mathbf{e}_B

The main part of BERT models is a multi-layer Transformer network. A Transformer layer consists of a self-attention sub-layer and an FFN sub-layer. Both of them follow the post-norm architecture: $\text{output} = \text{LNorm}(F(\text{input}) + \text{input})$, where $F(\cdot)$ is the core function of the sub-layer (either a self-attention model or an FFN), and $\text{LNorm}(\cdot)$ is the layer normalization unit. Typically, a number of Transformer layers are stacked to form a deep network. At each position of the sequence, the output representation is a real-valued vector which is produced by the last layer of the network.

There are several aspects one may consider in developing BERT models.

- **Vocabulary Size** ($|V|$). In Transformers, each input token is represented as an entry in a vocabulary V . Large vocabularies can cover more surface form variants of words, but may lead to increased storage requirements.
- **Embedding Size** (d_e). Every token is represented as a d_e -dimensional real-valued vector. As presented above, this vector is the sum of the token embedding, positional embedding, and segment embedding, all of which are also d_e -dimensional real-valued vectors.
- **Hidden Size** (d). The input and output of a sub-layer are of d dimensions. Besides, most of the hidden states of a sub-layer are d -dimensional vectors. In general, d can be roughly viewed as the width of the network.
- **Number of Heads** (n_{head}). In self-attention sub-layers, one needs to specify the number of heads used in multi-head self-attention. The larger this number is, the more sub-spaces in which attention is performed. In practical systems, we often set $n_{\text{head}} \geq 4$.
- **FFN Hidden Size** (d_{ffn}). The size of the hidden layer of the FFNs used in Transformers is typically larger than d . For example, a typical setting is $d_{\text{ffn}} = 4d$. For larger Transformers, such as recent large models, d_{ffn} may be set to a very large value.

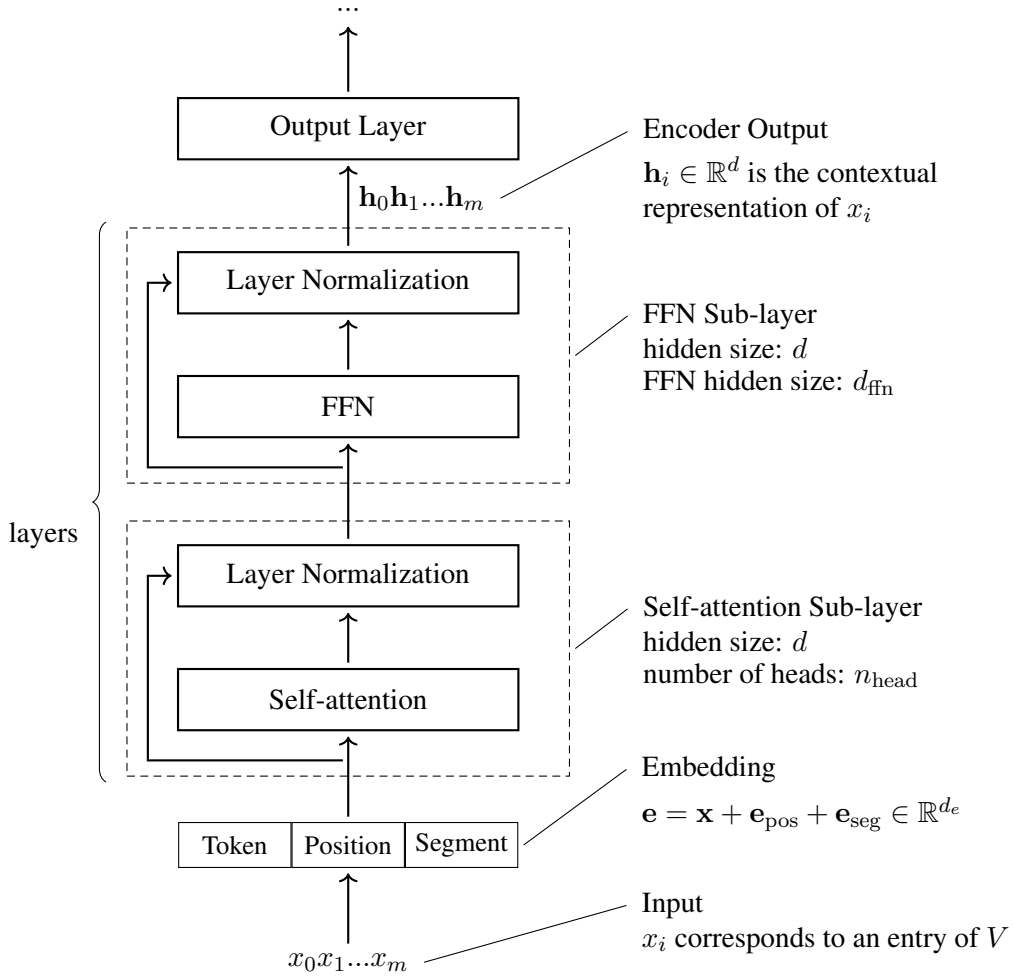


Fig. 1.6: The model architecture of BERT (Transformer encoder). The input tokens are first represented as embeddings, each of which is the sum of the corresponding token embedding, positional embedding and segment embedding. Then, the embedding sequence is processed by a stack of Transformer layers. Each layer in this stack includes a self-attention sub-layer and a FFN sub-layer. The output of the BERT model is a sequence of vectors produced by the final Transformer layer.

- **Model Depth (L).** Using deep networks is an effective way to improve the expressive power of Transformers. For BERT models, L is typically set to 12 or 24. However, networks with even greater depth are also feasible and can be applied for further enhancements.

Different settings of these hyper-parameters lead to different model sizes. There are two widely-used BERT models.

- BERT_{base}: $d = 768$, $L = 12$, $n_{\text{head}} = 12$, total number of parameters = 110M.
- BERT_{large}: $d = 1,024$, $L = 24$, $n_{\text{head}} = 16$, total number of parameters = 340M.

Training BERT models follows the standard training process of Transformers. Training larger models such as BERT_{large} requires more training effort and time. This is a common problem for pre-training, especially when a model is trained on a very large amount of data. In practice,

there are often considerations of training efficiency. For example, a practice is to first train a BERT model on relatively short sequences for a large number of training steps, and then continue training it on full-length sequences for the remaining training steps.

1.3.2 More Training and Larger Models

BERT is a milestone model in NLP, sparking many subsequent efforts to improve it. One direction is to scale up the model itself, including increasing training data and developing larger models.

RoBERTa, an extension of the standard BERT model, is an example of such efforts [Liu et al., 2019]. It introduces two major improvements. First, simply using more training data and more compute can improve BERT models without need of changing the model architectures. Second, removing the NSP loss does not decrease the performance on downstream tasks if the training is scaled up. These findings suggest exploring a general direction of pre-training: we can continue to improve pre-training by scaling it up on simple pre-training tasks.

A second approach to improving BERT models is to increase the number of model parameters. For example, in He et al. [2021]’s work, a 1.5 billion-parameter BERT-like model is built by increasing both the model depth and hidden size. However, scaling up BERT and various other pre-trained models introduces new challenges in training, for example, training very large models often becomes unstable and difficult to converge. This makes the problem more complicated, and requires careful consideration of various aspects, including model architecture, parallel computation, parameter initialization, and so on. In another example, Shuey et al. [2019] successfully trained a 3.9 billion-parameter BERT-like model, where hundreds of GPUs were used to manage the increased computational demands.

1.3.3 More Efficient Models

Compared to its predecessors, BERT is a relatively large model for the time it was proposed. This increase in model size results in larger memory requirements and a consequent slowdown in system performance. Developing smaller and faster BERT models is part of the broader challenge of building efficient Transformers, which has been extensively discussed in Tay et al. [2020]’s work and Xiao and Zhu [2023]’s work. However, a deeper discussion of this general topic is beyond the scope of our current discussion. Here we instead consider a few efficient variants of BERT.

Several threads of research are of interest to NLP researchers in developing efficient BERT models. First, work on knowledge distillation, such as training student models with the output of well-trained teacher models, shows that smaller BERT models can be obtained by transferring knowledge from larger BERT models. Given that BERT models are multi-layer networks with several different types of layers, knowledge distillation can be applied at different levels of representation. For example, beyond distilling knowledge from the output layers, it is also possible to incorporate training loss that measures the difference in output of hidden layers between teacher models and student models [Sun et al., 2020; Jiao et al., 2020]. Indeed, knowledge distillation has been one of the most widely-used techniques for learning small pre-trained models.

Second, conventional model compression methods can be directly applied to compress BERT models. One common approach is to use general-purpose pruning methods to prune the Transformer encoding networks [Gale et al., 2019]. This generally involves removing entire layers [Fan

et al., 2019] or a certain percentage of parameters in the networks [Sanh et al., 2020; Chen et al., 2020]. Pruning is also applicable to multi-head attention models. For example, Michel et al. [2019] show that removing some of the heads does not significantly decrease the performance of BERT models, but speeds up the inference of these models. Another approach to compressing BERT models is quantization [Shen et al., 2020]. By representing model parameters as low-precision numbers, the models can be greatly compressed. While this method is not specific to BERT models, it proves effective for large Transformer-based architectures.

Third, considering that BERT models are relatively deep and large networks, another thread of research uses dynamic networks to adapt these models for efficient inference. An idea in this paradigm is to dynamically choose the layers for processing a token, for example, in depth-adaptive models we exit at some optimal depth and thus skip the rest of the layers in the layer stack [Xin et al., 2020; Zhou et al., 2020]. Similarly, we can develop length-adaptive models in which the length of the input sequence is dynamically adjusted. For example, we can skip some of the tokens in the input sequence so that the model can reduce computational load on less important tokens, enhancing overall efficiency.

Fourth, it is also possible to share parameters across layers to reduce the size of BERT models. A simple way to do this is to share the parameters of a whole Transformer layer across the layer stack [Dehghani et al., 2018; Lan et al., 2020]. In addition to the reduced number of parameters, this enables reuse of the same layer in a multi-layer Transformer network, leading to savings of memory footprint at test time.

1.3.4 Multi-lingual Models

The initial BERT model was primarily focused on English. Soon after this model was proposed, it was extended to many languages. One simple way to do this is to develop a separate model for each language. Another approach, which has become more popular in recent work on large language models, is to train multi-lingual models directly on data from all the languages. In response, **multi-lingual BERT (mBERT)** models were developed by training them on text from 104 languages⁶. The primary difference from monolingual BERT models is that mBERT models use larger vocabularies to cover tokens from multiple languages. As a result, the representations of tokens from different languages are mapped into the same space, allowing for the sharing of knowledge across languages via this universal representation model.

One important application of multi-lingual pre-trained models is cross-lingual learning. In the cross-lingual setting, we learn a model on tasks in one language, and apply it to the same tasks in another language. In cross-lingual text classification, for example, we fine-tune a multi-lingual pre-trained model on English annotated documents. Then, we use the fine-tuned model to classify Chinese documents.

An improvement to multi-lingual pre-trained models like mBERT is to introduce bilingual data into pre-training. Rather than training solely on monolingual data from multiple languages, bilingual training explicitly models the relationship between tokens in two languages. The resulting model will have innate cross-lingual transfer abilities, and thus can be easily adapted to different languages. Lample and Conneau [2019] propose an approach to pre-training **cross-lingual language models (XLMs)**. In their work, a cross-lingual language model can be trained in either the causal language modeling or masked language modeling manner. For masked language modeling

⁶<https://github.com/google-research/bert/>

pre-training, the model is treated as an encoder. The training objective is the same as BERT: we maximize the probabilities of some randomly selected tokens which are either masked, replaced with random tokens, or kept unchanged in the input. If we consider bilingual data in pre-training, we sample a pair of aligned sentences each time. Then, the two sentences are packed together to form a single sequence used for training. For example, consider an English-Chinese sentence pair

鲸鱼 是 哺乳 动物 。 \leftrightarrow Whales are mammals .

We can pack them to obtain a sequence, like this

[CLS] 鲸鱼 是 哺乳 动物 。 [SEP] Whales are mammals . [SEP]

We then select a certain percentage of the tokens and replace them with [MASK].

[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]

The goal of pre-training is to maximize the product of the probabilities of the masked tokens given the above sequence. By performing training in this way, the model can learn to represent both the English and Chinese sequences, as well as to capture the correspondences between tokens in the two languages. For example, predicting the Chinese token 鲸鱼 may require the information from the English token *Whales*. Aligning the representations of the two languages essentially transforms the model into a “translation” model. So this training objective is also called **translation language modeling**. Figure 1.7 shows an illustration of this approach.

A benefit of multi-lingual pre-trained models is their inherent capability of handling code-switching. In NLP and linguistics, code-switching refers to switching among languages in a text. For example, the following is a mixed language text containing both Chinese and English:

周末 我们 打算 去 做 hiking , 你 想 一 起 来 吗 ?
(We plan to go hiking this weekend, would you like to join us?)

For multi-lingual pre-trained models, we do not need to identify whether a token is Chinese or English. Instead, every token is just an entry of the shared vocabulary. This can be imagined as creating a “new” language that encompasses all the languages we want to process.

The result of multi-lingual pre-training is influenced by several factors. Given that the model architecture is fixed, one needs to specify the size of the shared vocabulary, the number (or percentage) of samples in each language, the size of the model, and so on. [Conneau et al. \[2020\]](#) point out several interesting issues regarding large-scale multi-lingual pre-training for XLM-like models. First, as the number of supported languages increases, a larger model is needed to handle these languages. Second, a larger shared vocabulary is helpful for modeling the increased diversity in languages. Third, low-resource languages more easily benefit from cross-lingual transfer from high-resource languages, particularly when similar high-resource languages are involved in pre-training. However, **interference** may occur if the model is trained for an extended period,

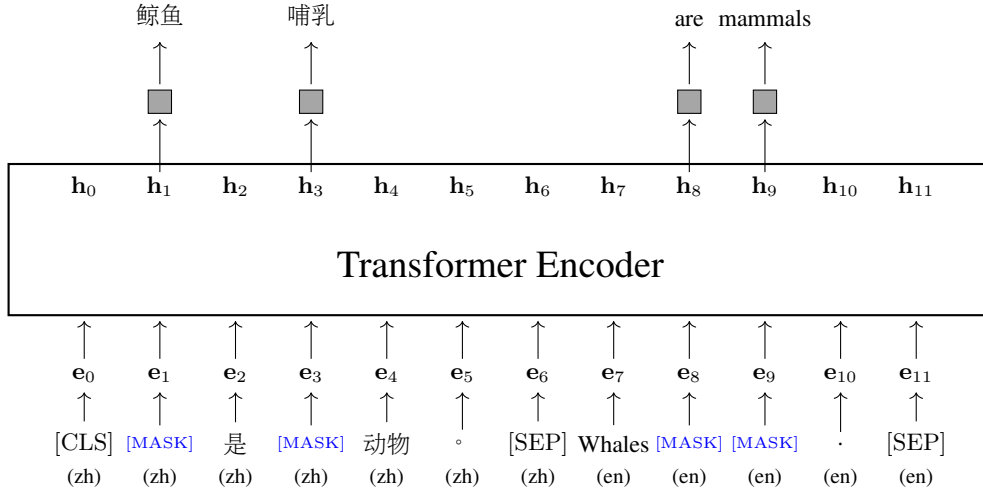


Fig. 1.7: An illustration of translation language modeling. For ease of understanding, we present a simple example where all the selected tokens are masked. The model is trained to predict these masked tokens. As the sequence contains tokens in two languages, predicting a token in one language allows access to tokens in the other language, thereby enabling cross-lingual modeling. In [Lample and Conneau \[2019\]](#)’s work, an input embedding (i.e., e_i) is the sum of the token embedding, positional embedding, and language embedding. This requires that each token is assigned with a language label. Thus we can distinguish tokens in different languages. In multi-lingual pre-training, particularly in work using shared vocabularies, specifying the language to which a token belongs is not necessary. The use of language embeddings in turn makes it difficult to handle code-switching. Therefore, we assume here that all token representations are language-independent.

meaning the overall performance of the pre-trained model starts decreasing at a certain point during pre-training. Thus, in practical systems, one may need to stop the pre-training early to prevent interference.

1.4 Applying BERT Models

Once a BERT model is pre-trained, it can then be used to solve NLP problems. But BERT models are not immediately ready for performing specific downstream tasks. In general, additional fine-tuning work is required to make them adapt. As a first step, we need a predictor to align the output of the model with the problem of interest. Let $\text{BERT}_{\hat{\theta}}(\cdot)$ be a BERT model with pre-trained parameters $\hat{\theta}$, and $\text{Predict}_{\omega}(\cdot)$ be a prediction network with parameters ω . By integrating the prediction network with the output of the BERT model, we develop a model to tackle the downstream tasks. This model can be expressed as

$$\mathbf{y} = \text{Predict}_{\omega}(\text{BERT}_{\hat{\theta}}(\mathbf{x})) \quad (1.21)$$

where \mathbf{x} is the input and \mathbf{y} is the output that fits the problem. For example, in classification problems, the model outputs a probability distribution over labels.

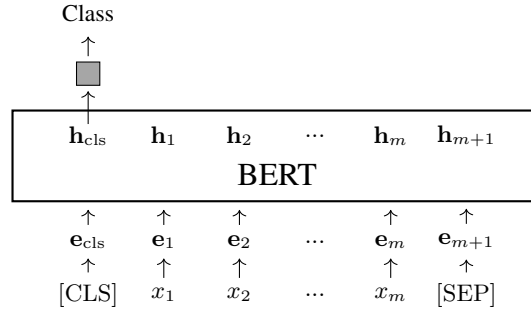
Then, we collect a set of labeled samples \mathcal{D} , and fine-tune the model by

$$(\tilde{\omega}, \tilde{\theta}) = \arg \min_{\omega, \hat{\theta}^+} \sum_{(\mathbf{x}, \mathbf{y}_{\text{gold}}) \in \mathcal{D}} \text{Loss}(\mathbf{y}_{\omega, \hat{\theta}^+}, \mathbf{y}_{\text{gold}}) \quad (1.22)$$

where $(\mathbf{x}, \mathbf{y}_{\text{gold}})$ represents a tuple of an input and its corresponding output. The notation of this equation seems a bit complicated, but the training/tuning process is standard. We optimize the model by minimizing the loss over the tuning samples. The outcome is the optimized parameters $\tilde{\omega}$ and $\tilde{\theta}$. The optimization starts with the pre-trained parameters $\hat{\theta}$. Here we use $\hat{\theta}^+$ to indicate that the parameters are initialized with $\hat{\theta}$, and use $\mathbf{y}_{\omega, \hat{\theta}^+}$ to denote the model output computed using the parameters ω and $\hat{\theta}^+$.

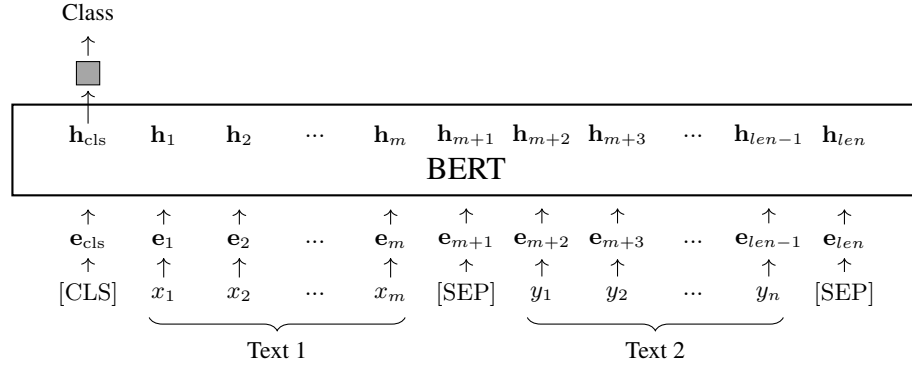
With the fine-tuned parameters $\tilde{\omega}$ and $\tilde{\theta}$, we can apply the model $\text{Predict}_{\tilde{\omega}}(\text{BERT}_{\tilde{\theta}}(\cdot))$ to new data of the same tasks for which the model was fine-tuned. The form of the downstream tasks determines the input and output formats of the model, as well as the architecture of the prediction network. In the following we list some tasks to which BERT models are generally suited.

- **Classification (Single Text).** One of the most widely-used applications of BERT models is text classification. In this task, a BERT model receives a sequence of tokens and encodes it as a sequence of vectors. The first output vector \mathbf{h}_{cls} (or \mathbf{h}_0) is typically used as the representation of the entire text. The prediction network takes \mathbf{h}_{cls} as input to produce a distribution of labels. Let $[\text{CLS}]x_1x_2\dots x_mx_m$ be an input text. See below for an illustration of BERT-based text classification.



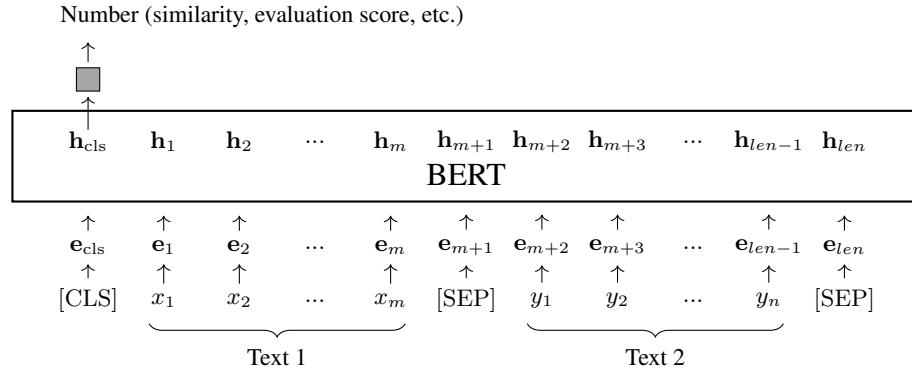
Here the gray box denotes the prediction network. Many NLP problems can be categorized as text classification tasks, and there have been several text classification benchmarks for evaluating pre-trained models. For example, we can classify texts by their grammatical correctness (grammaticality) or emotional tone (sentiment) [Socher et al., 2013; Warstadt et al., 2019]. Note that the prediction network could be any classification model, such as a deep neural network or a more traditional classification model. The entire model can then be trained or fine-tuned in the manner of a standard classification model. For example, the prediction network can be simply a Softmax layer and the model parameters can be optimized by maximizing the probabilities of the correct labels.

- **Classification (Pair of Texts).** Classification can also be performed on a pair of texts. Suppose we have two texts, $x_1\dots x_m$ and $y_1\dots y_n$. We can concatenate these texts to form a single sequence with a length len . Then, we predict a label for this combined text sequence based on the \mathbf{h}_{cls} vector, as follows



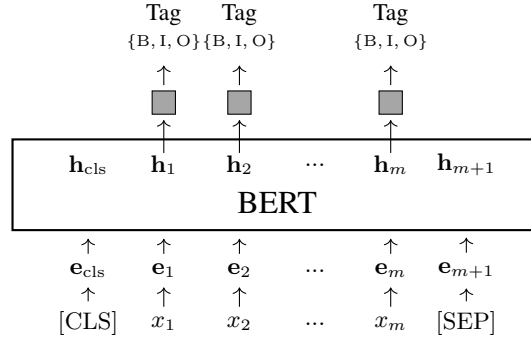
where $\text{len} = n + m + 2$. Text pair classification covers several problems, including semantic equivalence judgement (determine whether two texts are semantically equivalent) [Dolan and Brockett, 2005], text entailment judgement (determine whether a hypothesis can be logically inferred or entailed from a premise) [Bentivogli and Giampiccolo, 2011; Williams et al., 2018], grounded commonsense inference (determine whether an event is likely to happen given its context) [Zellers et al., 2018], and question-answering inference (determine whether an answer corresponds to a given question).

- **Regression.** Instead of generating a label distribution, we can have the prediction network output a real-valued score. For example, by adding a Sigmoid layer to the prediction network, the system can be employed to compute the similarity between two given sentences. The architecture is the same as that of BERT-based classification systems, with only the change of the output layer.



For training or fine-tuning, we can minimize the regression loss of the model output as usual.

- **Sequence Labeling.** Sequence labeling is a machine learning approach applicable to a wide range of NLP problems. This approach assigns a label to each token in an input sequence, and some linguistic annotations can then be derived from this sequence of labels. An example of sequence labeling in NLP is part-of-speech (POS) tagging. We label each word in a sentence with its corresponding POS tag. Another example is named entity recognition (NER) in which we label each word with an NER tag, and named entities are identified using these tags. See below for an illustration of the model architecture for NER.

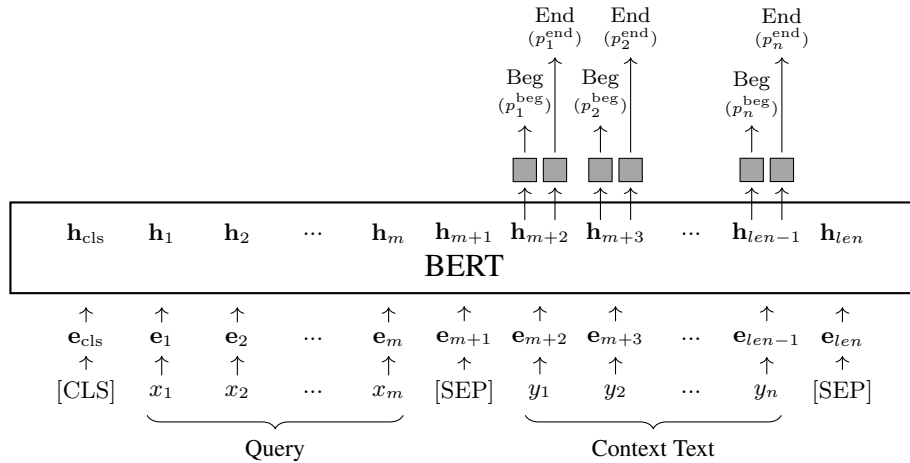


Here $\{B, I, O\}$ is the tag set of NER. For example, B-ORG means the beginning of an organization, I-ORG means the word is inside an organization, and O means the word does not belong to any named entity. This NER model can output a distribution over the tag set at each position, denoted as \mathbf{p}_i . The training or fine-tuning of the model can be performed over these distributions $\{\mathbf{p}_1, \dots, \mathbf{p}_m\}$. For example, suppose $p_i(\text{tag}_i)$ is the probability of the correct tag at position i . The training loss can be defined to be the negative likelihood

$$\text{Loss} = -\frac{1}{m} \sum_{i=1}^m \log p_i(\text{tag}_i) \quad (1.23)$$

Finding the best label sequence given a trained NER model is a well-studied issue in NLP. This is often achieved via dynamic programming, which, in the context of path finding over a lattice, has linear complexity [Huang, 2009].

- **Span Prediction.** Some NLP tasks require predicting a span in a text. A common example is reading comprehension. In this task, we are given a query $x_1 \dots x_m$ and a context text $y_1 \dots y_n$. The goal is to identify a continuous span in $y_1 \dots y_n$ that best answers the query. This problem can be framed as a sequence labeling-like task in which we predict a label for each y_j to indicate the beginning or ending of the span. Following Seo et al. [2017], we add two networks on top of the BERT output for y_j : one for generating the probability of y_j being the beginning of the span (denoted by p_j^{beg}), and one for generating the probability of y_j being the ending of the span (denoted by p_j^{end}). The resulting model architecture is shown as follows



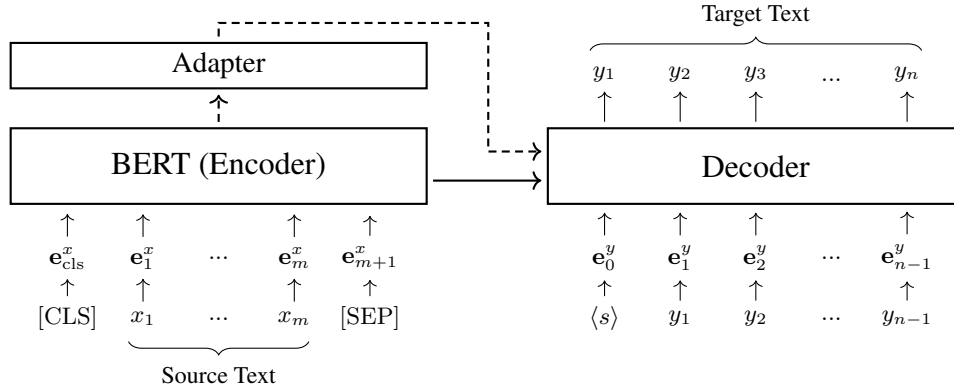
We pack the query and context text together to obtain the input sequence. The prediction networks are only applied to outputs for the context text, generating the probabilities p_j^{beg} and p_j^{end} at each position. The loss can be computed by summing the negative log likelihoods of the two models across the entire context text.

$$\text{Loss} = -\frac{1}{n} \sum_{j=1}^n (\log p_j^{\text{beg}} + \log p_j^{\text{end}}) \quad (1.24)$$

At test time, we search for the best span by

$$(\hat{j}_1, \hat{j}_2) = \arg \max_{1 \leq j_1 \leq j_2 \leq n} (\log p_{j_1}^{\text{beg}} + \log p_{j_2}^{\text{end}}) \quad (1.25)$$

- **Encoding for Encoder-Decoder Models.** While our focus in this section has been primarily on language understanding problems, it is worth noting that BERT models can be applied to a broader range of NLP tasks. In fact, BERT models can be used in all the scenarios where we need to encode a piece of text. One application that we have not mentioned is text generation which includes a range of tasks such as machine translation, summarization, question answering, and dialogue generation. These tasks can be formulated as sequence-to-sequence problems: we use an encoder to represent the source text, and a decoder to generate the corresponding target text. A straightforward method to apply BERT models is to consider them as encoders. Before fine-tuning, we can initialize the parameters of the encoder with those from a pre-trained BERT model. Then, the encoder-decoder model can be fine-tuned on pairs of texts as usual. The following shows the architecture of a neural machine translation system where a BERT model is applied on the source side.



Here $x_1 \dots x_m$ denotes the source sequence, $y_1 \dots y_n$ denotes the target sequence, $e_1^x \dots e_m^x$ denotes the embedding sequence of $x_1 \dots x_m$, and $e_1^y \dots e_n^y$ denotes the embedding sequence of $y_1 \dots y_n$. The adapter, which is optional, maps the output of the BERT model to the form that is better suited to the decoder.

Fine-tuning BERT models is a complicated engineering problem, influenced by many factors, such as the amount of fine-tuning data, the model size, and the optimizer used in fine-tuning. In general, we wish to fine-tune these models sufficiently so that they can perform well in the downstream tasks. However, fine-tuning BERT models for specific tasks may lead to overfitting,

which in turn reduces their ability to generalize to other tasks. For example, suppose we have a BERT model that performs well on a particular task. If we then fine-tune it for new tasks, this may decrease its performance on the original task. This problem is related to the **catastrophic forgetting** problem in continual training, where a neural network forgets previously learned information when updated on new samples. In practical applications, a common way to alleviate catastrophic forgetting is to add some old data into fine-tuning and train the model with more diverse data. Also, one may use methods specialized to catastrophic forgetting, such as experience replay [Rolnick et al., 2019] and elastic weight consolidation [Kirkpatrick et al., 2017]. The interested reader can refer to some surveys for more detailed discussions of this issue in continual learning [Parisi et al., 2019; Wang et al., 2023a;e].

1.5 Summary

In this chapter we have discussed the general idea of pre-training in NLP. In particular, we have discussed self-supervised pre-training and its application to encoder-only, decoder-only, and encoder-decoder architectures. Moreover, we have presented and compared a variety of pre-training tasks for these architectures. As an example, BERT is used to illustrate how sequence models are pre-trained via masked language modeling and applied to different downstream tasks.

Recent years have shown remarkable progress in NLP, led by the large-scale use of self-supervised pre-training. And sweeping advances are being made across many tasks, not only in NLP but also in computer vision and other areas of AI. One idea behind these advances is that a significant amount of knowledge about the world can be learned by simply training these AI systems on huge amounts of unlabeled data. For example, a language model can learn some general knowledge of a language by repeatedly predicting masked words in large-scale text. As a result, this pre-trained language model can serve as a foundation model, which can be easily adapted to address specific downstream NLP tasks. This paradigm shift in NLP has enabled the development of incredibly powerful systems for language understanding, generation, and reasoning [Manning, 2022]. However, it is important to recognize that we are still in the early stages of creating truly intelligent systems, and there is a long way to go. Nevertheless, large-scale pre-training has opened a door to intelligent systems that researchers have long aspired to develop, though several key research areas remain open for exploration, such as learning intelligence efficiently using reasonably small-sized data and acquiring complex reasoning and planning abilities.

Note that this chapter is mostly introductory and cannot cover all aspects of pre-training. For example, there are many methods to fine-tune a pre-trained model, offering different ways to better adapt the model to diverse situations. Moreover, large language models, which are considered one of the most significant achievements in AI in recent years, are skipped in this section. We leave the discussion of these topics to the following chapters.

CHAPTER 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 $\{x_0, x_1, \dots, x_m\}$ 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

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

or alternatively in a logarithmic form

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

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

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

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

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 $\Pr(x_i | x_0, \dots, x_{i-1})$ is maximized. This token is then appended to the end of the context sequence. In the next step, we repeat the same process, but based on the new context.

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

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

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

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 $\{x_0, \dots, x_{m-1}\}$). For each step, an output token is generated, shifting the sequence one position forward for the next prediction. To do this, the language model outputs a distribution $\Pr(\cdot | x_0, \dots, x_{i-1})$ at each position i , and the token x_i is selected according to this distribution. This model is trained by maximizing the log likelihood $\sum_{i=1}^m \log \Pr(x_i | x_0, \dots, x_{i-1})$ 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 $\{\mathbf{e}_0, \dots, \mathbf{e}_{m-1}\}$. \mathbf{e}_i is the sum of the token embedding of x_i and the positional embedding of i . The major body of the model is a stack of Transformer blocks (or layers). Each Transformer block has two stacked sub-layers, one for self-attention modeling and one for FFN modeling. These sub-layers can be defined using the post-norm architecture

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

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

or the pre-norm architecture

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

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

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

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 $-\infty$ otherwise.

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

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

where $\text{Merge}(\cdot)$ represents 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

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

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

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

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

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

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

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

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

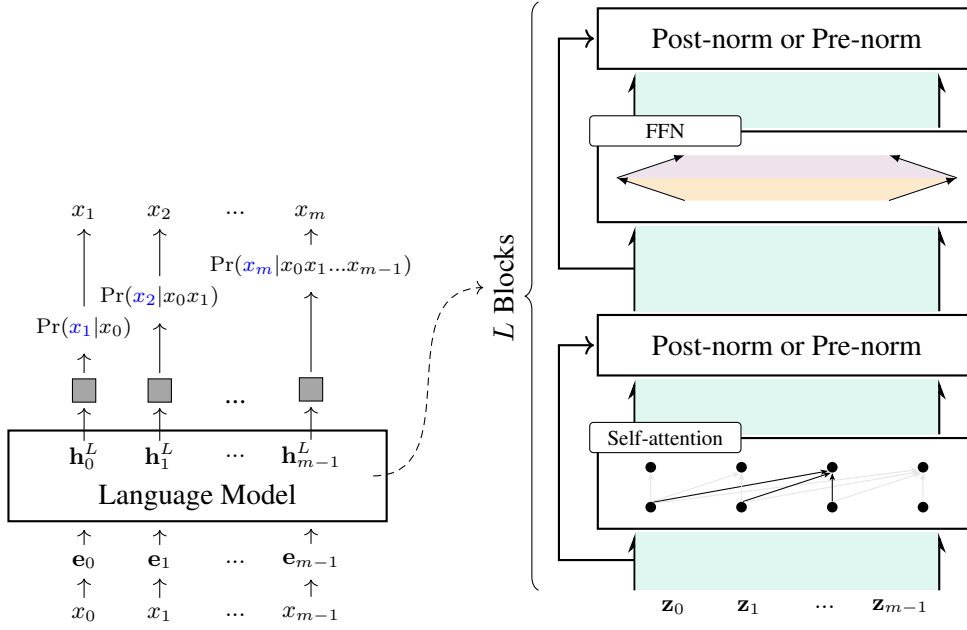


Fig. 2.1: The Transformer-decoder architecture for language modeling. The central components are L stacked Transformer blocks, each comprising a self-attention sub-layer and an FFN sub-layer. To prevent the model from accessing the right-context, a masking variable is incorporated into self-attention. 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. $\{z_0, \dots, z_{m-1}\}$ denote the inputs of a Transformer block, and $\{h_0^L, \dots, h_{m-1}^L\}$ denote the outputs of the last Transformer block.

model follows an autoregressive process. Each time the language model takes a token x_{i-1} as input and predicts a token x_i that maximizes the probability $\Pr(x_i | x_0, \dots, x_{i-1})$. It is important to note that, despite different implementation details, many LLMs share the same architecture described above. These models are called large because both their depth and width are significant. Table 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 $\mathbf{x} = x_0 \dots x_m$ in \mathcal{D} can be calculated using a language model

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

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

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

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., 2024a]	671B	61	7,168	128/128
Falcon [Penedo et al., 2023]	7B	32	4,544	71/71
	40B	60	8,192	128/128
	180B	80	14,848	232/232
Mistral [Jiang et al., 2023a]	7B	32	4,096	32/32

Table 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. In Section 2.2 we will present more discussions on these issues.

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

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

Here $\sum_{i=1}^n \log \Pr(y_i|x_0, \dots, x_m, y_1, \dots, y_{i-1})$ essentially expresses the same thing as the right-hand side of Eq. (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 $\mathbf{x} = \text{“John seems happy today .\n Question : Is this sentence grammatically correct?\n Answer :”}$ 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., $y = \text{“Yes”}$) 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 information⁵. To illustrate, consider the following examples (blue text = input and underlined text = output).

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

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

Yes

LLMs are powerful models but are expensive to build.
Does this sentence make sense grammatically?

Answer Option

- Yes
- No

Yes

Yes or No? Yes = the following sentence is grammatically correct. No = it contains grammatical errors.

LLMs are powerful models but are expensive to build.

Yes

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

Write a short story about three characters.

Character 1: A boy whose name is Andy.

Character 2: Andy's dog.

Character 3: Andy's friend, Jane.

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

Show me a recipe for making ice cream.

Ingredients:

2 cups heavy cream; 1 cup milk; 3/4 cup sugar; 2 tsp vanilla extract;
Pinch of salt.

Steps:

1. Whisk all ingredients until sugar dissolves.
2. Chill mixture for 2 hours in the fridge.
3. Freeze in a shallow dish, stirring every 30 minutes, 3-4 times.
4. Freeze solid for 2 hours.
5. Serve your homemade ice cream.

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

\$6.00

Write a Python program to calculate the sum of squares of the following numbers.

1 , 2 , 10 , -9 , 78

`numbers = [1,2,10,-9,78]`

`sum_of_squares = sum(x**2 for x in numbers)`

`print(sum_of_squares)`

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

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

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

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

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

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

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

We then define the loss function to be

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

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.

Figure 2.2 shows an overview of RLHF. Given that this section serves only as a brief introduction to concepts of LLMs, a detailed discussion of RLHF techniques will not be included. We instead illustrate the basic ideas behind RLHF using a simple example.

Suppose we have trained an LLM via pre-training and instruction fine-tuning. This LLM is deployed to respond to requests from users. For example, a user may input

How can I live a more environmentally friendly life?

We use the LLM to generate 4 different outputs (denoted by $\{\mathbf{y}_1, \dots, \mathbf{y}_4\}$) by sampling the output space

- Output 1 (\mathbf{y}_1): Consider switching to an electric vehicle or bicycle instead of traditional cars to reduce carbon emissions and protect our planet.
- Output 2 (\mathbf{y}_2): Adopt a minimalist lifestyle. Own fewer possessions to reduce consumption and the environmental impact of manufacturing and disposal.
- Output 3 (\mathbf{y}_3): Go off-grid. Generate your own renewable energy and collect rainwater to become completely self-sufficient and reduce reliance on non-renewable resources.
- Output 4 (\mathbf{y}_4): Support local farm products to reduce the carbon footprint of transporting food, while enjoying fresh, healthy food.

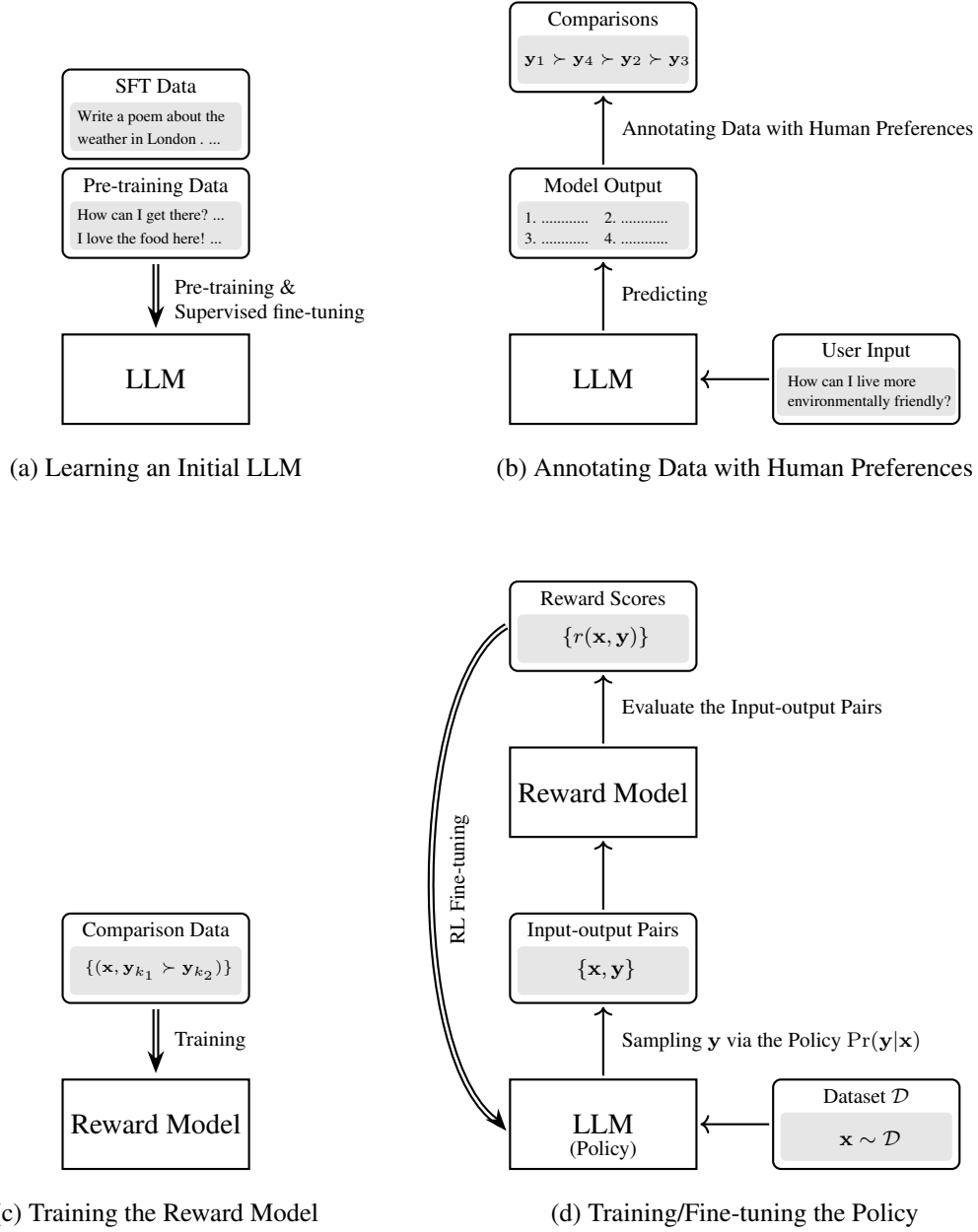


Fig. 2.2: An overview of RLHF. There are 4 key steps involved: a) training an initial LLM (i.e., policy) using pre-training and supervised fine-tuning; b) collecting human preference data by ranking the outputs of the LLM; c) training a reward model using the ranking results; d) RL fine-tuning of the policy based on the reward model. Double line arrows mean training or fine-tuning.

We then ask annotators to evaluate these outputs. One straightforward way is to assign a rating score to each output. In this case, the reward model learning problem can be framed as a task of training a regression model. But giving numerical scores to LLM outputs is not an easy task for annotators. It is usually difficult to design an annotation standard that all annotators can agree on and easily follow. An alternative method, which is more popular in the development of LLMs, is to rank these outputs. For example, a possible ranking of the above outputs is

$$y_1 \succ y_4 \succ y_2 \succ y_3$$

A reward model is then trained using this ranking result. In general, a reward model in RLHF is a language model that shares the same architecture as the target LLM, but with a smaller model size. Given the input \mathbf{x} and output \mathbf{y}_k , we concatenate them to form a sequence $\text{seq}_k = [\mathbf{x}, \mathbf{y}_k]$. This sequence is processed from left to right using forced decoding. Since each position can only access its left context in language modeling, the output of the top-most Transformer layer at the first position cannot be used as the representation of the sequence. Instead, a special symbol (e.g., $\langle s \rangle$) is added to the end of the sequence, and the corresponding output of the Transformer layer stack is considered as the representation of the entire sequence. An output layer, such as a linear transformation layer, is built on top of this representation to generate the reward, denoted by $R(\text{seq}_k)$ or $R(\mathbf{x}, \mathbf{y}_k)$.

We train this reward model using ranking loss. For example, a pair-wise ranking loss function can be written in the form

$$\text{Loss}_\omega(\mathcal{D}_r) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2}) \sim \mathcal{D}_r} \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \quad (2.19)$$

where ω represents the parameters of the reward model, and \mathcal{D}_r represents a set of tuples of an input and a pair of outputs. $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2}) \sim \mathcal{D}_r$ is a sampling operation which draws a sample $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2})$ from \mathcal{D}_r with some probability. As an example, suppose we first draw a model input \mathbf{x} with a uniform distribution and then draw a pair of model outputs with a probability of $\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}$ given \mathbf{x} (denoted by $\Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x})$). The corresponding loss function is given by

$$\begin{aligned} & \text{Loss}_\omega(\mathcal{D}_r) \\ &= -\sum \Pr(\mathbf{x}) \cdot \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \\ &= -\frac{1}{K} \sum \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2} | \mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \end{aligned} \quad (2.20)$$

where K represents the number of model inputs involved in sampling. While the form of these functions may seem complex, their idea is simple: we penalize the model if the predicted ranking of two outputs differs from the human-labeled ranking. By contrast, the model receives a bonus, if the predicted ranking matches the human-labeled ranking.

We can train the reward model by minimizing the above ranking loss

$$\hat{\omega} = \arg \min_{\omega} \text{Loss}_\omega(\mathcal{D}_r) \quad (2.21)$$

The resulting model $R_{\hat{\omega}}(\cdot)$ can be employed to evaluate any given pair of input and output. Note that although the reward model is trained using a ranking-based objective, it is used for scoring. This allows it to provide continuous supervision signals, which is very beneficial for training other models.

We now turn to the policy learning problem. A commonly adopted objective is to maximize the reward on a set of input-output pairs. Following an analogous form of Eq. (2.16), we obtain a simple training objective for RL fine-tuning

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \mathbb{E}_{(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+}) \sim \mathcal{D}_{\text{rlft}}} R_{\hat{\omega}}(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+}) \quad (2.22)$$

where the optimal parameters $\tilde{\theta}$ are obtained by fine-tuning the pre-trained parameters $\hat{\theta}$. $\mathcal{D}_{\text{rlft}}$ is

the RL fine-tuning dataset. For each sample $(\mathbf{x}, \mathbf{y}_{\hat{\theta}+})$, \mathbf{x} is sampled from a prepared dataset of input sequences, and $\mathbf{y}_{\hat{\theta}+}$ is sampled from the distribution $\Pr_{\hat{\theta}+}(\mathbf{y}|\mathbf{x})$ given by the policy.

In practice, more advanced reinforcement learning algorithms, such as **proximal policy optimization (PPO)**, are often used for achieving more stable training, as well as better performance. We leave the detailed discussion of reinforcement learning algorithms to the following parts of this book where RLHF is extensively used for alignment.

An interesting question arises here: why not consider learning from human preferences as a standard supervised learning problem? This question is closely related to our aforementioned discussion on the difficulty of data annotation. Often, describing human values and goals is challenging, and it is even more difficult for humans to provide outputs that are well aligned. As an alternative, annotating the preferences of a given list of model outputs offers a simpler task. By doing so, we can create a model that understands human preferences, which can then be used as a reward model for training policies. From the perspective of machine learning, RLHF is particularly useful for scenarios where the desired behavior of an agent is difficult to demonstrate but can be easily recognized by humans. Another advantage of RLHF is its ability to explore the sample space. By employing sampling techniques, models trained with reinforcement learning can venture beyond the annotated data set to explore additional samples. This exploratory ability allows RLHF to discover potentially beneficial policies that are not immediately apparent from the labeled data alone.

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 debugging⁶.

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 partner⁷.

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

⁷To 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., 2022c]. In COT prompting, we decompose complex reasoning problems into multiple problem-solving intermediate steps. These steps are

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

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

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

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

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

Tom initially has 12 marbles. After winning a game, he gains 7 more, increasing his total to 19 marbles. He loses 5 marbles the next day, bringing his total down to 14 marbles. His brother gifts him 3 more marbles, increasing his total to 17 marbles. Therefore, Tom now has 17 marbles. So the answer is 17.

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

Jack starts with 7 apples. He eats 2 apples for dinner, so we subtract 2 from 7, leaving him with 5 apples. His mom gives him 5 more apples, adding 5 to his current total. Now he has $5 + 5 = 10$ apples. The next day, Jack gives 3 apples to his friend John. Subtracting 3 from 10, Jack is left with 7 apples. Therefore, Jack has 7 apples left. So the answer is 7.

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

Furthermore, a special kind of COT, **zero-shot COT**, introduces instructions that provoke LLMs to think, without presenting any intermediate reasoning steps in the prompts [Kojima et al., 2022]. An interesting example is that by appending a simple sentence “Let’s think step by step.” to the end of a prompt, LLMs can generate reasoning steps to reach the final answer.

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

Let's think step by step.

1. Initial Quantity: Jack starts with 7 apples.
 2. After Dinner: He eats 2 apples, so $7 - 2 = 5$ apples remain.
 3. His Mom Gives More: His mom gives him 5 more apples, so $5 + 5 = 10$ apples are now with him.
 4. Giving to John: The next day, Jack gives 3 apples to his friend John, so $10 - 3 = 7$ apples are left.
- In the end, Jack has 7 apples left.

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

Given the following text snippets, classify their sentiment as Positive, Negative, or Neutral.

Example 1: "I had an amazing day at the park!"

Sentiment: Positive

Example 2: "The service at the restaurant was terrible."

Sentiment: Negative

Example 3: "I think it's going to rain today."

Sentiment: Neutral

Text: "This movie was a fantastic journey through imagination."

Sentiment: Positive

Translate the following Chinese phrases into English.

Example 1: "你好"

Translation: "Hello"

Example 2: "谢谢你"

Translation: "Thank you"

Phrase to translate: "早上好"

Translation: "Good Morning"

LLM	# of Tokens	Data
GPT3-175B [Brown et al., 2020]	0.5T	Webpages, Books, Wikipedia
Falcon-180B [Almazrouei et al., 2023]	3.5T	Webpages, Books, Conversations, Code, Technical Articles
LLaMA2-65B [Touvron et al., 2023a]	1.0T ~ 1.4T	Webpages, Code, Wikipedia, Books, Papers, Q&As
PaLM-450B [Chowdhery et al., 2022]	0.78T	Webpages, Books, Conversations, Code, Wikipedia, News
Gemma-7B [Gemma Team, 2024]	6T	Webpages, Mathematics, Code

Table 2.3: Amounts of training data used in some LLMs in terms of the number of tokens.

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.

2.2 Training at Scale

As a first step in developing LLMs, we need to train these models on large amounts of data. The training task is itself standard: the objective is to maximize the likelihood, which can be achieved via gradient descent. However, as we scale up both the model size and the amount of data, the problem becomes very challenging, for example, large models generally make the training unstable. In this section, we discuss several issues of large-scale training for LLMs, including data preparation, model modification, and distributed training. We also discuss the scaling laws for LLMs, which help us understand their training efficiency and effectiveness.

2.2.1 Data Preparation

The importance of data cannot be overstated in NLP. As larger neural networks are developed, the demand for data continues to increase. For example, developing LLMs may require trillions of tokens in pre-training (see Table 2.3), orders of magnitude larger than those used in training conventional NLP models. In general, we may want to gather as much training data as possible. However, larger training datasets do not mean better training results, and the development of LLMs raises new issues in creating or collecting these datasets.

A first issue is the quality of data. High-quality data has long been seen as crucial for training data-driven NLP systems. Directly using raw text from various sources is in general undesirable. For example, a significant portion of the data used to train recent LLMs comes from web scraping, which may contain errors and inappropriate content, such as toxic information and fabricated facts. Also, the internet is flooded with machine-generated content due to the widespread use of AI, presenting further challenges for processing and using web-scraped data. Researchers have found that training LLMs on unfiltered data is harmful [Raffel et al., 2020]. Improving data quality typically involves incorporating filtering and cleaning steps in the data processing workflow. For example, Penedo et al. [2023] show that by adopting a number of data processing techniques, 90%

of their web-scraped data can be removed for LLM training. In addition to large-scale web-scraped data, LLM training data often includes books, papers, user-generated data on social media, and so on. Most of the latest LLMs are trained on such combined datasets, which are found to be important for the strong performance of the resulting models.

A second issue is the diversity of data. We want the training data to cover as many types of data as possible, so that the trained models can adapt to different downstream tasks easily. It has been widely recognized that the quality and diversity of training data both play very important roles in LLMs. An interesting example is that incorporating programming code into training data has been found to be beneficial for LLMs. The benefits are demonstrated not only in enhancing the programming abilities of LLMs, but also in improving reasoning for complex problems, especially those requiring COT prompting. The concept “diversity” can be extended to include language diversity as well. For example, many LLMs are trained on multi-lingual data, and therefore we can handle multiple languages using a single model. While this approach shows strong abilities in multi-lingual and cross-lingual tasks, its performance on specific languages largely depends on the volume and quality of the data for those languages. It has been shown in some cases to provide poor results for low-resource languages.

A third issue is the bias in training data. This is not a problem that is specific to LLMs but exists in many NLP systems. A common example is gender bias, where LLMs show a preference for one gender over another. This can partly be attributed to class imbalance in the training data, for example, the term *nurses* is more often associated with women. In order to debias the data, it is common practice to balance the categories of different language phenomena, such as gender, ethnicity, and dialects. The bias in data is also related to the diversity issue mentioned above. For example, since many LLMs are trained and aligned with English-centric data, they are biased towards the cultural values and perspectives prevalent among English-speaking populations. Increasing language diversity in training data can somewhat mitigate the bias.

Another issue with collecting large-scale data is the privacy concern. If LLMs are trained on data from extensive sources, this potentially leads to risks regarding the exposure of sensitive information, such as intellectual property and personal data. This is particularly concerning given the capacity of LLMs to represent patterns from the data they are trained on, which might inadvertently involve memorizing and reproducing specific details. A simple approach to privacy protection is to remove or anonymize sensitive information. For example, anonymization techniques can be applied to remove personally identifiable information from training data to prevent LLMs from learning from such data. However, in practice, erasing or redacting all sensitive data is difficult. Therefore, many LLMs, particularly those launched for public service, typically work with systems that can detect the potential exposure of sensitive data, or are fine-tuned to reject certain requests that could lead to information leakage.

2.2.2 Model Modifications

Training LLMs is difficult. A commonly encountered problem is that the training process becomes more unstable as LLMs get bigger. For example, one needs to choose a small learning rate to achieve stable training with gradient descent, but this in turn results in much longer training times. Sometimes, even when the training configuration is carefully designed, training may diverge at certain points during optimization. The training of LLMs is generally influenced by many factors, such as parameter initialization, batching, and regularization. Here, we focus on common

modifications and improvements to the standard Transformer architecture, which are considered important in developing trainable LLMs.

2.2.2.1 Layer Normalization with Residual Connections

Layer normalization is used to stabilize training for deep neural networks. It is a process of subtracting the mean and dividing by the standard deviation. By normalizing layer output in this way, we can effectively reduce the covariate shift problem and improve the training stability. In Transformers, layer normalization is typically used together with residual connections. As described in Section 2.1.1, a sub-layer can be based on either the post-norm architecture, in which layer normalization is performed right after a residual block, or the pre-norm architecture, in which layer normalization is performed inside a residual block. While both of these architectures are widely used in Transformer-based systems [Wang et al., 2019], the pre-norm architecture has proven to be especially useful in training deep Transformers. Given this, most LLMs are based on the pre-norm architecture, expressed as $\text{output} = \text{LNorm}(F(\text{input})) + \text{input}$.

A widely-used form of the layer normalization function is given by

$$\text{LNorm}(\mathbf{h}) = \alpha \cdot \frac{\mathbf{h} - \mu}{\sigma + \epsilon} + \beta \quad (2.23)$$

where \mathbf{h} is a d -dimensional real-valued vector, μ is the mean of all the entries of \mathbf{h} , and σ is the corresponding standard deviation. ϵ is introduced for the sake of numerical stability. $\alpha \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ are the gain and bias terms.

A variant of layer normalization, called root mean square (RMS) layer normalization, only re-scales the input vector but does not re-center it [Zhang and Sennrich, 2019]. The RMS layer normalization function is given by

$$\text{LNorm}(\mathbf{h}) = \alpha \cdot \frac{\mathbf{h}}{\sigma_{\text{rms}} + \epsilon} + \beta \quad (2.24)$$

where σ_{rms} is the root mean square of \mathbf{h} , that is, $\sigma_{\text{rms}} = (\frac{1}{d} \sum_{k=1}^d h_k^2)^{\frac{1}{2}}$. This layer normalization function is used in LLMs like the LLaMA series.

2.2.2.2 Activation Functions in FFNs

In Transformers, FFN sub-layers are designed to introduce non-linearities into representation learning, and are found to be useful for preventing the representations learned by self-attention from degeneration⁸ [Dong et al., 2021]. A standard form of the FFNs used in these sub-layers can be expressed as

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_f + \mathbf{b}_f \quad (2.25)$$

where $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$, $\mathbf{b}_h \in \mathbb{R}^{d_h}$, $\mathbf{W}_f \in \mathbb{R}^{d_h \times d}$, and $\mathbf{b}_f \in \mathbb{R}^d$ are the parameters, and d_h is the hidden size. $\sigma(\cdot)$ is the activation function of the hidden layer. A common choice for $\sigma(\cdot)$ is the

⁸Here degeneration refers to the phenomenon in which the rank of a matrix is reduced after some processing.

rectified linear unit (ReLU), given by

$$\sigma_{\text{relu}}(\mathbf{h}) = \max(0, \mathbf{h}) \quad (2.26)$$

In practical implementations, increasing d_h is helpful and thus it is often set to a larger number in LLMs. But a very large hidden size poses challenges for both training and deployment. In this case, the design of the activation function plays a relatively more important role in wide FFNs. There are several alternatives to the ReLU in LLMs. One of these is the **gaussian error linear unit (GeLU)** which can be seen as a smoothed version of the ReLU. Rather than controlling the output by the sign of the input, the GeLU function weights its input by the percentile $\Pr(h \leq \mathbf{h})$. Here h is a d -dimensional vector whose entries are drawn from the standard normal distribution $\text{Gaussian}(0, 1)$ ⁹. Specifically, the GeLU function is defined to be

$$\begin{aligned} \sigma_{\text{gelu}}(\mathbf{h}) &= \mathbf{h} \Pr(h \leq \mathbf{h}) \\ &= \mathbf{h} \Phi(\mathbf{h}) \end{aligned} \quad (2.27)$$

where $\Phi(\mathbf{h})$ is the cumulative distribution function of $\text{Gaussian}(0, 1)$, which can be implemented in convenient ways [Hendrycks and Gimpel, 2016]. The GeLU function has been adopted in several LLMs, such as BERT, GPT-3, and BLOOM.

Another family of activation functions which is popular in LLMs is **gated linear unit (GLU)**-based functions. The basic form of GLUs is given by

$$\sigma_{\text{glu}}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (2.28)$$

where $\mathbf{W}_1 \in \mathbb{R}^{d \times d}$, $\mathbf{b}_1 \in \mathbb{R}^d$, $\mathbf{W}_2 \in \mathbb{R}^{d \times d}$, and $\mathbf{b}_2 \in \mathbb{R}^d$ are model parameters. Different choices of $\sigma(\cdot)$ result in different versions of GLU functions. For example, if $\sigma(\cdot)$ is defined to be the GeLU function, we will have the GeGLU function

$$\sigma_{\text{geglu}}(\mathbf{h}) = \sigma_{\text{gelu}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (2.29)$$

This activation function has been successfully applied in LLMs like Gemma.

As another example, consider $\sigma(\cdot)$ to be the Swish function $\sigma_{\text{swish}}(\mathbf{h}) = \mathbf{h} \odot \text{Sigmoid}(c\mathbf{h})$ [Ramachandran et al., 2017]. Then, the SwiGLU function is given by

$$\sigma_{\text{swiglu}}(\mathbf{h}) = \sigma_{\text{swish}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \quad (2.30)$$

Both the PaLM and LLaMA series are based on the SwiGLU function. For more discussions of GLUs, the reader can refer to Shazeer [2020]’s work.

⁹ $\Pr(h \leq \mathbf{h})$ is an informal notation. It refers to a vector, with each entry representing the percentile for the corresponding entry of \mathbf{h} .

2.2.2.3 Removing Bias Terms

Another popular model design is to remove the bias terms in affine transformations used in LLMs. This treatment can be applied to layer normalization, transformations of the inputs to QKV attention, and FFNs. For example, we can modify Eq. (2.25) to obtain an FFN with no bias terms

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h)\mathbf{W}_f \quad (2.31)$$

[Chowdhery et al. \[2022\]](#) report that removing bias terms helps improve the training stability of LLMs. This method has been used in several recent LLMs, such as LLaMA and Gemma.

2.2.2.4 Other Issues

Many LLMs also involve modifications to their positional embedding models. For example, one can replace sinusoidal positional encodings with rotary position embeddings so that the learned LLMs can handle long sequences better. These models will be discussed in Section 2.3.

Note that while model modifications are common in training LLMs, the stability of training can be improved in many different ways. For example, increasing the batch size as the training proceeds has been found to be useful for some LLMs. In general, achieving stable and efficient large-scale LLM training requires carefully designed setups, including learning schedules, optimizer choices, training parallelism, mixed precision training, and so on. Some of these issues are highly engineered, and therefore, we typically need a number of training runs to obtain satisfactory LLMs.

2.2.3 Distributed Training

Training LLMs requires significant amounts of computational resources. A common approach to improving training efficiency is to use large-scale distributed systems. Fortunately, alongside the rise of neural networks in AI, deep learning-oriented software and hardware have been developed, making it easier to implement LLMs and perform computations. For example, one can now easily fine-tune an LLM using deep learning software frameworks and a machine with multiple GPUs. However, scaling up the training of LLMs is still challenging, and requires significant efforts in developing hardware and software systems for stable and efficient distributed training.

An important consideration of distributed training is parallelism. There are several forms of parallelism: data parallelism, model parallelism, tensor parallelism, and pipeline parallelism. Despite different ways to distribute computations across devices, these parallelism methods are based on a similar idea: the training problem can be divided into smaller tasks that can be executed simultaneously. The issue of parallelism in training LLMs has been extensively studied [[Narayanan et al., 2021](#); [Fedus et al., 2022](#)]. Here we sketch the basic concepts.

- **Data Parallelism.** This method is one of the most widely used parallelism methods for training neural networks. To illustrate, consider the simplest case where the standard delta rule is used in gradient descent

$$\theta_{t+1} = \theta_t - lr \cdot \frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \quad (2.32)$$

where the new parameters θ_{t+1} is obtained by updating the latest parameters θ_t with a small step lr in the direction of the negative loss gradient. $\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t}$ is the gradient of the loss with respect to the parameters θ_t , and is computed on a minibatch of training sample $\mathcal{D}_{\text{mini}}$. In data parallelism, we divide $\mathcal{D}_{\text{mini}}$ into N smaller batches, denoted by $\{\mathcal{D}^1, \dots, \mathcal{D}^N\}$. Then, we distribute these batches to N workers, each with a corresponding batch. Once the data is distributed, these workers can work at the same time. The gradient of the entire minibatch is obtained by aggregating the gradients computed by the workers, like this

$$\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} = \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^1)}{\partial \theta_t}}_{\text{worker 1}} + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^2)}{\partial \theta_t}}_{\text{worker 2}} + \dots + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^N)}{\partial \theta_t}}_{\text{worker N}} \quad (2.33)$$

In ideal cases where the workers coordinate well and the communication overhead is small, data parallelism can achieve nearly an N -fold speed-up for training.

- **Model Parallelism.** Although data parallelism is simple and effective, it requires each worker to run the entire LLM and perform the complete forward and backward process. As LLMs grow larger, it sometimes becomes unfeasible to load and execute an LLM on a single device. In this case, we can decouple the LLM into smaller components and run these components on different devices. One simple way to do this is to group consecutive layers in the layer stack and assign each group to a worker. The workers operate in the order of the layers in the stack, that is, in the forward pass we process the input from lower-level to upper-level layers, and in the backward pass we propagate the error gradients from upper-level to lower-level layers. Consider, for example, a Transformer decoder with L stacked blocks. To distribute the computation load, each block is assigned to a worker. See the following illustration for a single run of the forward and backward passes of this model.

Worker L	B_L (\uparrow) B_L (\downarrow)
...	...
Worker 2	B_2 (\uparrow) B_2 (\downarrow)
Worker 1	B_1 (\uparrow) B_1 (\downarrow)

Here B_l denotes the computation of block l , and the symbols \uparrow and \downarrow denote the forward and backward passes, respectively. Note that this parallelism method forces the workers to run in sequence, so a worker has to wait for the previous worker to finish their job. This results in the devices being idle for most of the time. In practical systems, model parallelism is generally used together with other parallelism mechanisms to maximize the use of devices.

- **Tensor Parallelism.** Parallelism can also be performed in a single computation step. A common example is splitting a large parameter matrix into chunks, multiplying an input tensor with each of these chunks separately, and then concatenating the results of these multiplications to form the output. For example, consider the multiplication of the representation $\mathbf{h} \in \mathbb{R}^d$ with the parameter matrix $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ in an FFN sub-layer (see Eq. (2.25)). We can slice the matrix $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ vertically to a sequence of M sub-matrices

$$\mathbf{W}_h = \begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & \dots & \mathbf{W}_h^M \end{bmatrix} \quad (2.34)$$

where each sub-matrix \mathbf{W}_h^k has a shape of $d \times \frac{d_h}{M}$. The multiplication of \mathbf{h} with \mathbf{W}_h can be expressed as

$$\begin{aligned} \mathbf{h}\mathbf{W}_h &= \mathbf{h} \begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & \dots & \mathbf{W}_h^M \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{h}\mathbf{W}_h^1 & \mathbf{h}\mathbf{W}_h^2 & \dots & \mathbf{h}\mathbf{W}_h^M \end{bmatrix} \end{aligned} \quad (2.35)$$

We can perform matrix multiplications $\{\mathbf{h}\mathbf{W}_h^1, \mathbf{h}\mathbf{W}_h^2, \dots, \mathbf{h}\mathbf{W}_h^M\}$ on M devices separately. As a result, we distribute a large matrix multiplication across multiple devices, each of which may have relatively small memory. From the perspective of the design of modern GPUs, tensor parallelism over GPUs provides a two-level, tile-based approach to parallel computing. First, at a higher level, we decompose a matrix multiplication into sub-matrix multiplications that can directly fit into the memory of GPUs. Then, at a lower level, we execute these sub-matrix multiplications on GPUs using tile-based parallel algorithms that are specifically optimized for GPUs.

- **Pipeline Parallelism.** Above, in model parallelism, we have described a simple approach to spreading groups of model components across multiple devices. But this method is inefficient because only one device is activated at a time during processing. Pipeline parallelism addresses this issue by introducing overlaps between computations on different devices [Harlap et al., 2018; Huang et al., 2019]. To do this, a batch of samples is divided into a number of micro-batches, and then these micro-batches are processed by each worker as usual. Once a micro-batch is processed by a worker and passed to the next one, the following micro-batch immediately occupies the same worker. In other words, we create a pipeline in which different computation steps can overlap if multiple jobs are given to the pipeline. The following shows an illustration of pipeline parallelism for processing 3 micro-batches.

Worker L				$B_{L,1}$	$B_{L,2}$	$B_{L,3}$	$B_{L,1}$	$B_{L,2}$	$B_{L,3}$
...			...						
Worker 2		$B_{2,1}$	$B_{2,2}$	$B_{2,3}$			$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
Worker 1	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$				$B_{1,1}$	$B_{1,2}$	$B_{1,3}$

Here $B_{l,k}$ represents the processing of the k -th micro-batch by the l -th worker. Ideally we would like to maximize the number of micro-batches, and thus minimize the idle time of the workers. However, in practice, using small micro-batches often reduces GPU utilization and increases task-switching costs. This may, in turn, decrease the overall system throughput.

The ultimate goal of parallel processing is to achieve linear growth in efficiency, that is, the number of samples that can be processed per unit of time increases linearly with the number of devices. However, distributed training is complicated, and influenced by many factors in addition to the parallelism method we choose. One problem, which is often associated with distributed systems, is the cost of communication. We can think of a distributed system as a group of networked nodes. Each of these nodes can perform local computation or pass data to other nodes. If

there are a large number of such nodes, it will be expensive to distribute and collect data across them. Sometimes, the time savings brought about by parallelism are offset by the communication overhead of a large network. Another problem with large-scale distributed systems is that the synchronization of nodes introduces additional costs. As is often the case, some nodes may take longer to work, causing others to wait for the slowest ones. While we can use asynchronous training to handle heterogeneity in computational resources, this may lead to stale gradients and non-guaranteed convergence. Moreover, as more nodes are added to the network, there is more chance to have crashed nodes during training. In this case, we need to ensure that the whole system is fault tolerant. In many practical settings, to increase scalability, one needs to take into account additional issues, including architecture design, data transfer and computation overlap, load balancing, memory bandwidth and so on.

Training LLMs is so computationally expensive that, even though distributed training is already in use, researchers and engineers often still employ various model compression and speed-up methods to improve training efficiency [Weng, 2021]. One example is mixed precision training, in which low precision data (such as FP16 and FP8 data) is used for gradient computation on each individual node, and single or double precision data (such as FP32/FP64 data) is used for updating the model [Micikevicius et al., 2018]. A key operation in this approach is gradient accumulation where gradients need to be accumulated and synchronized across nodes. However, due to the non-associativity of floating-point addition, this can lead to slight numerical differences in accumulated gradients on different nodes, which may affect model convergence and final performance. This problem is more obvious if there are a large number of nodes involved in distributed training, especially given that low-precision numerical computations may encounter overflow and underflow issues, as well as inconsistencies across different hardware devices. Therefore, the design of distributed systems needs to consider these numerical computation issues to ensure satisfactory results and convergence.

2.2.4 Scaling Laws

The success of LLMs reveals that training larger language models using more resources can lead to improved model performance. Researchers have explained this as **scaling laws** of LLMs. More specifically, scaling laws describe the relationships between the performance of LLMs and the attributes of LLM training, such as the model size, the amount of computation used for training, and the amount of training data. For example, Hestness et al. [2017] show that the performance of deep neural networks is a power-law-like function of the training data size. In the beginning, when the amount of training data is not large, the performance of the model improves slowly. Afterward, when more training data is used, the model enters a phase of rapid performance improvement, and the performance curve resembles a power-law curve. Ultimately, the improvement in performance becomes slow again, and more data does not lead to significant gains. Figure 2.3 shows an example of such curves.

In NLP, a traditional view holds that the performance gains will disappear at a certain point as the training is scaled up. However, recent results show that, if we consider the problem on a larger scale, scaling up training is still a very effective method for obtaining stronger LLMs. For example, both closed-source and open-source LLMs can benefit from more data, even though trillions of tokens have already been used for training.

With the increase in the scale of model training, LLMs exhibit new capabilities, known as the

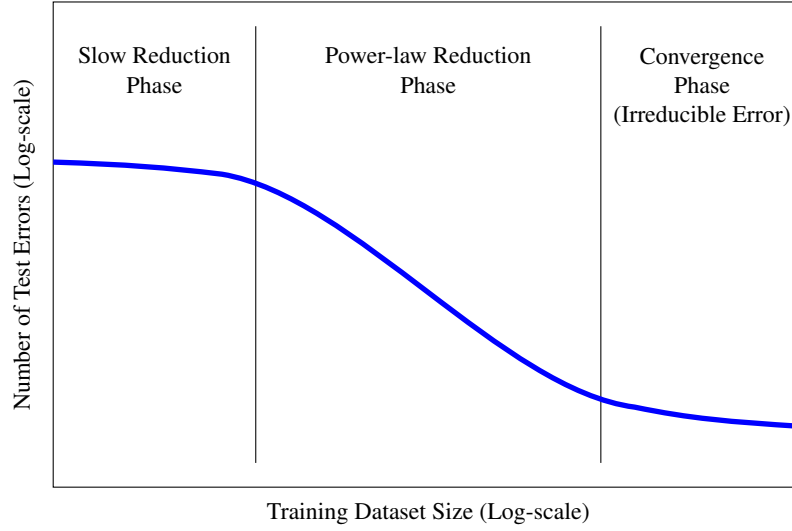


Fig. 2.3: A scaling law of test error against a variable of interest (e.g., training dataset size) [Hestness et al., 2017]. The curve of the scaling law can be divided into three phases. At the beginning, the number of test errors decreases slowly when more training data is used, but this only lasts for a short period. In the second phase, the number of test errors decreases drastically, and the curve becomes a power law curve. After that, the error reduction slows down again in the third phase. Note that there are irreducible errors that cannot be eliminated, regardless of the amount of training data.

emergent abilities of LLMs. For example, Wei et al. [2022b] studied the scaling properties of LLMs across different model sizes and amounts of computational resources. Their work shows that some abilities emerge when we scale the model size to certain level. The appearance of emergent abilities has demonstrated the role of scaled training in enhancing the performance of LLMs, and it has also, to some extent, motivated researchers to continuously attempt to train larger models. As larger and stronger LMs continue to appear, our understanding of the scaling laws continues to mature. This helps researchers predict the performance of LLMs during training and estimate the minimal computational resources required to achieve a given level of performance.

To understand how model performance scales with various factors considered during training, it is common to express the model performance as a function of these factors. For example, in the simplest case, we can express the loss or error of an LLM as a function of a single variable of interest. However, there are no universal scaling laws that can describe this relationship. Instead, different functions are proposed to fit the learning curves of LLMs.

Let x be the variable of interest (such as the number of model parameters) and $\mathcal{L}(x)$ be the loss of the model given x (such as the cross-entropy loss on test data). The simplest form of $\mathcal{L}(x)$ is a power law

$$\mathcal{L}(x) = ax^b \quad (2.36)$$

where a and b are parameters that are estimated empirically. Despite its simplicity, this function has successfully interpreted the scaling ability of language models and machine translation systems in terms of model size (denoted by N) and training dataset size (denoted by D) [Gordon et al., 2021; Hestness et al., 2017]. For example, Kaplan et al. [2020] found that the performance of their language model improves as a power law of either N or D after an initial transient period, and expressed these relationships using $\mathcal{L}(N) = (\frac{N}{8.8 \times 10^{13}})^{-0.076}$ and $\mathcal{L}(D) = (\frac{D}{5.4 \times 10^{13}})^{-0.095}$ (see Figure 2.4).

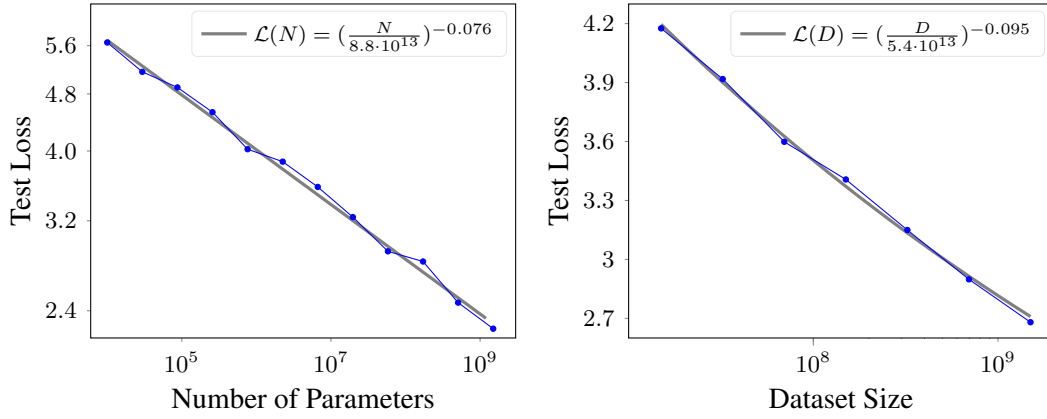


Fig. 2.4: Test loss against model size (N) and training dataset size (D) (data points are plotted for illustrative purposes). We plot test loss as a function of N , which is defined as $\mathcal{L}(N) = \left(\frac{N}{8.8 \times 10^{13}}\right)^{-0.076}$, and a function of D , which is defined as $\mathcal{L}(D) = \left(\frac{D}{5.4 \times 10^{13}}\right)^{-0.095}$ [Kaplan et al., 2020].

An improvement to this scaling law is to add an **irreducible error** term to the power law. The form of $\mathcal{L}(x)$ is then given by

$$\mathcal{L}(x) = ax^b + \epsilon_{\infty} \quad (2.37)$$

where ϵ_{∞} is the irreducible error that accounts for the error due to unknown variables, which is present even as $x \rightarrow \infty$. Eq. (2.37) is one of the most widely used forms for designing scaling laws of LLMs. For example, Rosenfeld et al. [2020] developed a scaling law that involves both model scaling and dataset scaling, like this

$$\mathcal{L}(N, D) = aN^b + cD^d + \epsilon_{\infty} \quad (2.38)$$

An example of such formulation is the Chinchilla scaling law. It states that the test loss per token is the sum of the inverse proportion functions of N and D , with an additional irreducible error term. Hoffmann et al. [2022] express this scaling law as

$$\mathcal{L}(N, D) = \underbrace{\frac{406.4}{N^{0.34}}}_{\text{model scaling}} + \underbrace{\frac{410.7}{D^{0.28}}}_{\text{dataset scaling}} + \underbrace{1.69}_{\text{irreducible error}} \quad (2.39)$$

All the scaling laws mentioned above are based on monotonic functions. So they cannot cover functions with inflection points, such as double descent curves. In response, researchers have explored more sophisticated functions to fit the learning curves. Examples of such functions can be found in Alabdulmohsin et al. [2022] and Caballero et al. [2023]’s work.

The significance of scaling laws lies in providing directional guidance for LLM research: if we are still in the region of the power law curve, using more resources to train larger models is a very promising direction. While this result “forces” big research groups and companies to invest more in computational resources to train larger models, which is very expensive, scaling laws continuously push the boundaries of AI further away. On the other hand, understanding scaling laws helps researchers make decisions in training LLMs. For example, given the computational

resources at hand, the performance of LLMs may be predicted.

One last note on scaling laws in this section. For LLMs, a lower test loss does not always imply better performance on all downstream tasks. To adapt LLMs, there are several steps such as fine-tuning and prompting that may influence the final result. Therefore, the scaling laws for different downstream tasks might be different in practice.

2.3 Long Sequence Modeling

We have already seen that, in large-scale training, larger language models can be developed by using more data and computational resources. However, scaling up can also occur in other directions. For instance, in many applications, LLMs are adapted to process significantly long sequences. An interesting example is that we pre-train an LLM on extensive texts of normal length and then apply it to deal with very long token sequences, far beyond the length encountered in pre-training. Here we use $\Pr(\mathbf{y}|\mathbf{x})$ to denote the text generation probability where \mathbf{x} is the context and \mathbf{y} is the generated text. There are broadly three types of long sequence modeling problems.

- **Text generation based on long context** (i.e., \mathbf{x} is a long sequence). For example, we generate a short summary for a very long text.
- **Long text generation** (i.e., \mathbf{y} is a long sequence). For example, we generate a long story based on a few keywords.
- **Long text generation based on long context** (i.e., both \mathbf{x} and \mathbf{y} are long sequences). For example, we translate a long document from Chinese to English.

Recently, NLP researchers have been more interested in applying and evaluating LLMs on tasks where extremely long input texts are involved. Imagine an LLM, which reads a C++ source file containing tens of thousands of lines, and outlines the functionality of the program corresponding to the source file. Such models, capable of handling extensive textual contexts, are sometimes called **long-context LLMs**. In this section we will restrict ourselves to long-context LLMs, but the methods discussed here can be applicable to other problems.

For Transformers, dealing with long sequences is computationally expensive, as the computational cost of self-attention grows quadratically with the sequence length. This makes it infeasible to train and deploy such models for very long inputs. Two strands of research have tried to adapt Transformers to long-context language modeling.

- The first explores efficient training methods and model architectures to learn self-attention models from long-sequence data.
- The other adapts pre-trained LLMs to handle long sequences with modest or no fine-tuning efforts.

Here, we will discuss the former briefly since it can be found in general discussions of efficient Transformer architectures [Tay et al., 2020; Xiao and Zhu, 2023]. We will focus on the latter,

highlighting popular methods in recent LLMs. We will also discuss the strengths and limitations of these long-sequence models.

2.3.1 Optimization from HPC Perspectives

We begin our discussion by considering improvements to standard Transformer models from the perspectives of high-performance computing. Most of these improvements, though not specifically designed for LLMs, have been widely applied across various deep learning models [Kim et al., 2023]. A commonly used approach is to adopt a low-precision implementation of Transformers. For example, we can use 8-bit or 16-bit fixed-point data types for arithmetic operations, instead of 32-bit or 64-bit floating-point data types. Using these low-precision data types can increase the efficiency and memory throughput, so that longer sequences can be processed more easily. An alternative approach is to improve Transformers by using hardware-aware techniques. For example, on modern GPUs, the efficiency of Transformers can be improved by using IO-aware implementations of the self-attention function [Dao et al., 2022; Kwon et al., 2023].

Another way to handle long sequences is through sequence parallelism [Li et al., 2023b; Korthikanti et al., 2023]. Specifically, consider the general problem of attending the query \mathbf{q}_i at the position i to the keys \mathbf{K} and values \mathbf{V} . We can divide \mathbf{K} by rows and obtain a set of sub-matrices $\{\mathbf{K}^{[1]}, \dots, \mathbf{K}^{[n_u]}\}$, each corresponding to a segment of the sequence. Similarly, we can obtain the sub-matrices of \mathbf{V} , denoted by $\{\mathbf{V}^{[1]}, \dots, \mathbf{V}^{[n_u]}\}$. Then, we assign each pair of $\mathbf{K}^{[u]}$ and $\mathbf{V}^{[u]}$ to a computing node (e.g., a GPU of a GPU cluster). The assigned nodes can run in parallel, thereby parallelizing the attention operation.

Recall that the output of the self-attention model can be written as

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \sum_{j=0}^{m-1} \alpha_{i,j} \mathbf{v}_j \quad (2.40)$$

where $\alpha_{i,j}$ is the attention weight between positions i and j . In Transformers, $\alpha_{i,j}$ is obtained by normalizing the rescaled version of the dot product between \mathbf{q}_i and \mathbf{k}_j . Let $\beta_{i,j}$ denote the attention score between \mathbf{q}_i and \mathbf{k}_j . We have

$$\beta_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} + \text{Mask}(i, j) \quad (2.41)$$

where $\text{Mask}(i, j)$ is the masking variable for (i, j) . Then, we define the attention weight $\alpha_{i,j}$ to be

$$\begin{aligned} \alpha_{i,j} &= \text{Softmax}(\beta_{i,j}) \\ &= \frac{\exp(\beta_{i,j})}{\sum_{j'} \exp(\beta_{i,j'})} \end{aligned} \quad (2.42)$$

On each computing node, we need to implement these equations. Given the keys and values assigned to this node, computing the numerator of the right-hand side of Eq. (2.42) (i.e., $\exp(\beta_{i,j})$) is straightforward, as all the required information is stored on the node. However, computing the denominator of the right-hand side of Eq. (2.42) involves a sum of $\exp(\beta_{i,j'})$ over all j' s, which

requires transferring data to and from other nodes. To illustrate, suppose that \mathbf{v}_j and \mathbf{k}_j are placed on node u . We can rewrite Eq. (2.42) as

$$\alpha_{i,j} = \frac{\overbrace{\exp(\beta_{i,j})}^{\text{node } u}}{\underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[1]}} \exp(\beta_{i,j'})}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[n_u]}} \exp(\beta_{i,j'})}_{\text{node } n_u}} \quad (2.43)$$

where the notation $\mathbf{k}_{j'} \in \mathbf{K}^{[u]}$ represents that $\mathbf{k}_{j'}$ is a row vector of $\mathbf{K}^{[u]}$. In a straightforward implementation, we first perform the summations $\{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})\}$ separately on the corresponding nodes. Then, we collect these summation results from different nodes to combine them into a final result. This corresponds to a collective operation in the context of parallel processing. There are many efficient implementations of such operations, such as the all-reduce algorithms. Hence the sum of all $\exp(\beta_{i,j})$ values can be computed using optimized routines in collective communication toolkits.

Given the attention weights $\{\alpha_{i,j}\}$, we then compute the attention results using Eq. (2.40). The problem can be re-expressed as

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[1]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[n_u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } n_u} \quad (2.44)$$

Like Eq. (2.43), Eq. (2.44) can be implemented as a summation program in parallel processing. First, perform the weighted summations of values on different nodes simultaneously. Then, we collect the results from these nodes via collective operations.

Note that, although this section primarily focuses on long sequence modeling, much of the motivation for sequence parallelism comes from the distributed training methods of deep networks, as discussed in Section 2.2.3. As a result, the implementation of these methods can be based on the same parallel processing library.

2.3.2 Efficient Architectures

One difficulty of applying Transformers to long sequences is that self-attention has a quadratic time complexity with respect to the sequence length. Moreover, a **key-value cache** (or **KV cache** for short) is maintained during inference, and its size increases as more tokens are processed. Although the KV cache grows linearly with the sequence length, for extremely long input sequences, the memory footprint becomes significant and it is even infeasible to deploy LLMs for such tasks. As a result, the model architecture of long-context LLMs generally moves away from the standard Transformer, turning instead to the development of more efficient variants and alternatives.

One approach is to use sparse attention instead of standard self-attention. This family of models is based on the idea that only a small number of tokens are considered important when attending to a given token, and so most of the attention weights between tokens are close to zero.

As a consequence, we can prune most of the attention weights and represent the attention model in a compressed form. To illustrate, consider the self-attention model

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \alpha(\mathbf{Q}, \mathbf{K})\mathbf{V} \quad (2.45)$$

where the attention weight matrix $\alpha(\mathbf{Q}, \mathbf{K}) \in \mathbb{R}^{m \times m}$ is obtained by

$$\begin{aligned} \alpha(\mathbf{Q}, \mathbf{K}) &= \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \text{Mask}\right) \\ &= \begin{bmatrix} \alpha_{0,0} & 0 & 0 & \dots & 0 \\ \alpha_{1,0} & \alpha_{1,1} & 0 & \dots & 0 \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & \dots & \alpha_{m-1,m-1} \end{bmatrix} \end{aligned} \quad (2.46)$$

Each row vector $[\alpha_{i,0} \dots \alpha_{i,i} \ 0 \dots 0]$ corresponds to a distribution of attending the i -th token to every token of the sequence. Since language models predict next tokens only based on their left-context, we normally write the output of the attention model at position i as

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &= \begin{bmatrix} \alpha_{i,0} & \dots & \alpha_{i,i} \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i \end{bmatrix} \\ &= \sum_{j=0}^i \alpha_{i,j} \mathbf{v}_j \end{aligned} \quad (2.47)$$

where $\mathbf{K}_{\leq i} = \begin{bmatrix} \mathbf{k}_0 \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$ and $\mathbf{V}_{\leq i} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$ are the keys and values up to position i .

In the original version of self-attention $[\alpha_{i,0} \dots \alpha_{i,i}]$ is assumed to be dense, that is, most of the values are non-zero. In sparse attention, some of the entries of $[\alpha_{i,0} \dots \alpha_{i,i}]$ are considered non-zero, and the remaining entries are simply ignored in computation. Suppose $G \subseteq \{0, \dots, i\}$ is the set of indices of the non-zero entries. For language models, the output of the sparse attention model at position i is given by

$$\text{Att}_{\text{sparse}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j \in G} \alpha'_{i,j} \mathbf{v}_j \quad (2.48)$$

Here $\{\alpha'_{i,j}\}$ are normalized over G . Hence their values are different from the original attention weights (in fact we have $\alpha'_{i,j} > \alpha_{i,j}$). The sparsity of the model is determined by how large G is. Sparse attention models differ in the way we define G . One simple approach is to define G based on heuristically designed patterns. For example, a widely-used pattern involves having G cover a window of tokens located near position i [Parmar et al., 2018].

While sparse attention reduces the computation through the use of sparse operations, such models still have significant limitations as we must keep the entire KV cache (i.e., $\mathbf{K}_{\leq i}$ and $\mathbf{V}_{\leq i}$)

during inference. If the sequence is very long, storing this cache will become highly memory-intensive. To address this, we can consider a different form of attention models where the KV cache is not explicitly retained. Linear attention is one such approach [Katharopoulos et al., 2020]. It uses a kernel function $\phi(\cdot)$ to project each query and key onto points $\mathbf{q}'_i = \phi(\mathbf{q}_i)$ and $\mathbf{k}'_i = \phi(\mathbf{k}_i)$, respectively. By removing the Softmax function under such transformations¹⁰, the form of the resulting attention model is given by

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &\approx \text{Att}_{\text{linear}}(\mathbf{q}'_i, \mathbf{K}'_{\leq i}, \mathbf{V}_{\leq i}) \\ &= \frac{\mathbf{q}'_i \mu_i}{\mathbf{q}'_i \nu_i} \end{aligned} \quad (2.49)$$

where μ_i and ν_i are variables that are computed in the recurrent forms

$$\mu_i = \mu_{i-1} + \mathbf{k}'_i{}^T \mathbf{v}_i \quad (2.50)$$

$$\nu_i = \nu_{i-1} + \mathbf{k}'_i{}^T \mathbf{k}'_i \quad (2.51)$$

μ_i and ν_i can be seen as representations of the history up to position i . A benefit of this model is that we need not keep all past queries and values. Instead only the latest representations μ_i and ν_i are used. So the computational cost of each step is a constant, and the model can be easily extended to deal with long sequences.

In fact, this sequential approach to long sequence modeling arises naturally when we adopt a viewpoint of recurrent models. Such models read one token (or a small number of tokens) at a time, update the recurrent state using these inputs, and then discard them before the next token arrives. The output at each step is generated based only on the recurrent state, rather than on all the previous states. The memory footprint is determined by the recurrent state which has a fixed size. Recurrent models can be used in real-time learning scenarios where data arrives in a stream and predictions can be made at any time step. In NLP, applying recurrent models to language modeling is one of the earliest successful attempts to learn representations of sequences. Although Transformer has been used as the foundational architecture in LLMs, recurrent models are still powerful models, especially for developing efficient LLMs. More recently, recurrent models have started their resurgence in language modeling and have been reconsidered as a promising alternative to Transformers [Gu and Dao, 2023]. Figure 2.5 shows a comparison of the models discussed in this subsection.

2.3.3 Cache and Memory

LLMs based on the standard Transformer architecture are global models. The inference for these models involves storing the entire left-context in order to make predictions for future tokens. This requires a KV cache where the representations (i.e., keys and values) of all previously-generated

¹⁰In the new space after this transformation, the Softmax normalization can be transformed into the simple scaling normalization.

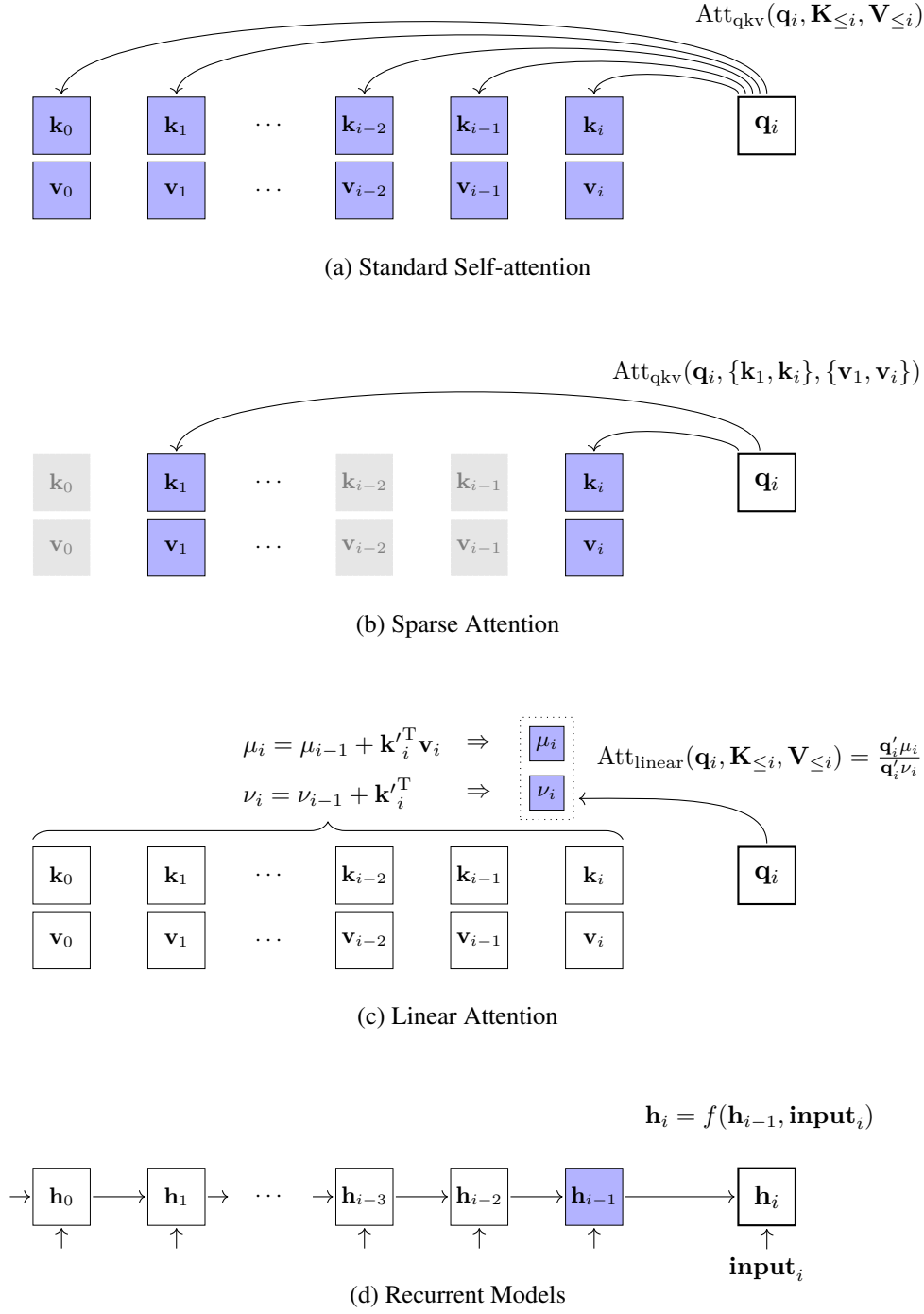


Fig. 2.5: Illustrations of self-attention, sparse attention, linear attention and recurrent models. Blue boxes = cached states for producing the output at position i . $f(\cdot)$ = a recurrent cell.

tokens are kept, and the cost of caching grows as the inference proceeds. Above, we have discussed methods for optimizing this cache via efficient attention approaches, such as sparse attention and linear attention. Another idea, which may have overlap with the previous discussion, is to explicitly encode the context via an additional memory model.

2.3.3.1 Fixed-size KV Cache

A straightforward approach is to represent the keys and values using a fixed-size memory model. Suppose we have a memory Mem which retains the contextual information. We can write the attention operation at position i in a general form

$$\text{Att}(\mathbf{q}_i, \text{Mem}) = \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \quad (2.52)$$

In this model, Mem is simply the KV cache, i.e., $\text{Mem} = (\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$. Thus the size of Mem is determined by i . If we define Mem as a fixed-size variable, then the cost of performing $\text{Att}(\mathbf{q}_i, \text{Mem})$ will be fixed. There are several alternative ways to design Mem.

- One of the simplest methods is to consider a fixed-size window of previous keys and values. Mem is therefore given by

$$\text{Mem} = (\mathbf{K}_{[i-n_c+1, i]}, \mathbf{V}_{[i-n_c+1, i]}) \quad (2.53)$$

where n_c denotes the size of the window. The notation $\mathbf{K}_{[i-n_c+1, i]}$ and $\mathbf{V}_{[i-n_c+1, i]}$ denote the keys and values over positions from $i - n_c + 1$ to i .¹¹ This model can be seen as a type of local attention model.

- It is also possible to define Mem as a pair of summary vectors, which leads to a more compressed representation of the history. A simple way to summarize the previous keys and values is to use the moving average of them. For example, Mem can be defined as the unweighted moving average of the previous n_c keys and values

$$\text{Mem} = \left(\frac{\sum_{j=i-n_c+1}^i \mathbf{k}_j}{n_c}, \frac{\sum_{j=i-n_c+1}^i \mathbf{v}_j}{n_c} \right) \quad (2.54)$$

Alternatively, we can use a weighted version of moving average

$$\text{Mem} = \left(\frac{\sum_{j=i-n_c+1}^i \beta_{j-i+n_c} \mathbf{k}_j}{\sum_{j=1}^{n_c} \beta_j}, \frac{\sum_{j=i-n_c+1}^i \beta_{j-i+n_c} \mathbf{v}_j}{\sum_{j=1}^{n_c} \beta_j} \right) \quad (2.55)$$

Here $\{\beta_1, \dots, \beta_{n_c}\}$ are the coefficients, which can be either learned as model parameters or determined via heuristics. For example, they can be set to increasing coefficients (i.e., $\beta_1 < \beta_2 < \dots < \beta_{n_c-1} < \beta_{n_c}$) in order to give larger weight to positions that are closer to i . We can extend the moving average to include all the positions up to i . This leads to the cumulative average of the keys and values, given in the form

$$\text{Mem} = \left(\frac{\sum_{j=0}^i \mathbf{k}_j}{i+1}, \frac{\sum_{j=0}^i \mathbf{v}_j}{i+1} \right) \quad (2.56)$$

¹¹More formally, we write $\mathbf{K}_{[i-n_c+1, i]} = \begin{bmatrix} \mathbf{k}_{i-n_c+1} \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$ and $\mathbf{V}_{[i-n_c+1, i]} = \begin{bmatrix} \mathbf{v}_{i-n_c+1} \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$. Sometimes we denote $\mathbf{K}_{[i-n_c+1, i]}$ by $\{\mathbf{k}_{i-n_c+1}, \dots, \mathbf{k}_i\}$ and $\mathbf{V}_{[i-n_c+1, i]}$ by $\{\mathbf{v}_{i-n_c+1}, \dots, \mathbf{v}_i\}$ for notation simplicity.

In general, the cumulative average can be written using a recursive formula

$$\text{Mem}_i = \frac{(\mathbf{k}_i, \mathbf{v}_i) + i \cdot \text{Mem}_{i-1}}{i + 1} \quad (2.57)$$

where Mem_i and Mem_{i-1} denote the cumulative averages of the current and previous positions, respectively. An advantage of this model is that we only need to store a single key-value pair during inference, rather than storing all the key-value pairs. Note that the above memory models are related to recurrent models, and more advanced techniques have been used to develop alternatives to self-attention mechanisms in Transformers [Ma et al., 2023].

- The memory Mem can also be a neural network. At each step, it takes both the previous output of the memory and the current states of the model as input, and produces the new output of the memory. This neural network can be formulated as the function

$$\text{Mem} = \text{Update}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \quad (2.58)$$

Here Mem and Mem_{pre} represent the outputs of the memory at the current step and the previous step, respectively. S_{kv} is a set of key-value pairs, representing the recent states of the model. This formulation is general and allows us to develop various memory models by selecting different $\text{Update}(\cdot)$ and S_{kv} configurations. For example, if S_{kv} only contains the latest key-value pair $(\mathbf{k}_i, \mathbf{v}_i)$ and $\text{Update}(\cdot)$ is defined as a recurrent cell, then Eq. (2.58) can be expressed as an RNN-like model

$$\text{Mem} = f((\mathbf{k}_i, \mathbf{v}_i), \text{Mem}_{\text{pre}}) \quad (2.59)$$

where $f(\cdot)$ is a recurrent cell. Recurrence can also be applied to segment-level modeling for efficiency consideration. A simple approach is that we can divide the sequence into segments, and treat S_{kv} as a segment. Applying recurrent models to $\text{Update}(\cdot)$ will result in memory models that operate on segments. A special example is that we define $\text{Update}(\cdot)$ as an FIFO function that adds S_{kv} into the memory and removes the oldest key-value segment from the memory, given by

$$\text{Mem} = \text{FIFO}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \quad (2.60)$$

Consider a memory which includes two segments, one for current segment, and one for the previous segment. In the attention operation, each position can access the history key-value pairs in two closest consecutive segments. This essentially defines a local memory, but it and its variants have been widely used segment-level recurrent models [Dai et al., 2019; Hutchins et al., 2022; Bulatov et al., 2022].

- The above memory models can be extended to involve multiple memories. An example of this approach is compressive Transformer [Rae et al., 2019]. It employs two distinct fixed-size memories: one for modeling local context (denoted by Mem), and the other for modeling and compressing long-term history (denoted by CMem). The KV cache in this model is the combination of Mem and CMem . The attention function can be written as

$$\text{Att}_{\text{com}}(\mathbf{q}_i, \text{Mem}, \text{CMem}) = \text{Att}_{\text{qkv}}(\mathbf{q}_i, [\text{Mem}, \text{CMem}]) \quad (2.61)$$

where $[\text{Mem}, \text{CMem}]$ is a combined memory of Mem and CMem . As with other segment-level models, the compressive Transformer model operates on segments of the sequence. Each segment is a sequence of n_s consecutive tokens, and we denote S_{kv}^k as the key-value pairs corresponding to the tokens of the k -th segment. When a new segment arrives, Mem is updated in an FIFO fashion: we append the n_c key-value pairs in S_{kv}^k to Mem , and then pop the n_s oldest key-value pairs from Mem , which is given by

$$\text{Mem} = \text{FIFO}(S_{\text{kv}}^k, \text{Mem}_{\text{pre}}) \quad (2.62)$$

The popped key-value pairs are then used to update the compressive memory CMem . These n_s key-value pairs are compressed into $\frac{n_s}{c}$ key-value pairs via a compression network. CMem is an FIFO which appends the compressed $\frac{n_s}{c}$ key-value pairs to the tail of the queue, and drops the first $\frac{n_s}{c}$ key-value pairs of the queue. It is given by

$$\text{CMem} = \text{FIFO}(C_{\text{kv}}^k, \text{CMem}_{\text{pre}}) \quad (2.63)$$

where C_{kv}^k represents the set of compressed key-value pairs. Implicit in the compressive Transformer model is that local context should be represented explicitly with minimal information loss, while long-range context can be more compressed.

- We have already seen that both global and local contexts are useful and can be modeled using attention models. This view motivates the extension to attention models for combining both local and long-term memories [Ainslie et al., 2020; Zaheer et al., 2020; Gupta and Berant, 2020]. A simple but widely-used approach is to involve the first few tokens of the sequence in attention, serving as global tokens. This approach is usually applied along with other sparse attention models. An advantage of incorporating global tokens of the sequence is that it helps smooth the output distribution of the Softmax function used in attention weight computation, and thus stabilizes model performance when the context size is very large [Xiao et al., 2024]. One drawback, however, is that using a fixed-size global memory may result in information loss. When dealing with long sequences, we need to enlarge the KV cache for sufficient representations of the context, but this in turn increases the computational cost.

Figure 2.6 shows illustrations of the above approaches. Note that, while we focus on optimization of the KV cache here, this issue is closely related to those discussed in the previous section. All of the methods we have mentioned so far can broadly be categorized as efficient attention approaches, which are widely used in various Transformer variants.

2.3.3.2 Memory-based Models

The modeling of memories discussed above was based on updates to the KV cache, and the resulting models are typically referred to as **internal memories**. We now consider another family of models, called **external memories**, which operate as independent models to access large-scale contexts for LLMs. Many such models are based on **memory-based methods** which have been extensively discussed in machine learning [Bishop, 2006]. A common example is nearest neighbor algorithms: we store context representations in a datastore, and try to find the most similar

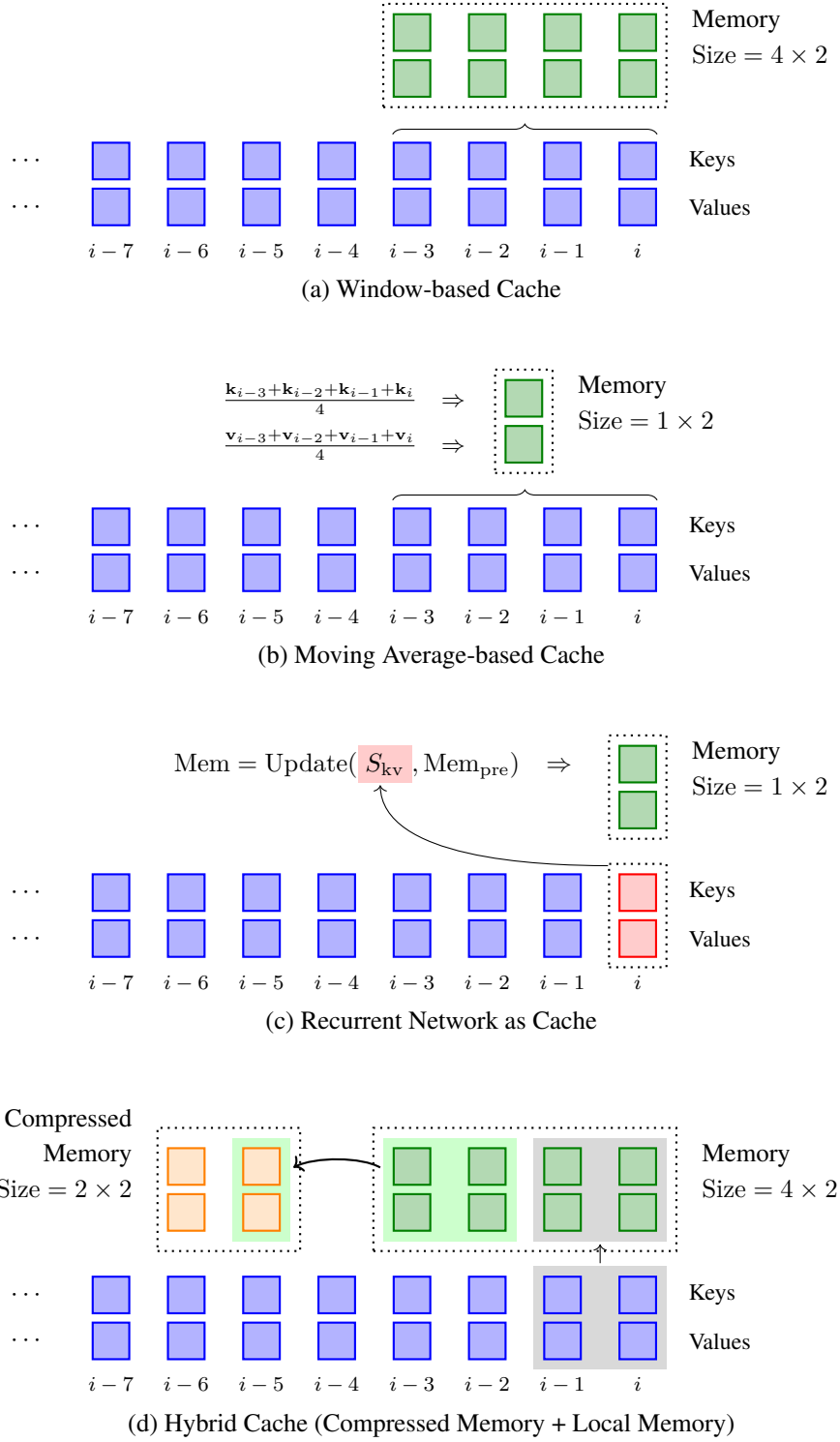


Fig. 2.6: Illustrations of fixed-size KV caches in LLMs. Blue boxes represent the keys and values generated during LLM inference, green boxes represent the keys and values stored or encoded in the primary memory, and orange boxes represent the keys and values stored or encoded in the compressed memory.

stored representations to match a given query. The retrieved context representations are then used to improve attention for this query.

Here, we consider the *k*-nearest neighbors (*k*-NN) method which is one of the most popular memory-based methods. Since our focus is language modeling in this section, we define a sample in the datastore as a key-value pair corresponding to some context state. Note that “context” is a broad concept here, not just a sequence prefix in text generation. One might, for example, view the entire dataset as the context for predicting tokens. This allows us to retrieve the closest context situation in a set of sequences, rather than a given sequence prefix. Although we will restrict ourselves to context modeling for a single sequence, in this subsection, we discuss a relatively more general case.

Suppose we have a set of keys $\{\mathbf{k}_j\}$ with corresponding values $\{\mathbf{v}_j\}$, and suppose we store these key-value pairs in a vector database¹². For each query \mathbf{q}_i , we find its *k* nearest neighbours by growing the radius of the sphere centered as \mathbf{q}_i until it contains *k* data points in $\{\mathbf{k}_j\}$. This results in a set of *k* keys along with their corresponding values, denoted by Mem_{knn} . As before, we denote Mem as the local memory for the query, such as the KV cache of neighboring tokens. Our goal is to attend query \mathbf{q}_i to both the local memory Mem and the long-term memory Mem_{knn} . There are, of course, several ways to incorporate Mem and Mem_{knn} into the attention model. For example, we might simply combine them to form a single KV cache $[\text{Mem}, \text{Mem}_{knn}]$, and attend \mathbf{q}_i to $[\text{Mem}, \text{Mem}_{knn}]$ via standard QKV attention. Or we might use Mem and Mem_{knn} in separate attention steps. An example of such approaches is the model developed by Wu et al. [2021]. It linearly combines the two types of attention, given by

$$\text{Att}(\mathbf{q}_i, \text{Mem}, \text{Mem}_{knn}) = \mathbf{g} \odot \text{Att}_{\text{local}} + (1 - \mathbf{g}) \odot \text{Att}_{knn} \quad (2.64)$$

$$\text{Att}_{\text{local}} = \text{Att}(\mathbf{q}_i, \text{Mem}) \quad (2.65)$$

$$\text{Att}_{knn} = \text{Att}(\mathbf{q}_i, \text{Mem}_{knn}) \quad (2.66)$$

Here $\mathbf{g} \in \mathbb{R}^d$ is the coefficient vector, which can be the output of a learned gate.

Given the *k*-NN-based memory model described above, the remaining task is to determine which key-value pairs are retained in the datastore. For standard language modeling tasks, we consider the previously seen tokens in a sequence as the context, so we can add the keys and values of all these tokens into the datastore. In this case, the resulting *k*-NN-based attention model is essentially equivalent to a sparse attention model [Gupta et al., 2021].

Alternatively, we can extend the context from one sequence to a collection of sequences. For example, we might collect all key-value pairs across the sequences in a training dataset and add them to the datastore to model a larger context. Thus, LLMs can predict tokens based on a generalized context. A problem with this approach is that the computational cost would be large if many sequences are involved. Since these sequences are part of our training data, we can build and optimize an index for the vectors in the datastore before running the LLMs. As a result, the retrieval of similar vectors can be very efficient, as in most vector databases.

In fact, all the above-mentioned methods can be viewed as instances of a retrieval-based approach. Instead of using retrieval results to improve attention, we can apply this approach in other ways as well. One application of *k*-NN-based search is *k*-NN language modeling (or *k*-NN LM) [Khandelwal et al., 2020]. The idea is that, although it is attempting to extend the context used in self-attention by incorporating nearest neighbors in representation learning, in practice, similar hidden states in Transformers are often highly predictive of similar tokens in subsequent positions.

¹²A vector database, or vector store, is a database that provides highly optimized retrieval interfaces for finding stored vectors that closely match a query vector.

In k -NN LM, each item in the datastore is a key-value tuple (\mathbf{z}, w) , where \mathbf{z} represents a hidden state of the LLM at a position, and w represents the corresponding prediction. A typical way to create the datastore is to collect the output vector of the Transformer layer stack and the corresponding next token for each position of each sequence in a training dataset. During inference, we have a representation \mathbf{h}_i given a prefix. Given this representation, we first search the datastore for k closest matching data items $\{(\mathbf{z}_1, w_1), \dots, (\mathbf{z}_k, w_k)\}$. Here $\{w_1, \dots, w_k\}$ are thought of as reference tokens for prediction, and thus can be used to guide the token prediction based on \mathbf{h}_i . One common way to make use of reference tokens is to define a distribution over the vocabulary V ,

$$\Pr_{knn}(\cdot|\mathbf{h}_i) = \text{Softmax}([-d_0 \quad \dots \quad -d_{|V|}]) \quad (2.67)$$

where d_v equals the distance between \mathbf{h}_i and \mathbf{z}_j if w_j equals the v -th entry of V , and equals 0 otherwise. We use a linear function with a coefficient λ that interpolates between the retrieval-based distribution $\Pr_{knn}(\cdot|\mathbf{h}_i)$ and the LLM output distribution $\Pr_{lm}(\cdot|\mathbf{h}_i)$

$$\Pr(\cdot|\mathbf{h}_i) = \lambda \cdot \Pr_{knn}(\cdot|\mathbf{h}_i) + (1 - \lambda) \cdot \Pr_{lm}(\cdot|\mathbf{h}_i) \quad (2.68)$$

Then, as usual, we can choose the next token y by maximizing the probability $\Pr(y|\mathbf{h}_i)$.

As with information retrieval (IR) systems, the datastore can also manage texts and provide access to relevant texts for a query. For example, we can store a collection of text documents in a search engine with full-text indexing, and then search it for documents that match a given text-based query. Applying IR techniques to LLMs leads to a general framework called **retrieval-augmented generation (RAG)**. The RAG framework works as follows. We use the context \mathbf{x} as the query and find the k most relevant document pieces $\{\mathbf{c}_1, \dots, \mathbf{c}_k\}$ from the datastore via efficient IR techniques¹³. These search results are combined with the original context via a prompting template $g(\cdot)$ ¹⁴, resulting in an augmented input for the LLM

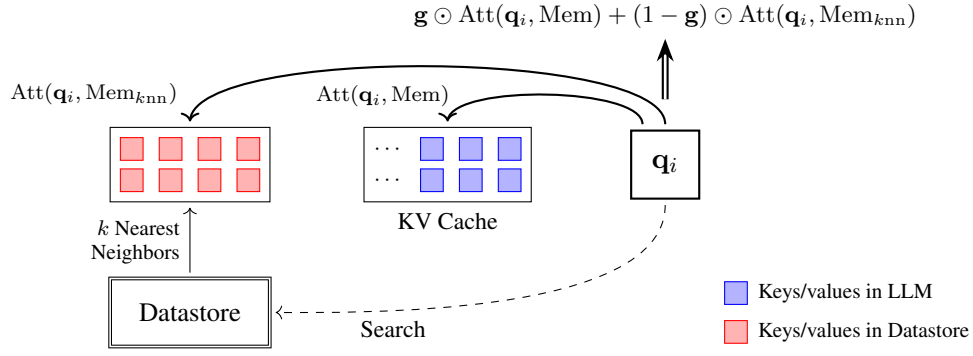
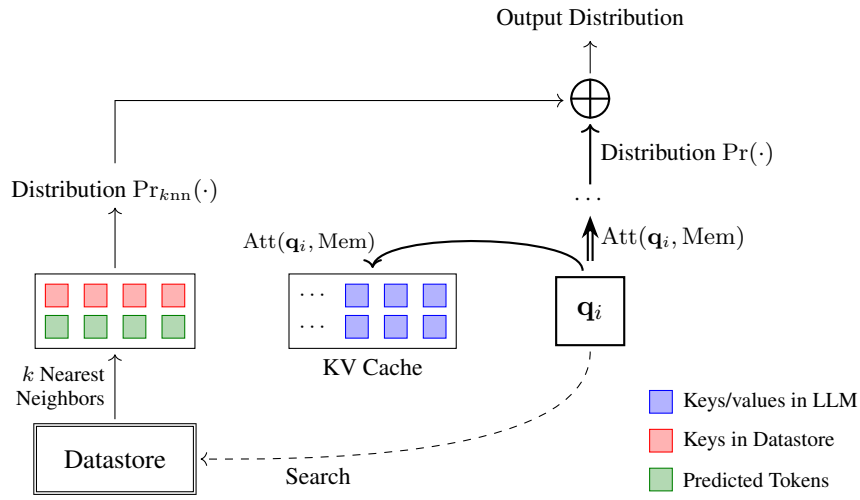
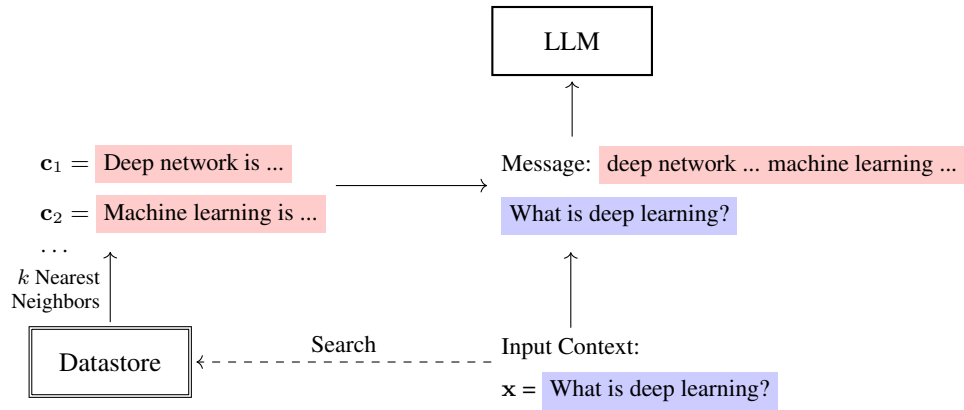
$$\mathbf{x}' = g(\mathbf{c}_1, \dots, \mathbf{c}_k, \mathbf{x}) \quad (2.69)$$

Then, we use \mathbf{x}' as the context and predict the following text using the model $\Pr(y|\mathbf{x}')$. One advantage of RAG is that we need not modify the architecture of LLMs, but instead augment the input to LLMs via an additional IR system. Figure 2.7 shows a comparison of the use of different external memories in LLMs.

¹³In piratical applications, queries are typically generated using a query generation system, which may expand it with variations of tokens and query intent.

¹⁴For example, the template could be:

```
message = { *c1 * } ... { *ck * }
input: { *x * }
output: ____
```

(a) k -NN Search Augmented Attention(b) k -NN Language Modeling

(c) Retrieval-augmented Generation

Fig. 2.7: Illustrations of external memories (or datastores) for language modeling.

2.3.3.3 Memory Capacity

A memory model in LLMs, in the form of a simple key-value cache or a datastore, can broadly be seen as an encoder of contextual information. Ideally, before we say that a memory model is representative of the entire context in token prediction, we need to make sure that the model can accurately represent any part of the context. The standard KV cache is one such model that completely stores all past history. In this case, the model is said to have adequate capacity for memorizing the context. In many practical applications, however, complete memorization is not required. Instead, the goal is to enable LLMs to access important contextual information. As a result, efficient and compressed memory models are developed, as described in this section. Note that, the longer the sequence, the more difficult it becomes for a low-capacity memory model to capture important contextual information. It is therefore common practice to simply increase the model capacity when processing long contexts.

While high-capacity models are generally favorable, they are difficult to train and deploy. A challenging scenario is that the tokens arrive in a stream and the context continuously grows. Developing LLMs for such tasks is difficult as we need to train Transformers on extremely long sequences. A possible way to address this difficulty is to use non-parametric methods, such as retrieval-based methods. For example, as discussed above, we can use a vector database to store previously generated key-value pairs, and thus represent the context by this external memory model. Although this approach side-steps the challenge of representing long context in Transformers, building and updating external memory models are computationally expensive. These models are more often used in problems where the context is given in advance and fixed during inference, and hence unsuitable for streaming context modeling.

In cases where the size of the context continuously grows, applying fixed-size memory models is a commonly used approach. For example, in recurrent models, a sequence of arbitrary length can be summarized into a set of hidden states by which we have a fixed computational cost per step. While recurrent models were initially found to be not very good at handling long-distance dependencies in sequence modeling in early applications of deep learning to NLP, recent advancements have shown that their variants are now effective in modeling extremely long sequences. [Bulatov et al., 2022; Hutchins et al., 2022; Munkhdalai et al., 2024; Ma et al., 2024].

There is no general definition of memory capacity in LLMs. A simple approach might consider how much storage is used to retain contextual information. For example, memory capacity could be defined by the size of the KV cache in Transformers or the vector database used in retrieval-based methods. A related concept is model complexity. In machine learning, there are several ways to define the model complexity of a model. One of the simplest methods is by counting the number of parameters. However, it should be emphasized that the memory models discussed here primarily serve to store information, rather than add trainable parameters. Therefore, a model with a large memory capacity is not necessarily more complex. Nevertheless, in practice determining the capacity of a memory model is not straightforward. In general, we need to control the trade-off between maximizing the performance and controlling the memory footprint.

2.3.4 Sharing across Heads and Layers

In Transformers, the KV cache is a data structure that can be dynamically adjusted along multiple dimensions, such as heads, layers, and sequence length. For example, consider an LLM with L layers. Each layer has τ attention heads, and each head produces a d_h -dimensional output. During

inference, we store the keys and values for up to m tokens. The space complexity of this caching mechanism is $O(L \cdot \tau \cdot d_h \cdot m)$. As we have seen previously, this complexity can be reduced by caching the keys and values for fewer tokens. For example, in sliding window attention, a fixed-size window is used to cache the keys and values in local context. And this model has a space complexity of $O(L \cdot \tau \cdot d_h \cdot m_w)$, with m_w being the size of the window.

In addition to reducing m , we can also decrease the size of the KV cache along other dimensions. A widely-used approach is to enable sharing across heads in multi-head self-attention. Recall from Section 2.1.1 that multi-head self-attention uses multiple sets of queries, keys, and values (each set is called a head), each performing the QKV attention mechanism as usual. This can be expressed as

$$\text{Output} = \text{Merge}(\text{head}_1, \dots, \text{head}_\tau) \mathbf{W}^{\text{head}} \quad (2.70)$$

where $\text{head}_j \in \mathbb{R}^{d_h}$ is computed using the standard QKV attention function

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[j]}, \mathbf{V}_{\leq i}^{[j]}) \quad (2.71)$$

Here, $\mathbf{q}_i^{[j]}$, $\mathbf{K}_{\leq i}^{[j]}$, and $\mathbf{V}_{\leq i}^{[j]}$ are the query, keys, and values that are projected onto the j -th feature sub-space. So this model can be interpreted as performing attention on a group of feature sub-spaces in parallel (see Figure 2.8 (b)). The KV cache needs to retain the keys and values for all these heads, that is, $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), \dots, (\mathbf{K}_{\leq i}^{[\tau]}, \mathbf{V}_{\leq i}^{[\tau]})\}$.

One refinement to the multi-head attention model, called **multi-query attention (MQA)**, is to share keys and values across heads, while allowing queries to be unique for each head [Shazeer, 2019]. In MQA, there is a single set of keys and values $(\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$. In addition, there are τ queries $\{\mathbf{q}_i^{[1]}, \dots, \mathbf{q}_i^{[\tau]}\}$, each corresponding to a different head. For each head, we have

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \quad (2.72)$$

Figure 2.8 (c) illustrates this model. By sharing keys and values, the size of the KV cache would be $O(L \cdot d_h \cdot m)$.

Grouped query attention (GQA) is a natural extension to multi-head attention and MQA [Ainslie et al., 2023]. In GQA, heads are divided into n_g groups, each corresponding to a shared set of keys and values. Hence we have n_g sets of keys and values $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), \dots, (\mathbf{K}_{\leq i}^{[n_g]}, \mathbf{V}_{\leq i}^{[n_g]})\}$. See Figure 2.8 (d) for an illustration. Let $g(j)$ be the group id for the j -th head. The GQA model can be expressed as

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[g(j)]}, \mathbf{V}_{\leq i}^{[g(j)]}) \quad (2.73)$$

The size of the KV cache of GQA is $O(L \cdot n_g \cdot d_h \cdot m)$. One benefit of GQA is that we can trade-off between computational efficiency and model expressiveness by adjusting n_g . When $n_g = \tau$, the model becomes the standard multi-head attention model. By contrast, when $n_g = 1$, it becomes the MQA model.

Sharing can also be performed across layers. Such a method falls into the family of shared weight and shared activation methods, which have been extensively used in Transformers [Dehghani et al., 2018; Lan et al., 2020]. For example, one can share KV activations or attention

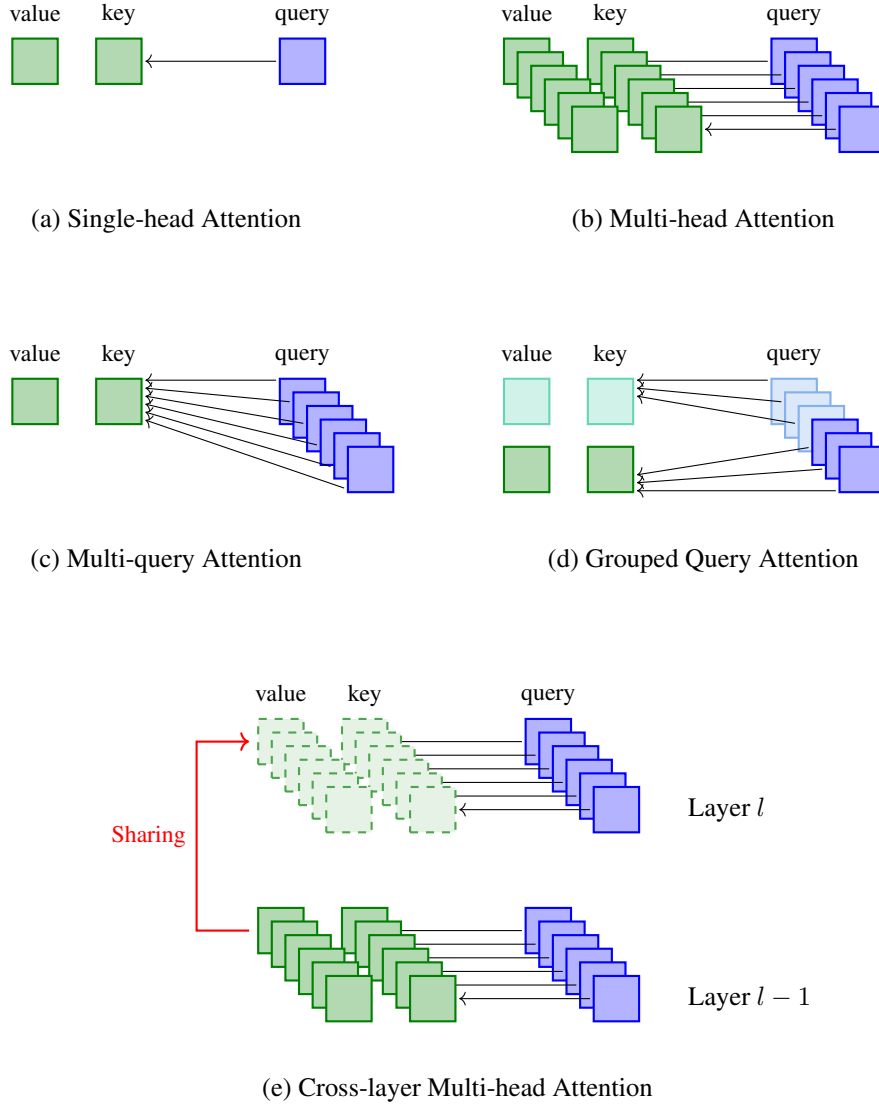


Fig. 2.8: Illustration of QKV attention based on different multi-head and sharing mechanisms. (a) = single-head attention, and (b-e) = attention with multiple heads.

weights across layers to reduce both computation and memory footprints [Xiao et al., 2019; Brandon et al., 2024]. Figure 2.8 (e) shows an illustration of this method, where a query in a layer directly accesses the KV cache of a lower-level layer.

2.3.5 Position Extrapolation and Interpolation

Since Transformer layers are order-insensitive to input, we need some way to encode positional information in the input tokens. To do this, it is common to add positional embeddings to token embeddings, and then feed these combined embeddings into the Transformer layer stack as input. In this case, the embedding at position i can be expressed as

$$\mathbf{e}_i = \mathbf{x}_i + \text{PE}(i) \quad (2.74)$$

where $\mathbf{x}_i \in \mathbb{R}^d$ denotes the token embedding, and $\text{PE}(i) \in \mathbb{R}^d$ denotes the positional embedding. In general, the token embedding \mathbf{x}_i is a position-independent vector, and so the positional embedding $\text{PE}(i)$ is used to encode the positional context. A straightforward approach is to treat $\text{PE}(i)$ as a learnable variable and train it alongside other model parameters. In this way, we can learn a unique representation for each position, and thus distinguish the tokens appearing at different positions of a sequence.

Representations of positions using learned vectors can work well in tasks where the sequences at training and test times are of similar lengths. In practice, however, we often impose length restrictions on sequences during training to prevent excessive computational costs, but wish to apply the trained models to much longer sequences during inference. In this case, using learned positional embeddings has obvious drawbacks, as there are no trained embeddings for positions that are not observed in the training phase.

An alternative approach to modeling positional information is to develop positional embeddings that can generalize: once trained, the embedding model can be used to handle longer sequences. Suppose that we train a positional embedding model on sequences with a maximum length of m_l , and we wish to apply the trained model to a sequence of length m ($m \gg m_l$). If the embedding model is limited in the range of positions that we can observe from training data, then this model will simply fail to deal with new data outside that range. See Figure 2.9 (a) for an illustration where the learned embedding model cannot model data points outside the training domain if it lacks the ability to extrapolate.

There are several approaches to making positional embedding models generalize. They can be grouped into two classes.

- **Extrapolation.** The model learned on observed data points (i.e., positions) can be directly employed to assign meaningful values to data points beyond the original range. For example, suppose we have a series of numbers 1, 2, ..., 10, and we want to understand the meaning of a new number, 15. Knowing that these numbers are natural numbers used for ordering, we can easily infer that 15 is a number that follows 10, even though 15 has not been observed before. Figure 2.9 (b) shows an example of this approach, where a function is learned to fit the data points within a specific range and then applied to estimate the values of data points outside that range.
- **Interpolation.** This approach maps a larger range of data points into the original observation range. For example, suppose we have a model designed for numbers in the range $[1, 10]$. When given a new range of $[1, 20]$, we can scale this down by dividing every number by 2, thereby fitting all numbers into $[1, 10]$. This scaling allows us to use the model trained on the range $[1, 10]$ to describe data points in the expanded range of $[1, 20]$. See Figure 2.9 (c) for an illustration of this approach.

In fact, positional embeddings in many systems have achieved some level of generalization. For example, sinusoidal encoding, the most common positional embedding method, employs sine and cosine functions that can naturally extend to sequences of any length. Although this approach might seem direct and simple, it does not perform well when we significantly extend the sequences for processing. In this subsection, we will discuss several alternative methods based on either extrapolation or interpolation.

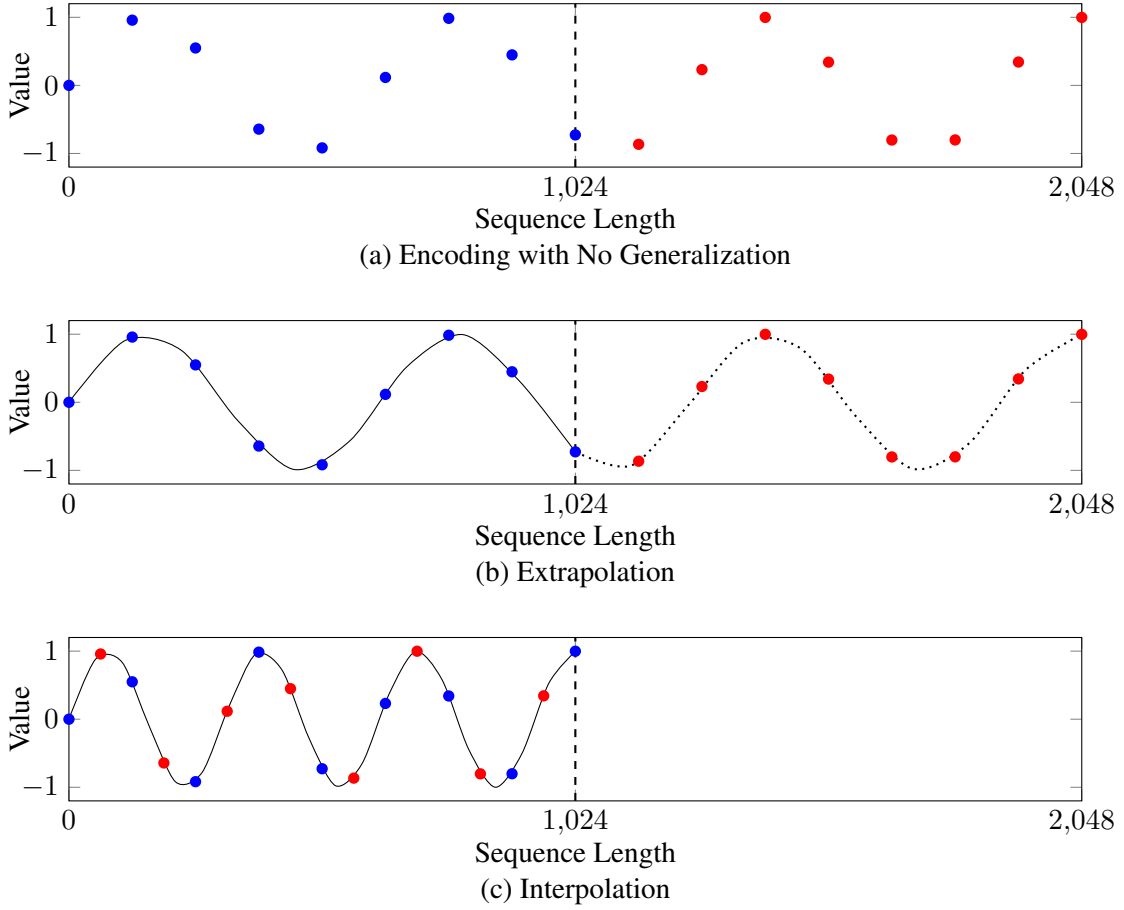


Fig. 2.9: Illustrations of different positional embedding methods for a range of positions. Blue points represent the positions that have been observed during training, and red points represent the positions that are newly observed at test time. In sub-figure (a), the encoding model only memorizes the points seen during training, and cannot generalize. In sub-figures (b) and (c), the model can generalize through extrapolation and interpolation.

2.3.5.1 Attention with Learnable Biases

One problem with Eq. (2.74) is that the embedding model treats each token independently and therefore ignores the distance between different tokens. A common improvement to this model, called relative positional embedding, is to consider the pairwise relationship between tokens [Shaw et al., 2018]. The general idea behind this is to obtain the offset between any pair of positions and incorporate it into the self-attention model. One of the simplest forms of self-attention with relative positional embedding is given by

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j=0}^i \alpha(i, j) \mathbf{v}_j \quad (2.75)$$

$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + \text{PE}(i, j)}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (2.76)$$

The only difference between this model and the original self-attention model is that a bias term $\text{PE}(i, j)$ is added to the query-key product in this new model. Intuitively, $\text{PE}(i, j)$ can be interpreted as a distance penalty for the pair of positions i and j . As i moves away from j , the value of

$\text{PE}(i, j)$ decreases.

$\text{PE}(i, j)$ can be defined in several different ways. Here, we consider the T5 version of relative positional embedding, called the T5 bias [Raffel et al., 2020]. For each pair of query \mathbf{q}_i and key \mathbf{k}_j , the offset between them is defined to be¹⁵

$$d(i, j) = i - j \quad (2.77)$$

A simple design for the bias $\text{PE}(i, j)$ is to share the same learnable variable for all query-key pairs with the same offset, i.e., $\text{PE}(i, j) = u_{i-j}$, where u_{i-j} is the variable corresponding to the offset $i - j$. However, simply assigning a unique value to each offset will restrict this model to observed offsets. When $i - j$ is larger than the maximum trained offset, the model cannot generalize.

The T5 bias instead adopts a generalization of this model. Rather than assigning each query-key offset a unique bias term, it groups difference offsets into “buckets”, each corresponding to one learnable parameter. More specifically, the bias terms for $n_b + 1$ buckets are given as follows.

- For buckets 0 to $\frac{n_b+1}{2} - 1$, each bucket corresponds to one offset, that is, bucket 0 \leftrightarrow offset 0, bucket 1 \leftrightarrow offset 1, bucket 2 \leftrightarrow offset 2, and so on. We express this as $b(i - j) = i - j$.
- For buckets $\frac{n_b+1}{2}$ to n_b , the size of each bucket increases logarithmically. For example, the bucket number for a given offset $i - j \geq \frac{n_b+1}{2}$ can be defined as

$$b(i - j) = \frac{n_b + 1}{2} + \left\lfloor \frac{\log(i - j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b + 1}{2} \right\rfloor \quad (2.78)$$

where the parameter dist_{\max} is typically set to a relatively large number to indicate the maximum offset we may encounter.

- When $i - j > \text{dist}_{\max}$, we place $i - j$ in the last bucket. In other words, bucket n_b contains all the offsets that are not assigned to the previous buckets.

Together, these can be expressed as the function

$$b(i - j) = \begin{cases} i - j & 0 \leq i - j < \frac{n_b+1}{2} \\ \min(n_b, \frac{n_b+1}{2} + \left\lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \right\rfloor) & i - j \geq \frac{n_b+1}{2} \end{cases} \quad (2.79)$$

Figure 2.10 shows an illustration of these buckets. We see that in the first half of the buckets, each bucket is associated with only one value of $i - j$, while in the second half, the bucket size increases as $i - j$ grows. The last bucket is designed to handle sequences of arbitrarily long lengths.

¹⁵For language modeling, a query is only allowed to attend to its left-context, and so we have $i - j \geq 0$. In the more general case of self-attention, where a token can attend to all tokens in the sequence, we may have negative offsets when $i < j$.

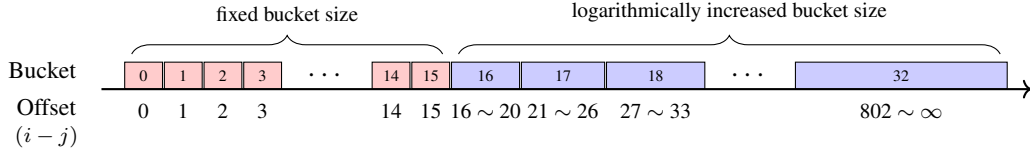


Fig. 2.10: Illustration of distributing query-key offsets into buckets in the T5 model ($n_b = 32$ and $\text{dist}_{\max} = 1024$). Boxes represent buckets. In the first half of the buckets, we use a fixed bucket size. In the second half of the buckets, we increase the bucket size logarithmically. The last bucket contains all the query-key offsets that are not covered by previous buckets.

All $\text{PE}(i, j)$ s in a bucket share the same bias term $u_{b(i-j)}$. Substituting $\text{PE}(i, j) = u_{b(i-j)}$ into Eq. (2.76), the attention weight for \mathbf{q}_i and \mathbf{k}_j becomes¹⁶

$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + u_{b(i-j)}}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (2.81)$$

The parameters $\{u_0, \dots, u_{n_b}\}$ are learned as common parameters during training. It should be emphasized that this model can generalize to long sequences. This is because $\text{PE}(i, j)$ s with similar query-key offsets share the same parameter, and this sharing strategy is particularly important for achieving good generalization, given that large query-key offsets are rare in training. In practice, we often set n_b to a moderate number, and thus it can help control the overfitting of positional embedding models.

2.3.5.2 Attention with Non-learned Biases

Relative positional embedding models are based on a set of learned biases for the query-key product in self-attention. An alternative approach is to give these biases fixed values via heuristics, rather than training them on a particular dataset. One benefit of this heuristics-based approach is that it does not rely on a training process and thus can be directly applied to any sequences once the biases are set.

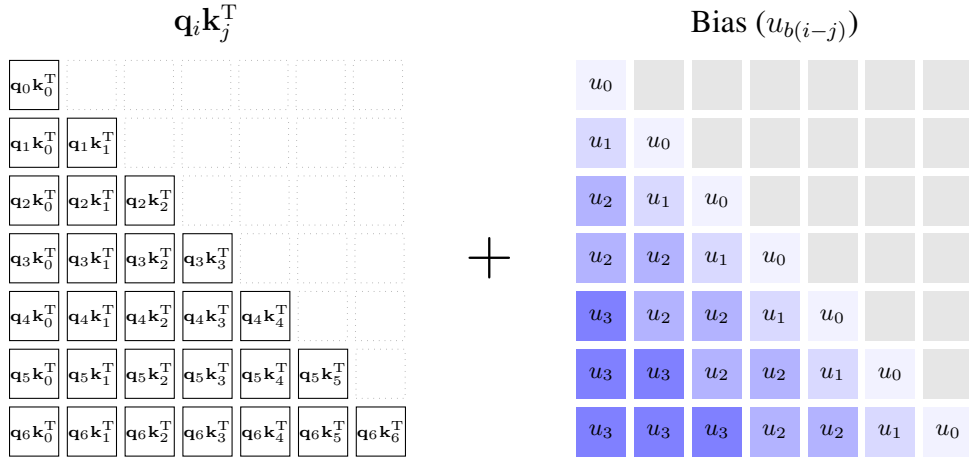
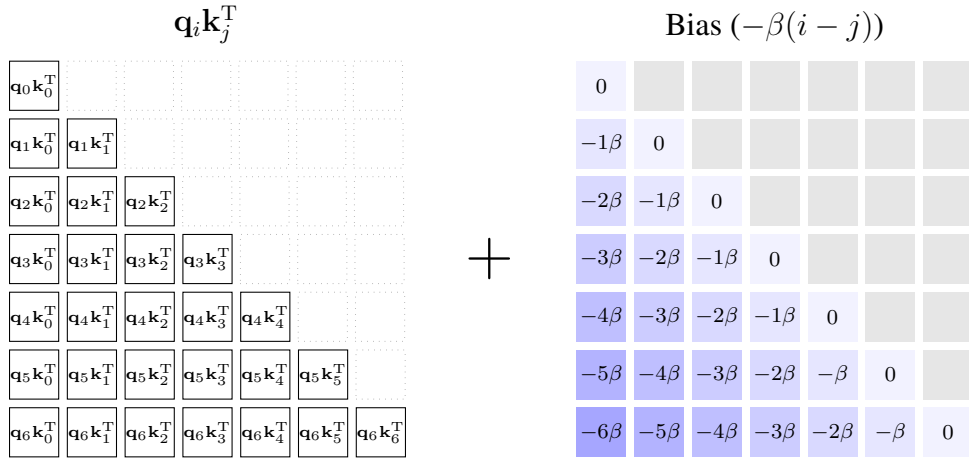
One example of such an approach is Press et al. [2022]’s approach, called **attention with linear biases** or **ALiBi** for short. In the ALiBi approach, the bias term is defined as the negative scaled query-key offset

$$\begin{aligned} \text{PE}(i, j) &= -\beta \cdot (i - j) \\ &= \beta \cdot (j - i) \end{aligned} \quad (2.82)$$

where β is the scaling factor. Adding this term to the query-key product, we obtain a new form of

¹⁶Note that, in Raffel et al. [2020]’s T5 model, the rescaling operation for the query-key product is removed. The attention weight $\alpha(i, j)$ is then given by

$$\alpha(i, j) = \text{Softmax}(\mathbf{q}_i \mathbf{k}_j^T + u_{b(i-j)} + \text{Mask}(i, j)) \quad (2.80)$$

(a) The T5 bias ($n_b = 3$ and $\text{dist}_{\max} = 5$)

(b) The ALiBi bias

Fig. 2.11: Query-key products with biases (above = the T5 bias and below = the ALiBi bias). The color scale of the biases ranges from light blue denoting small absolute values to deep blue denoting large absolute values.

attention weights

$$\alpha(i, j) = \text{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^T + \beta \cdot (j - i)}{\sqrt{d}} + \text{Mask}(i, j)\right) \quad (2.83)$$

This model can be interpreted as adding a fixed penalty to $\mathbf{q}_i \mathbf{k}_j^T$ whenever j moves one step away from i . So we do not need to adapt it to a range of sequence lengths, and can employ it to model arbitrarily long sequences. See Figure 2.11 for a comparison of the T5 bias and the ALiBi bias.

In general, the scalar β should be tuned on a validation dataset. However, Press et al. [2022] found that setting β to values decreasing geometrically by a factor of $\frac{1}{2^a}$ for multi-head attention performs well on a variety of tasks. Specifically, for a self-attention sub-layer involving n_{head}

Entry	Query-Key Bias (PE(i, j))
T5 [Raffel et al., 2020]	$u_{b(i-j)}$
ALiBi [Press et al., 2022]	$-\beta \cdot (i - j)$
Kerple [Chi et al., 2022]	$-\beta_1 (i - j)^{\beta_2}$ (power) $-\beta_1 \log(1 + \beta_2 (i - j))$ (logarithmic)
Sandwich [Chi et al., 2023]	$\sum_{k=1}^{\bar{d}/2} \cos((i - j)/10000^{2k/\bar{d}})$
FIRE [Li et al., 2024b]	$f(\psi(i - j)/\psi(\max(m_{\text{len}}, i)))$

Table 2.4: Query-key biases as relative positional embeddings. $\beta, \beta_1, \beta_2, \bar{d}$, and m_{len} are hyper-parameters. In the T5 model, $b(i - j)$ denotes the bucket assigned to $i - j$. In the FIRE model, $\psi(\cdot)$ is a monotonically increasing function such as $\psi(x) = \log(cx + 1)$, and $f(\cdot)$ is an FFN.

heads, the scalar for the k -th head is given by

$$\beta_k = \frac{1}{2^{\frac{8}{k}}} \quad (2.84)$$

The ALiBi approach provides a simple form of relative positional embeddings. There are other similar methods for designing query-key biases using the offset $i - j$. Table 2.4 shows a comparison of such biases. As an aside it is worth noting that the form of the right-hand side of Eq. (2.82) is very similar to length features used in conventional feature-based systems. For example, in statistical machine translation systems, such features are widely used to model word reordering problems, resulting in models that can generalize well across different translation tasks [Koehn, 2010].

2.3.5.3 Rotary Positional Embedding

As with sinusoidal embeddings, rotary positional embeddings are based on hard-coded values for all dimensions of an embedding [Su et al., 2024]. Recall that in the sinusoidal embedding model, positions are represented as combinations of sine and cosine functions with different frequencies. These embeddings are then added to token embeddings to form the inputs to the Transformer layer stack. Rotary positional embeddings instead model positional context as rotations to token embeddings in a complex space. This leads to a model expressed in the form of multiplicative embeddings

$$\mathbf{e}_i = \mathbf{x}_i R(i) \quad (2.85)$$

where $R(i) \in \mathbb{R}^{d \times d}$ is the rotation matrix representing the rotations performed on the token embedding $\mathbf{x}_i \in \mathbb{R}^d$.

For simplicity, we will first consider embeddings with only two dimensions and return to a discussion of the more general formulation later. Suppose we have a 2-dimensional token embedding $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$. We can represent it as a vector in a plane, originating at the origin $(0, 0)$ and terminating at (x_1, x_2) . A counterclockwise rotation of this vector refers to an operation of moving the vector around the origin while maintaining its magnitude, as shown in Figure 2.12 (a). The degree of rotation is usually defined by a specific angle, denoted by θ . The rotation can be

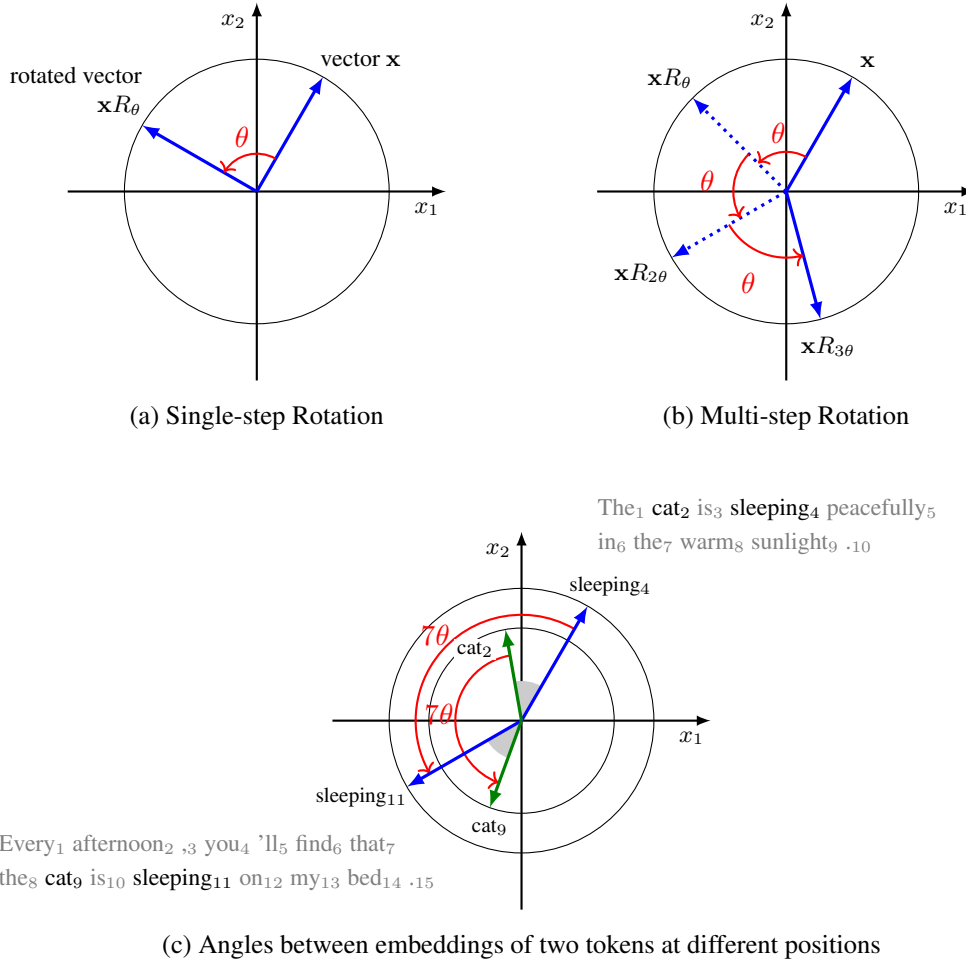


Fig. 2.12: Illustrations of vector rotations in a plane. Sub-figures (a) and (b) show rotations of a vector in a single step and multiple steps, respectively. Sub-figure (c) shows the embeddings of tokens *cat* and *sleeping* in two different sentences. We show these sentences with a subscript affixed to each token to indicate its position. If we represent tokens as vectors, we can add positional information by rotating these vectors. This rotation preserves the “distances” between the vectors. For example, given that the distance between *cat* and *sleeping* is the same in both sentences, the angle between their embeddings also remains the same during rotation.

expressed mathematically in the form

$$\begin{aligned}
 \text{Ro}(\mathbf{x}, \theta) &= \mathbf{x}R_\theta \\
 &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \\
 &= \begin{bmatrix} \cos \theta \cdot x_1 - \sin \theta \cdot x_2 & \sin \theta \cdot x_1 + \cos \theta \cdot x_2 \end{bmatrix} \quad (2.86)
 \end{aligned}$$

where $R_\theta = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$ is the rotation matrix. If two or more rotations are performed on the same vector, we can rotate the vector further. This follows from the fact that the composition of successive rotations is itself a rotation. More formally, rotating a vector by an angle θ for t times

can be expressed as

$$\begin{aligned}\text{Ro}(\mathbf{x}, t\theta) &= \mathbf{x}R_{t\theta} \\ &= \begin{bmatrix} \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 & \sin t\theta \cdot x_1 + \cos t\theta \cdot x_2 \end{bmatrix}\end{aligned}\quad (2.87)$$

If we interpret t as the position of a token represented by \mathbf{x} in a sequence, then we will find that the above equation defines a simple positional embedding model. As shown in Figure 2.12 (b), we start moving the token from position 0. Each time we move one step forward, the vector is rotated by the angle θ . Upon arriving at the position t , the representation of the token with positional context is given by $\text{Ro}(\mathbf{x}, i\theta)$. As the rotations do not change the magnitude of the embedding, the original “meaning” of the token is retained. The positional information is injected into the embedding, when it gets rotated.

A popular way to understand vector rotation is to define it in complex spaces. It is easy to transform each vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ in the 2D Euclidean space \mathbb{R}^2 to a complex number $\mathbf{x}' = x_1 + ix_2$ in the complex space \mathbb{C} via a bijective linear map. Then, the rotation of \mathbf{x} with the angle $t\theta$ corresponds to the multiplication by $e^{it\theta}$. Given that $e^{it\theta} = \cos t\theta + i \sin t\theta$, the rotation operation can be re-expressed in the form

$$\begin{aligned}\mathbf{x}R_{t\theta} &\mapsto \mathbf{x}'e^{it\theta} \\ &= (x_1 + ix_2)(\cos t\theta + i \sin t\theta) \\ &= \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 + i(\sin t\theta \cdot x_1 + \cos t\theta \cdot x_2)\end{aligned}\quad (2.88)$$

Here we denote the token representation $\mathbf{x}'e^{it\theta}$ by $C(\mathbf{x}, t\theta)$. The inner product of the representations of the tokens at positions t and s can be written as

$$\langle C(\mathbf{x}, t\theta), C(\mathbf{y}, s\theta) \rangle = (\mathbf{x}'\overline{\mathbf{y}'})e^{i(t-s)\theta} \quad (2.89)$$

where $\overline{\mathbf{y}'}$ is the complex conjugate of \mathbf{y}' . As can be seen, the result of this inner product involves a term $t - s$, and so it can model the offset between the two tokens.

Now we go back to representations in the 2D Euclidean space. The dot-product of $\text{Ro}(\mathbf{x}, t\theta)$ and $\text{Ro}(\mathbf{y}, s\theta)$ is can be written as a function of $(t - s)\theta$

$$\begin{aligned}\text{Ro}(\mathbf{x}, t\theta)[\text{Ro}(\mathbf{y}, s\theta)]^T &= \mathbf{x}R_{t\theta}[\mathbf{y}R_{s\theta}]^T \\ &= \mathbf{x}R_{t\theta}[R_{s\theta}]^T\mathbf{y}^T \\ &= \mathbf{x}R_{(t-s)\theta}\mathbf{y}^T\end{aligned}\quad (2.90)$$

Given this result, if we consider $\text{Ro}(\mathbf{x}, t\theta)$ and $\text{Ro}(\mathbf{y}, s\theta)$ as the query and the key, then the self-attention operation will implicitly involve the modeling of relative positional context.

This rotary positional embedding can be extended to multi-dimensional embeddings. For a d -dimensional token embedding $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix}$, we can treat it as a $\frac{d}{2}$ -dimensional complex vector $\mathbf{x}' = \begin{bmatrix} x'_1 & x'_2 & \dots & x'_{d/2} \end{bmatrix} = \begin{bmatrix} x_1 + ix_2 & x_3 + ix_4 & \dots & x_{d-1} + ix_d \end{bmatrix}$, where each consecutive pair of items forms a complex number. Then, the rotary positional embedding in

the complex space is given by

$$C(\mathbf{x}, t\theta) = \sum_{k=1}^{d/2} x'_k e^{it\theta_k} \vec{e}_k \quad (2.91)$$

where \vec{e}_k is the standard basis vector with a single non-zero value in the k -th coordinate and 0's elsewhere [Biderman et al., 2021].

Although this formula involves a complicated expression, its equivalent form in the d -dimensional Euclidean space is relatively easy to understand. We can write it as

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix} \begin{bmatrix} R_{t\theta_1} & & & \\ & R_{t\theta_2} & & \\ & & \ddots & \\ & & & R_{t\theta_{d/2}} \end{bmatrix} \quad (2.92)$$

where $R_{t\theta_k} = \begin{bmatrix} \cos t\theta_k & \sin t\theta_k \\ -\sin t\theta_k & \cos t\theta_k \end{bmatrix}$. $\theta = [\theta_1, \dots, \theta_{d/2}]$ are the parameters for controlling the angles of rotations in different dimensions. Typically, θ_k is set to $10000^{-\frac{2(k-1)}{d}}$, which is analogous to the setting in sinusoidal embeddings.

In a practical implementation, Eq. (2.92) can be rewritten into a form that relies solely on the element-wise product and addition of vectors.

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d-1} \\ x_d \end{bmatrix}^T \odot \begin{bmatrix} \cos t\theta_1 \\ \cos t\theta_1 \\ \vdots \\ \cos t\theta_{d/2} \\ \cos t\theta_{d/2} \end{bmatrix}^T + \begin{bmatrix} -x_2 \\ x_1 \\ \vdots \\ -x_d \\ x_{d-1} \end{bmatrix}^T \odot \begin{bmatrix} \sin t\theta_1 \\ \sin t\theta_1 \\ \vdots \\ \sin t\theta_{d/2} \\ \sin t\theta_{d/2} \end{bmatrix}^T \quad (2.93)$$

Finally, we rewrite Eq. (2.85) to obtain the form of the embedding at position i

$$\mathbf{e}_i = \text{Ro}(\mathbf{x}_i, i\theta) \quad (2.94)$$

2.3.5.4 Position Interpolation

In position interpolation, our goal is to map the positions in the new sequence to match the observed range in training. Suppose the sequence length for training ranges from 0 to m_l . When $m > m_l$ at test time, we represent the positions in $[0, m]$ such that our representations fit $[0, m_l]$.

To illustrate, consider the rotary positional embedding model described above. The embedding of each token is described by a model $\text{Ro}(\mathbf{x}_i, i\theta)$ in which $\theta = [\theta_1, \dots, \theta_{d/2}]$ are the parameters. $\text{Ro}(\mathbf{x}_i, i\theta)$ can be cast in the form of a linear combination of two periodic functions (see Eq.

(2.93))

$$\cos i\theta = \begin{bmatrix} \cos i\theta_1 & \dots & \cos i\theta_{d/2} \end{bmatrix} \quad (2.95)$$

$$\sin i\theta = \begin{bmatrix} \sin i\theta_1 & \dots & \sin i\theta_{d/2} \end{bmatrix} \quad (2.96)$$

θ_k is an exponential function of k and takes the form

$$\theta_k = b^{-\frac{2(k-1)}{d}} \quad (2.97)$$

where b is the base. The period of $\cos i\theta_k$ and $\sin i\theta_k$ is

$$T_k = 2\pi \cdot b^{\frac{2(k-1)}{d}} \quad (2.98)$$

The key idea behind position interpolation is to adjust this period so that the new positions can be encoded within the range $[0, m_l]$. One way to achieve this is to scale up T_k by $\frac{m}{m_l}$, given by

$$T'_k = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(k-1)}{d}} \quad (2.99)$$

Hence all points in $[0, m]$ are compressed into $[0, m_l]$. This linear scaling can be easily realized by modifying the input to the embedding model [Chen et al., 2023c]. The new model with linear positional interpolation is given by

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}(\mathbf{x}_i, \frac{m_l}{m} i\theta) \quad (2.100)$$

Another method of positional interpolation is to scale the base¹⁷. Suppose that the base b is scaled by λ . We wish the period of this new model in the last dimension of θ (i.e., dimension $\frac{d}{2}$) to be equal to that of the linear positional interpolation model. This can be expressed as

$$2\pi \cdot (\lambda b)^{\frac{2(\frac{d}{2}-1)}{d}} = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(\frac{d}{2}-1)}{d}} \quad (2.101)$$

Solving this equation, we obtain

$$\begin{aligned} \lambda &= \left(\frac{m}{m_l} \right)^{\frac{d}{2(\frac{d}{2}-1)}} \\ &= \left(\frac{m}{m_l} \right)^{\frac{d}{d-2}} \end{aligned} \quad (2.102)$$

This gives an embedding model

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}(\mathbf{x}_i, i\theta') \quad (2.103)$$

¹⁷This method was first proposed in https://www.reddit.com/r/LocalLLaMA/comments/141z7j5/ntkaware_scaled_rope_allows_llama_models_to_have/

where

$$\theta' = \left[(\lambda b)^{-\frac{0}{d}}, (\lambda b)^{-\frac{2}{d}}, \dots, (\lambda b)^{-\frac{d-2}{d}} \right] \quad (2.104)$$

Note that scaling the base provides a non-uniform method for scaling the periods across different dimensions of θ . This method has been found to be helpful for extending LLMs to longer sequences, and several improvements have been developed [Peng et al., 2024; Ding et al., 2024].

2.3.6 Remarks

In this section, we have presented a variety of methods for long-context language modeling. We close this section by discussing some interesting issues related to these methods.

2.3.6.1 Need for Long Context

One of the ultimate goals of long-context LLMs is that these models can precisely encode infinite context. The so-called infinite context refers more to the fact that an LLM can continuously read words. This motivates LLMs that can handle extremely long context or stream data. As discussed in Section 2.3.3, it is common to use fixed-size memory models to process continuously expanding context. Many such systems are based on recurrent architectures or their variants, because they are inherently suited to model time series problems where the effects of past inputs continue indefinitely. Another way to achieve infinite memory is to develop alternatives to self-attention models, for example, one can use continuous-space attention models to encode context, which removes the dependency on context length [Martins et al., 2022].

When studying long-context LLMs, it is natural to wonder what mechanisms may explain the use of long context in language modeling. Can we compress the representation of infinite context into a relatively small-sized model? Are all context tokens useful for predicting next tokens? How do LLMs prepare for token prediction when they see the context? Can we know in advance which contextual information will be critical for prediction? General answers to all these questions are not obvious, but they inspire follow-on research of explainable models, and some interesting results have been found. For example, Deletang et al. [2024] conducted extensive experiments to show that LLMs are powerful in-context compressors. Although viewing predictive models as compression models has long been studied in machine learning, it also provides insights into our understanding of the LLM scaling laws. Pal et al. [2023] and Wu et al. [2024] investigated whether the features learned up to the current step, though not intentionally, are already sufficient for predicting tokens at the following steps. Note that the need for long-context in language modeling is highly dependent on the problem that we address. A related issue is where to apply LLMs and how to evaluate them. For example, in summarization tasks we may only need to distill and focus on a few key aspects of the text, while in retrieval-like tasks we need to “memorize” the entire context so that the relevant information can be accessed. We will discuss the evaluation issue later in this subsection.

2.3.6.2 Pre-training or Adapting LLMs?

Training LLMs requires significant computational costs. Although it is straightforward to train LLMs on long sequence data, the training becomes computationally unwieldy for large data sets. It

is common practice to pre-train LLMs on general datasets, and then adapt them with modest fine-tuning effort. For example, LLMs with relative or rotary positional embeddings can be directly trained on large-scale data in the pre-training phase. While the resulting models may exhibit some abilities to extrapolate lengths in the inference phase, it may be more effective to fine-tune them on longer sequences.

Ideally, we would like to pre-train LLMs with standard Transformer architectures and adapt them to new tasks. This allows us to use many off-the-shelf LLMs and efficiently adapt them to handle long sequences. However, when new architectures are adopted, it seems inevitable that we need to train these models from scratch. This poses practical difficulties for developing long-context LLMs, as we cannot leverage well-developed, pre-trained models and must instead train them ourselves. On the other hand, fine-tuning is still an effective way to adapt LLMs with certain architectures that are different from those in pre-training. An example is models augmented with external memories. In these models, the pre-trained LLMs are fixed, and the focus is on how to make these LLMs collaborate with the memory models. In RAG, for instance, it is common to fine-tune LLMs to improve their use of retrieval-augmented inputs. Another example of fine-tuning LLMs for long-context modeling is that we train an LLM with full attention models, and then replace them with sparse attention models in the fine-tuning phase. The pre-trained LLM provides initial values of model parameters used in a different model, and this model is then fine-tuned as usual.

2.3.6.3 Evaluating Long-context LLMs

Evaluating long-context LLMs is important, but it is a new issue in NLP. The general idea is that, if we input a long context to an LLM, then we can check from the output of the LLM whether it understands the entire context and makes use of it in predicting following tokens. In conventional research of NLP, such evaluations are often aimed at examining the ability of NLP models in handling long-range dependencies. However, the size of contexts used in recent LLMs is much larger than that used in NLP systems a few years ago. This motivates researchers to develop new evaluation benchmarks and metrics for long-context LLMs.

One approach is to use the perplexity metric. However, in spite of its apparent simplicity, this method tends to reflect more on the LLMs' ability to make use of local context rather than global context. It is therefore tempting to develop evaluation methods that are specific to long-context LLMs. Popular methods include various synthetic tasks where artificially generated or modified data is used to evaluate specific capabilities of long-context LLMs. In needle-in-a-haystack¹⁸ and passkey retrieval tasks [Mohtashami and Jaggi, 2024; Chen et al., 2023c], for instance, LLMs are required to identify and extract a small, relevant piece of information from a large volume of given text. The assumption here is that an LLM with sufficient memory should remember earlier parts of the text as it processes new information. This LLM can thus pick out the relevant details, which might be sparse and hidden among much irrelevant information, from the text. Alternatively, in copy memory tasks (or copy tasks for short), LLMs are used to repeat the input text or a specific segment multiple times. These tasks were initially proposed to test the extent to which recurrent models can retain and recall previously seen tokens [Hochreiter and Schmidhuber, 1997; Arjovsky et al., 2016], and have been adopted in evaluating recent LLMs [Bulatov et al., 2022; Gu and Dao, 2023].

¹⁸https://github.com/gkamradt/LLMTest_NeedleInAHaystack

Another approach to evaluating long-context LLMs is to test them on NLP tasks that involve very long input sequences. Examples include long-document or multi-document summarization, long-document question answering, code completion, and so on. A benefit of this approach is that it can align evaluations with user expectations.

Although many methods have been developed, there is still no general way to evaluate long-context LLMs [Liu et al., 2024c]. One problem is that most of these methods focus on specific aspects of LLMs, rather than their fundamental ability to model very long contexts. Even though an LLM can pick out the appropriate piece of text from the input, we cannot say that it truly understands the entire context. Instead, it might just remember some important parts of the context, or even simply recall the answer via the model learned in pre-training. Moreover, the data used in many tasks is small-scale and relatively preliminary, leading to discrepancies between evaluation results and actual application performance. A more interesting issue is that the results of LLMs are influenced by many other factors and experimental setups, for example, using different prompts can lead to very different outcomes. This makes evaluation even more challenging because improvements may not solely result from better modeling of long contexts, and there is a risk of overclaiming our results. Nevertheless, many open questions remain in the development and evaluation of long-context LLMs. For example, these models still suffer from limitations such as restricted context length and high latency. Studying these issues is likely to prove valuable future directions.

2.4 Summary

In this chapter, we have discussed the concept of LLMs and related techniques. This can be considered a general, though not comprehensive, introduction to LLMs, laying the foundation for further discussions on more advanced topics in subsequent chapters. Furthermore, we have explored two ways to scale up LLMs. The first focuses on the large-scale pre-training of LLMs, which is crucial for developing state-of-the-art models. The second focuses on methods for adapting LLMs to long inputs, including optimizing attention models, designing more efficient and compressed KV caches, incorporating memory models, and exploring better positional embeddings.

The strength of LLMs lies in their ability to break the constraints of training NLP models for a limited number of specific tasks. Instead, LLMs learn from large amounts of text through the simple task of token prediction — we predict the next token in a sentence given its prior tokens. A general view is that, by repeating this token prediction task a large number of times, LLMs can acquire some knowledge of the world and language, which can then be applied to new tasks. As a result, LLMs can be prompted to perform any task by framing it as a task of predicting subsequent tokens given prompts. This emergent ability in language models comes from several dimensions, such as scaling up training, model size, and context size. It is undeniable that scaling laws are currently the fundamental principle adopted in developing large language models, although simply increasing model size has yet to prove sufficient for achieving AGI. These continuously scaled LLMs have been found to show capabilities in general-purpose language understanding, generation, and reasoning. More recently, it has been found that scaling up the compute at inference time can also lead to significant improvements in complex reasoning tasks [OpenAI, 2024].

Given their amazing power, LLMs have attracted considerable interest, both in terms of techniques and applications. As a result, the explosion of research interest in LLMs has also led to a

vast number of new techniques and models. However, we do not attempt to provide a comprehensive literature review on all aspects of LLMs, given the rapid evolution of the field. Nevertheless, one can still gain knowledge about LLMs from general reviews [[Zhao et al., 2023](#); [Minaee et al., 2024](#)] or more focused discussions on specific topics [[Ruan et al., 2024](#)].

CHAPTER 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. → Label: Positive

Example 2: I'm frustrated with the delays. → Label: Negative

What is the label for "That comment was quite hurtful."?

Label: _____

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

In this chapter, we focus on prompting techniques in LLMs. We begin by considering several interesting prompt designs commonly used in prompt engineering. Then, we discuss a series of

refinements to these methods. Finally, we explore approaches for automating prompt design.

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 x . The LLM generates a text y by maximizing the probability $\Pr(y|x)$. In this generation process, the prompt acts as the condition on which we make predictions, and it can contain any information that helps describe and solve the problem.

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

Please give me some suggestions for a fun weekend.

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

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

premise = the weather is nice this weekend

then we can generate a prompt

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

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


```

Here is a sentence
{*sentence1*}
Here is another sentence
{*sentence2*}
Compute the semantic similarity between the two sentences
_____

```

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

```

John: {*utterance1*}
David: {*utterance2*}
John: {*utterance3*}
David: {*utterance4*}
John: {*utterance5*}
David: {*utterance6*}
John: {*utterance7*}
David: _____

```

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

```

Q: {*question*}
A: _____

```

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

```

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

```

In practical systems, it is common to represent and store such data in key-value pairs, such as the JSON format¹.

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

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

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

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

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.

{*problem*}

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

You will be provided with a math problem. Please solve the problem.

{*problem*}

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

You will be provided with a math problem, along with a solution. Evaluate the correctness of this solution, and identify any errors if present. Then, work out your own solution.

Problem: {**problem**}

Solution: {**solution**}

The prompts presented here are closely related to a long line of research on reasoning problems in LLMs. It is impossible to provide a complete discussion of all related issues because this topic covers a large family of methods. But we will see a relatively more detailed discussion on how to improve prompting through more reasoning in Section 3.2.

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

You are an expert that can generate answers to input queries. You have now been provided with a query and the corresponding context information. Please generate an answer based on this context information. Note that you need to provide the answer in your own words, not just copy from the context provided.

Context information: {**IR-result**}

Query: {**query**}

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

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

Context information: {**table**}

Query: {**query**}

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

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

```
[English] = [I have an apple.]  
[German] = [Ich habe einen Apfel.]  
[English] = [I have an orange.]  
[German] = _____
```

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

Above, we have discussed only a few strategies for writing good prompts. There are, of course, many such methods, and one needs to develop their own through practice. Interested readers can refer to various online documents for more information, such as OpenAI’s manual on the GPT series models².

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

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 polarity classification, we are required to categorize a given text into one of three categories: negative, positive, or neutral. Below is a simple prompt for doing this (for easy reading, we highlight the task description in the prompt).

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

The polarity of the text can be classified as negative.

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

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

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

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

The polarity of the text is negative

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

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

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

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

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

What is the polarity of the text?

Just answer: positive, negative, or neutral.

Negative

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

Analyze the polarity of the following text and classify it as positive, negative, or neutral. Here's what each category represents:

Positive: This indicates that the text conveys a positive emotion or attitude. For example, texts expressing happiness, satisfaction, excitement, or admiration are considered positive.

Negative: This refers to a text that expresses a negative emotion or attitude. It encompasses feelings of sadness, anger, frustration, or criticism.

Neutral: Neutral sentiment is used to describe texts that do not exhibit clear positive or negative emotions but instead convey informational, factual, or indifferent tones.

Text:

The service at the restaurant was slower than expected, which was a bit frustrating.

What is the polarity of the text?

Negative

While it seems straightforward to use LLMs for classification problems, there are still issues that have not been well addressed. For example, when dealing with a large number of categories, it remains challenging to effectively prompt LLMs. Note that if we face a very difficult classification problem and have a certain amount of labeled data, fine-tuning LLMs or adopting “BERT + classifier”-like architectures is also desirable.

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 text³.

³The text is from <https://edition.cnn.com/travel>

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

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

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

Text:

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

Named Entity:

Tom Jenkins, European Tourism Organisation, UK, Europe, 2024

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

...

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

You will be provided with a text. Your task is to {**task-description**}

Text: {**text**}

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

extract the keywords from the text.

extract the key events from the text.

detect the coreference and anaphoric links between entities from the text.

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

3.1.4.3 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 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 $\ll \dots \gg$ annotates the detailed calculation steps (called **calculation annotation**)⁴.

3.2 Advanced Prompting Methods

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

3.2.1 Chain of Thought

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

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

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

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

A: The answer is 6.

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

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

A: The answer is 4.

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

A: The answer is 7.

The problem here is that, although we have shown a similar question-answer pair, it remains difficult for the LLM to reason out the correct answer. In CoT, not only can LLMs learn from the

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

correspondence between questions and answers but they may gain more from detailed problem-solving steps that are used to derive the answers. To do this, we can incorporate some reasoning steps into the prompt to obtain a CoT prompt.

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

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

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

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

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

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

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

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

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

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

Following the instruction “Let’s think step by step”, the LLM is prompted to generate detailed reasoning steps. As discussed in [Kojima et al. \[2022\]](#)’s work, prompting with such instructions may result in LLMs generating only the reasoning steps without a clear conclusion. In this case, a second round of prompting can be used to extract the answer from these reasoning steps. For example, [Kojima et al. \[2022\]](#) create a second prompt which combines both the input and output

in the first round of prompting. Using this combined input, the LLM can continue its reasoning process and then generate the correct answer. Furthermore, it is possible to prompt LLMs to reason using instructions other than “Let’s think step by step”, such as “Let’s think logically” and “Please show me your thinking steps first”.

While we have illustrated CoT methods using an algebraic reasoning problem, these methods can be applied to a variety of different problems. Typical problem-solving scenarios for CoT include mathematical reasoning, logical reasoning, commonsense reasoning, symbolic reasoning, code generation, and so on. See Figure 3.1 for more examples of applying CoT in various tasks.

CoT today is one of the most active fields of prompt engineering. This has not only led to improved performance for LLM prompting but has opened the door to a wide range of methods for studying and verifying reasoning capabilities of LLMs. Although we have focused on the basic idea of CoT in this section, it can be improved in several ways. For example, we can consider the reasoning process as a problem of searching through many possible paths, each of which may consist of multiple intermediate states (i.e., reasoning steps). In general, we wish the search space to be well-defined and sufficiently large, so that we are more likely to find the optimal result. For this reason, an area of current LLM research is aimed at designing better structures for representing reasoning processes, allowing LLMs to tackle more complex reasoning challenges. These structures include tree-based structures [Yao et al., 2024], graph-based structures [Besta et al., 2024], and so on. By using these compact representations of reasoning paths, LLMs can explore a wider range of decision-making paths, analogous to System 2 thinking⁵. Another line of research focuses on prompting LLMs with multi-round interactions. This involves decomposing complex problems into sub-problems, verifying and refining model outputs, employing model ensembling, and so on. Note that these methods and the issues involved are not limited to CoT. In fact, they are often used as more general approaches to improving LLMs, while CoT can be seen as a way to test the capabilities of LLMs. We will see discussions of some of these issues in the following subsections.

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

3.2.2 Problem Decomposition

We have seen that LLMs can benefit from solving a complex problem by breaking it down into simpler problem-solving tasks. Such an approach can be seen as an example of a broader paradigm known as **problem decomposition**, which has been extensively explored and discussed in psychology and computer science. From the psychological perspective, complex problem-solving refers to a process of addressing a problem using knowledge that helps overcome the barriers of

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

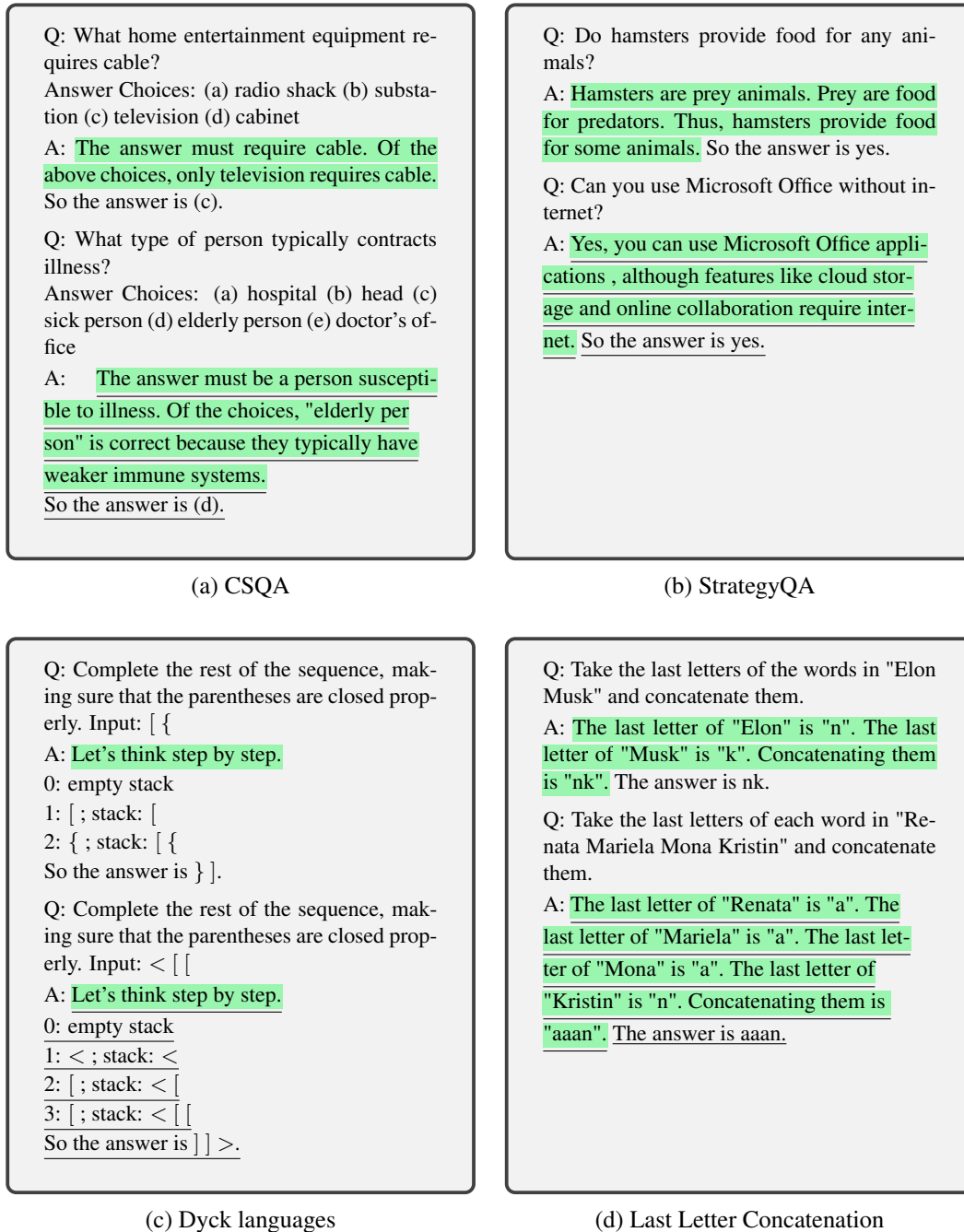


Fig. 3.1: CoT in four different reasoning tasks, including CSQA, StrategyQA, Dyck languages, and Last Letter Concatenation. The CoT parts are highlighted in green.

the problem⁶. There are generally no standard or clear paths to a solution for a complex problem. However, it is often advantageous to employ strategies that decompose the problem, thereby making it easier to tackle the corresponding sub-problems with less effort. For example, consider writing a blog about the risks of AI. If we simply prompt an LLM with the instruction “Please write a blog about the risks of AI”, the LLM may generate a blog with arbitrary structures and

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

writing styles. A better method, instead, could be to outline the blog and provide more detailed information about each section. Consider the following prompt

You are a blog writer. Please follow the provided outline below to write a blog about the risks of AI.

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

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

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

You are provided with a text. Please determine whether it discusses the risks of AI.

{*document*}

If the document is long, the computation will be expensive. Alternatively, we can divide the document into relatively short segments and perform the same task on each segment. These segments can be processed in parallel to further reduce the computational cost. Next, we determine

the relevancy of each segment to the topic of AI risks. The final output is then generated using another prompt.

Your task is to determine whether a text discusses the risks of AI. This text has been divided into segments, and you have obtained the relevancy of each segment to the topic of AI risks. Based on this, please provide your final result.

Segment 1: {**relevancy-to-the-topic1**}

Segment 2: {**relevancy-to-the-topic2**}

Segment 3: {**relevancy-to-the-topic3**}

...

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

A general framework for problem decomposition involves two elements.

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

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

However, many real-world problems require complex reasoning. One key characteristic of these problems is that the reasoning steps may not be fixed. The reasoning path can vary for different problems, and each step of reasoning may depend on the outcomes of prior steps. In

such cases, it is undesirable to use fixed sub-problem generation in advance. Instead, sub-problems should be generated dynamically based on the input problem, and, if possible, generated on the fly during the reasoning process. This makes problem decomposition more challenging compared with designing divide-and-conquer algorithms. Ideally, we would like to jointly design both the systems for sub-problem generation and sub-problem solving. But a more practical and widely used approach is to adopt separate models for these tasks. A straightforward way to achieve this is to adapt an LLM for these tasks by either prompting or tuning the model.

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

TASK	Your task is to decompose a problem into several sub-problems. You will be given a few examples to illustrate how to achieve this.
DEMO	<p>Q: In a community, 5% of the population are infants, 15% are children, 40% are adults, and 40% are seniors. Which group makes up the largest portion of the population?</p> <p>A: To answer the question “Which group makes up the largest portion of the population?”, we need to know: “How many percent are infants?”, “How many percent are children?”, “How many percent are adults?”, “How many percent are seniors?”.</p> <p>Q: Alice, Bob, and Charlie brought beads for their group project in their craft class. Alice has twice as many beads as Bob, and Bob has five times as many beads as Charlie. If Charlie has 6 beads, how many beads can they use for their craft project?</p> <p>A: To answer the question “How many beads can they use for their craft project?”, we need to know: “How many beads does Bob have?”, “How many beads does Alice have?”.</p>
USER	<p>Q: The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius. What was the duration of the environmental study?</p> <p>A: To answer the question “What was the duration of the environmental study?”, we need to know: “When did the environmental study start?”, “When did the environmental study end?”.</p>

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

we need to answer the first sub-problem by prompting the LLM, like this

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

SUB-PROB1 Q: When did the environmental study start?
A: The environmental study started in 2015.

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

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

SUB-PROB1 Q: When did the environmental study start?
A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?
A: The environmental study ended in 2020.

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

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

SUB-PROB1 Q: When did the environmental study start?
A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?
A: The environmental study ended in 2020.

FINAL Q: What was the duration of the environmental study?
A: The duration of the environmental study was 5 years.

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

Another improvement is to explore methods for better decomposing problems and organizing problem-solving paths. To describe these approaches, we will use the symbol p_0 to denote the

input problem, and use the symbols $\{p_1, \dots, p_n\}$ to denote the sub-problems corresponding to p_0 . For least-to-most prompting, we decompose p_0 into $\{p_1, \dots, p_n\}$, given by

$$\{p_1, \dots, p_n\} = G(p_0) \quad (3.2)$$

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

$$a_i = S_i(p_i, \{p_0, p_{<i}, a_{<i}\}) \quad (3.3)$$

where $p_{<i} = \{p_1, \dots, p_{i-1}\}$ and $a_{<i} = \{a_1, \dots, a_{i-1}\}$. $S_i(\cdot)$ denotes the function that solves the sub-problem p_i given the context $\{p_0, p_{<i}, a_{<i}\}$. The last step is to generate the answer to the original problem p_0 , which can be expressed in a similar manner to Eq. (3.3).

$$a_0 = S_0(p_0, \{p_{\leq n}, a_{\leq n}\}) \quad (3.4)$$

One way to refine this model is to modify the $G(\cdot)$ function so that the model can dynamically generate answers. Instead of generating all sub-problems at one time, we can generate each of them during problem-solving [Dua et al., 2022]. To do this, we can replace Eq. (3.2) with

$$p_i = G_i(p_0, \{p_{<i}, a_{<i}\}) \quad (3.5)$$

Hence we obtain a sub-problem generation model that operates in a step-by-step manner. At each step i , we first generate the sub-problem p_i by prompting an LLM with the original problem p_0 and the problem-solving history $\{p_{<i}, a_{<i}\}$. We then generate the answer a_i for this sub-problem using the same or a different LLM, based on the same contextual information (see Eq. (3.3)). This method effectively expands the reasoning capacity of LLMs by allowing them to dynamically generate and solve sub-problems in intermediate reasoning steps. As a result, the reasoning paths are not fixed in advance, and the models can choose and adapt their reasoning strategies during problem-solving.

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

If we generalize the above formulation a bit further, we can consider it as a reinforcement learning problem. A typical method is to model a problem-solving process as a decision making process. In each step of this process, an action is taken based on the current state. These actions

can include all functions for sub-problem generation and solving (i.e., $G_i(\cdot)$ and $S_i(\cdot)$). Thus, the action sequence corresponds to a problem-solving path. Since the discussion of reinforcement learning problems is beyond the scope of this chapter, we skip the precise description of this learning task. Nevertheless, developing an agent or controller to determine when and how to generate and solve a sub-problem is also a natural choice.

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

Broadly speaking, problem decomposition is also related to the compositionality issue in NLP [Drozdov et al., 2022; Press et al., 2023]. For example, in semantic parsing, we map natural language sentences into structured meaning representations by breaking them down into constituent parts and understanding the sentences based on the meanings of these parts and the rules used to combine them. In early studies of this field, highly compositional sentences were considered easier for testing systems, as it is relatively straightforward to decompose such sentences and compose the meanings of their parts. However, the task becomes much more difficult when more generalization is required for modeling compositionality in new data. In this case, we want systems to have improved abilities of **compositional generalization**. In more recent research on LLMs, this issue has been frequently discussed in compositional reasoning tasks, such as SCAN⁷, as it is considered an important aspect of testing the language understanding and reasoning abilities of LLMs. This also presents new tasks for developing and examining problem decomposition methods.

In LLMs, one interesting application of problem decomposition is tool use. In some cases, it is necessary to integrate external tools into LLMs to access accurate data not available during training or fine-tuning. For example, LLMs can integrate with APIs to fetch real-time data such as weather updates, stock market prices, or news feeds, enabling them to provide up-to-date responses to user queries. When using tools, LLM predictions might include markers that indicate where and how to call external APIs. This requires decomposing the problem into sub-problems, with some handled by the LLMs and others by external tools. More detailed discussions on this topic will be presented in Section 3.2.5.

3.2.3 Self-refinement

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

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

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

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

Translate the following text from Chinese to English:

Chinese:

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

English:

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

Then, we can prompt it to refine the translation:

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

Chinese:

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

English:

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

Refinement:

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

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

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

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

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

The last two steps can be repeated multiple times, which leads to an iterative self-refinement process. In this framework, receiving accurate and detailed feedback is critical as it guides the subsequent refinement step. In general, feedback can be obtained in various ways [Fernandes et al., 2023; Pan et al., 2024]. We can manually examine the model’s predictions to identify issues. Also, we can use a feedback model to automatically generate feedback. For instance, a reward model trained on some labeled data can help evaluate the model output and return a score indicating its quality.

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

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

Generate a response to the question: “What are some major environmental concerns today?”

Response:

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

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

You will be provided with a question and its corresponding response. Evaluate the accuracy and depth of this response, and identify any factual inaccuracies or areas where additional details could enhance understanding.

Question:

What are some major environmental concerns today?

Response:

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

Feedback:

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

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

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

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

Question:

What are some major environmental concerns today?

Response:

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

Feedback:

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

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

Now refine the response based on the provided feedback.

Refined Response:

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

Ideally, if a strong LLM is adopted, we would like to have it perform all three steps without extra training. On the other hand, if we have enough labeled data for the task of interest, we can enhance the performance of the LLM using supervised learning. For example, we can fine-tune the LLM to better adapt it to refinement tasks, or alternatively, use task-specific models, which may not necessarily be based on LLMs [Welleck et al., 2023; Schick et al., 2023]. In a broader sense, improving LLMs for self-refinement tasks can be seen as an alignment issue. For example, it has been found that some self-correction abilities can be activated through RLHF [Ganguli et al., 2023]. However, discussing these issues is beyond the scope of this chapter. Further discussion can be found in Chapter 4.

In LLMs, self-refinement is related to several concepts that reveal the psychological aspects of these models, such as the ability to self-reflect. A view is that if LLMs are capable of self-reflection, their predictions can become more accurate and even possess self-correcting capabilities. This self-reflection can be activated in various ways, for example, by prompting these LLMs to engage in more in-depth and careful thinking, or by providing examples from which the models can learn and reflect. To illustrate, we consider here the **deliberate-then-generate (DTG)** method presented in Li et al. [2023a]’s work, where LLMs are prompted to deliberate. In DTG, we are given an initial model output which may contain errors. LLMs are then prompted to identify the error types of this model output and provide an improved output. Below is a template of DTG prompting for Chinese-to-English translation tasks.

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

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

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

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

In this example, the input translation is not generated by LLMs but is instead randomly sampled from the dataset. So it is simply an incorrect translation for the source sentence, and we can set the error type accordingly. The LLMs then generate a new translation by taking both the source sentence and the incorrect translation as input. The design of this prompt can also be considered as activating the learning capabilities of LLMs through “negative evidence” [Marcus, 1993], thereby enabling them to reflect and produce better outcomes through contrastive analysis. Nevertheless, this method does not rely on any feedback and can enhance the performance of a single LLM prediction via simple prompting.

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

errors in earlier steps may negatively impact subsequent problem-solving, and determining when to stop iterating often requires additional engineering effort.

3.2.4 Ensembling

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

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

Make this text simpler.

{*text*}

Condense and simplify this text.

{*text*}

Rewrite for easy reading.

{*text*}

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

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

$$\hat{\mathbf{y}} = \text{Combine}(\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_K) \quad (3.6)$$

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

$$\hat{y}_j = \arg \max_{y_j} \sum_{k=1}^K \log \Pr(y_j|\mathbf{x}_k, \hat{y}_1, \dots, \hat{y}_{j-1}) \quad (3.7)$$

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

$$\Pr(\mathbf{y}|p) = \int \Pr(\mathbf{y}|\mathbf{x}) \Pr(\mathbf{x}|p) d\mathbf{x} \quad (3.8)$$

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

While the Bayesian treatment is mathematically well-defined, it is common practice in NLP to assume a non-informative or uniform prior and focus instead on constructing a set of diverse prompts. Consequently, the output can be computed using a straightforward combination model, as described in Eq. (3.6). The issue of creating high-quality, diverse prompts has been studied in CoT and other in-context learning areas. Most of the research focuses on incorporating a variety of demonstration examples across different prompts. Here, we list some of these methods.

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

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

In addition to providing diverse prompts for LLMs, another approach is to make use of the inherent variance in the outputs of LLMs. One simple way to generate multiple outputs is to sample outputs from the hypothesis space. This is straightforward for LLMs because their search algorithms are naturally suited to producing multiple results. For example, if beam search is used, we can easily collect all the complete hypotheses in the beam to generate multiple output

candidates. Also, we can modify the search algorithms to implement more powerful sampling methods and explore a wider range of high-quality hypotheses.

The next question is how to combine diverse model predictions. This leads us back to the model combination issue as described in Eq. (3.6). Here we consider the **self-consistency** method, which outputs not the prediction with the highest probability, but rather the one that best aligns with other predictions [Wang et al., 2022a; 2023b]. First, an LLM is prompted with CoT as usual and generates multiple reasoning paths by sampling. Then, the frequency of each answer is counted across these reasoning paths. The final output is the answer that has the highest count. Below shows three different predictions by prompting an LLM with CoT.

Prediction 1 (Correct):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Each coin flip is an independent event with a 50% chance of landing heads (H) and 50% tails (T). Consider the possible outcomes for three flips: HHH, HHT, HTH, HTT, THH, THT, TTH, TTT. Out of these, only HHH, HHT, HTH, and THH are relevant. We are only interested in scenarios with exactly one head: HTT, THT, and TTH. There are 3 such outcomes out of 8 total, giving a probability of 3/8.

Prediction 2 (Correct):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Using a binomial probability formula, where p (success or heads) is 0.5, and each of the 3 trials (flips) is independent, we look for exactly one success (head). The formula for exactly k successes in n trials is given by the binomial coefficient formula $\binom{n}{k}p^k(1-p)^{n-k}$. Plugging in the values, we get $\binom{3}{1}(0.5)^1(0.5)^{3-1} = 3 \cdot 0.5 \cdot 0.25 = 0.375$.

Prediction 3 (Wrong):

Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?

A: Let's think step by step.

Since each coin flip has a 50% chance of landing heads and a 50% chance of landing tails, and we are looking for the case where only one of the three flips results in heads, we might initially think this is simply a matter of picking which flip is the heads. There are three flips, so one might incorrectly assume that the chance of picking one specific outcome like this would be 1 out of 3. Thus, they might conclude that the probability of exactly one head is $1/3 = 33.3\%$.

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

Self-consistency provides a criterion for determining the best prediction in a pool of candidates. Since the prompt and the model are fixed in this method, it is not strictly a prompt ensembling method. Instead, it can be seen as an instance of output ensembling methods, also known as hypothesis selection methods, which have long been explored in NLP, particularly for text generation problems [Xiao et al., 2013]. In these methods, multiple outputs are generated by varying model architectures or parameters. Each output is then assigned a score by some criterion, and the outputs are re-ranked based on these scores. There are various ways to define the scoring function, such as measuring the agreement between an output and others, and using a stronger model to rescore each output⁸. Figure 3.2 shows a comparison of different ensembling methods for LLMs.

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

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

$$\begin{aligned} \text{Risk}(\mathbf{y}) &= \mathbb{E}_{\mathbf{y}_r \sim \Pr(\mathbf{y}_r|\mathbf{x})} R(\mathbf{y}, \mathbf{y}_r) \\ &= \sum_{\mathbf{y}_r \in \Omega} R(\mathbf{y}, \mathbf{y}_r) \cdot \Pr(\mathbf{y}_r|\mathbf{x}) \end{aligned} \quad (3.9)$$

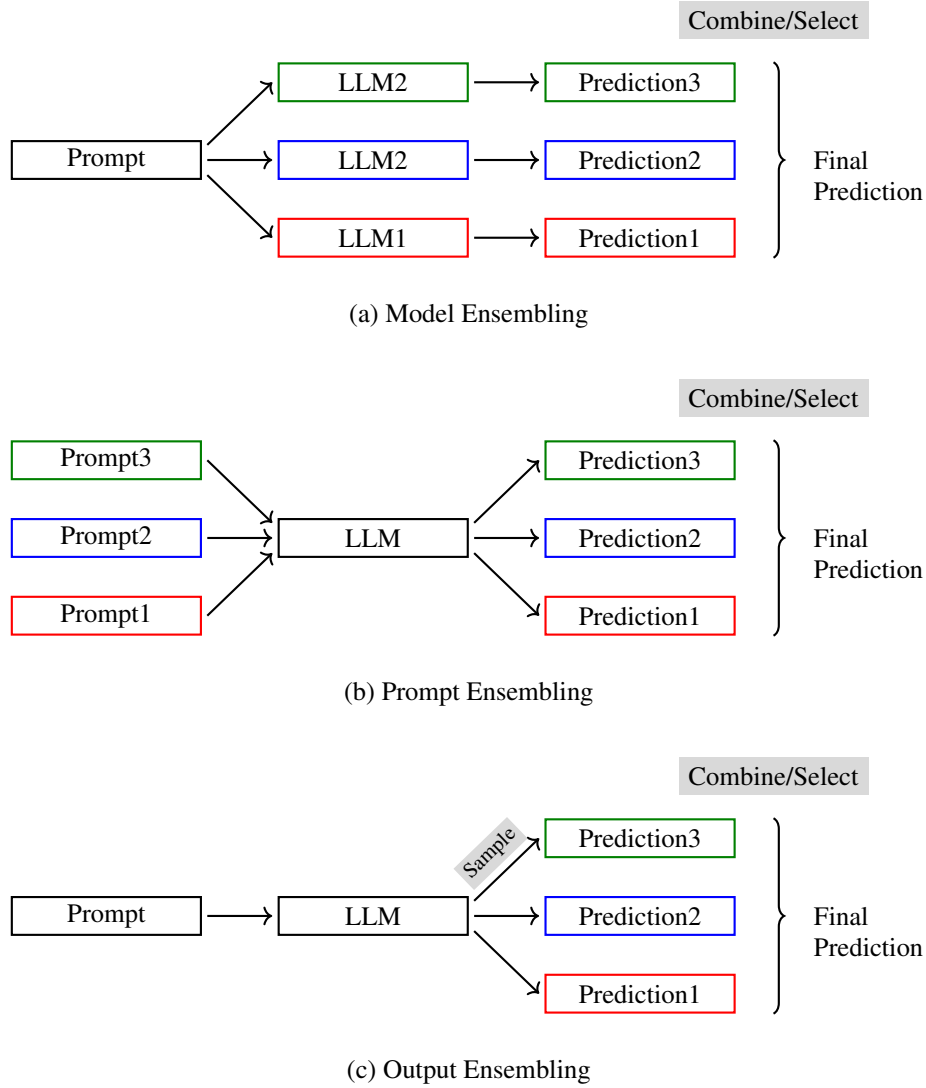


Fig. 3.2: Ensembling methods for LLMs. In standard model ensembling (a), multiple LLMs varying in architectures or parameters are used. Each LLM receives the same prompt and produces a prediction. These predictions are combined to generate the final prediction. In prompt ensembling (b), we have one LLM and multiple prompts. The LLM produces a prediction for each prompt, and these predictions are combined as usual. In output ensembling (c), the LLM samples multiple predictions over the prediction space given a prompt. It can be seen as a method to boost the performance of the LLM itself. Note that these ensembling methods can be combined to increase the diversity of predictions. For example, we can use both prompt ensembling and output ensembling to obtain more diverse predictions.

as correct or incorrect, although these verifiers are typically used as scoring models. Given a reasoning path for a problem, the verifiers can be used to score either the entire path (called outcome-based approaches) [Cobbe et al., 2021], or each individual reasoning step (called process-based approaches) [Uesato et al., 2022; Lightman et al., 2024].

3.2.5 RAG and Tool Use

RAG is generally employed when standard LLMs, which rely solely on pre-trained knowledge, lack accuracy and depth in the generated text. By drawing from external databases and documents,

RAG can significantly improve the quality of responses, ensuring they are both contextually relevant and factually correct. Such an approach is particularly useful in scenarios that require high factual accuracy and up-to-date information, such as complex question answering.

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

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

Steps 1 and 2 can be implemented by using an external information retrieval system. For example, we can store the collection of texts in a vector database and then retrieve the most similar texts through vector-based search techniques. Since information retrieval is not the focus of this chapter, we will assume that such systems are available off-the-shelf and use them directly.

Here we present how to prompt LLMs to make use of retrieved texts. To illustrate, consider an example of using LLMs to answer the following question.

Where will the 2028 Olympics be held?

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

(Wikipedia)

The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28, is an upcoming international multi-sport event scheduled to take place from July 14-30, 2028, in the United States. ...

(The Sporting News)

In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after hosting the Summer Games in 1932 and 1984. It will also be the first time the United States has hosted an Olympic Games since the 2002 Winter Games in Salt Lake City. ...

...

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

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.

Question:

Where will the 2028 Olympics be held?

Relevant Text 1:

The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28 ...

Relevant Text 2:

In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after ...

...

The 2028 Olympics will be held in Los Angeles.

This prompt assumes that the provided texts are relevant to the question and expects the LLM to generate a faithful response using these texts. However, the information retrieval system may sometimes provide irrelevant or incorrect texts, which may lead the LLM to produce an incorrect answer. One straightforward way to address this issue is to improve the accuracy of the information retrieval system. Nevertheless, as with most AI systems, errors may still occur. Therefore, it is also necessary to enhance the robustness of the LLM, so that it can make reasonable predictions even when the input is inaccurate. Below is a new prompt that enables the LLM to be more faithful to the facts, and allows it to choose not to answer questions when the information provided is inaccurate.

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.

Please note that your answers need to be as accurate as possible and faithful to the facts. If the information provided is insufficient for an accurate response, you may simply output "No answer!".

Question:

Where will the 2028 Olympics be held?

Relevant Text 1:

The 2024 Summer Olympics, officially the Games of the XXXIII Olympiad and branded as Paris 2024, were an international multi-sport event ...

...

No answer!

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

Both RAG and fine-tuning are common methods for adapting LLMs using task-specific data. Standard RAG is training-free and can be directly applied to LLMs. To further improve RAG, it

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

One reason we discuss RAG here is that it can be broadly regarded as an instance of the general problem decomposition framework (see Section 3.2.2). RAG divides problem-solving into two steps. In the first step, we collect relevant and supporting information for a given query from various knowledge sources. In the second step, we use LLMs to generate responses based on the collected information. If we extend the concept of problem decomposition further, we will find that many tasks requiring the use of external systems or tools can be treated as similar problems. One such example is tool use in LLMs. In many applications, LLMs need to employ external databases, APIs, and even simulation tools to generate accurate responses. For example, LLMs can access real-time data from financial markets to provide up-to-date investment advice or integrate with healthcare databases to offer personalized medical insights. This integration extends the capabilities of LLMs by allowing them to interact with, and in some contexts, influence or control external systems. Consequently, LLMs function more as **autonomous agents** rather than mere text generators [Franklin and Graesser, 1996].

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

Your task is to answer the following question. You may use external tools, such as web search, to assist you.

Question:

Where will the 2028 Olympics be held?

The information regarding this question is given as follows:

`{tool: web-search, query: "2028 Olympics"}`

So the answer is: Los Angeles

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

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

Problem:

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

Solution:

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

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

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

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

3.3 Learning to Prompt

So far in this chapter, we have considered several basic prompting strategies and various refinements to them. However, all the prompts we have discussed were designed manually. This leads to a number of problems: First, designing high-quality prompts is inherently difficult and requires substantial manual effort. For example, extensive experimentation with different prompts is often needed to identify the most effective ones. Since different LLMs may respond better to certain

types of prompts, developing universally effective prompts can be even more resource-intensive. Second, manual prompt design relies heavily on human expertise, which can limit the diversity of approaches and overlook potentially effective prompts that are not immediately obvious to humans. Third, prompts created by humans can be complex and redundant, leading to longer inputs for LLMs and higher computational costs.

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

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

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

3.3.1 Prompt Optimization

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

- **Prompt Search Space.** This defines all possible prompts that the algorithms can explore. For example, one can edit some seed prompts to generate a set of diverse candidate prompts.
- **Performance Estimation.** Once a prompt is chosen, it needs to be evaluated. For example, a straightforward way is to input it to an LLM and measure its performance on a validation set.
- **Search Strategy.** The search process is generally the same as that used in many AI systems. At each step, the system explores a set of promising prompts in the search space and

evaluates them. This process continues as more prompts are explored. The outcome of the search is the best-performing prompt observed until the search stops.

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

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

You are given a task to complete using LLMs. Please write a prompt to guide the LLMs.

{*task-description*}

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

You are provided with several input-output pairs for a task. Please write an instruction for performing this task.

Input: {*input1*} Output: {*output1*}

Input: {*input2*} Output: {*output2*}

...

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

- **Evaluation.** Once we obtain the candidate pool C , we need to evaluate the prompts in C . One method is to feed each prompt into an LLM and assess the results on the downstream

task. For example, we can evaluate the output of the LLM given an input using a pre-defined metric, or alternatively, use the log-likelihood of the output as a measure of the quality of the prompt.

- **Pruning.** If C contains a large number of prompts, it is reasonable to prune the unpromising prompts within it, thus reducing the computational burden in subsequent steps. This is a standard pruning problem. Given the evaluation score for each prompt, a simple method is to keep only a certain percentage of the prompts and discard the rest.
- **Expansion.** Expansion is a key operation in search algorithms used to explore different states in the search space. The expansion operation here can be defined as a function

$$C' = \text{Expand}(C, f) \quad (3.10)$$

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

Below is a prompt for an LLM. Please provide some new prompts to perform the same task.

Input: {*prompt*}

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

In prompt optimization, the expansion step plays a key role, as it defines how we explore the search space, and our goal is to find optimal results with minimal effort. One improvement to this step is to treat the problem as a paraphrasing task. A simple method is to apply off-the-shelf paraphrasing systems, either based on LLMs or other models, to transform input prompts into semantically equivalent forms [Jiang et al., 2020]. Alternatively, we can define specific edit operations, such as insertions and modifications, for each token. A given prompt can be edited into new prompts by applying these operations [Prasad et al., 2023]. Also, further evaluation and pruning can be applied to filter out low-quality prompts. In addition to framing prompt generation as a paraphrasing problem, we can improve the quality of prompts during expansion by learning from feedback [Pryzant et al., 2023]. This approach is somewhat related to the self-refinement issue discussed in Section 3.2.3. An LLM can be used to generate feedback on an input prompt, which is then revised based on this feedback. This feedback-and-revision cycle can be repeated multiple times until the result converges or the desired outcome is achieved.

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

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

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

3.3.2 Soft Prompts

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

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

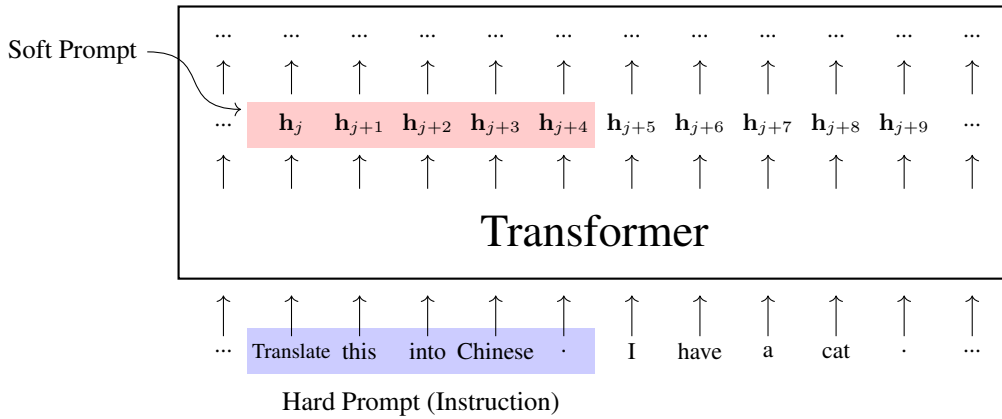


Fig. 3.3: Illustration of hard and soft prompts. Here the hard prompt is the instruction we input to the LLM for performing the task. The LLM encodes this instruction as usual, and the intermediate representations corresponding to the instruction can be viewed as some sort of soft prompt.

humans, soft prompts are encoded in a format that is more comprehensible to the model rather than to humans. To illustrate, consider a simple prompt

Translate the sentence into Chinese.
Consider it done!

Here, the instruction “Translate the sentence into Chinese” can be seen as a hard prompt, denoted by the token sequence $c_1 \dots c_5$. By feeding these tokens into an LLM, they are transformed into a sequence of real-valued vectors $h_1 \dots h_5$, each corresponding to a token. We can roughly think of $h_1 \dots h_5$ as a soft prompt, as illustrated in Figure 3.3.

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

3.3.2.1 Adapting LLMs with Less Prompting

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

models can respond correctly when dealing with similar prompts with those in fine-tuning.

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

$$\mathbf{x} = (\mathbf{c}, \mathbf{z}) \quad (3.11)$$

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

$$\begin{aligned} \hat{\theta} &= \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) \\ &= \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\mathbf{y}|\mathbf{c}, \mathbf{z}) \end{aligned} \quad (3.12)$$

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

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

Translate the following sentence from English to Chinese.

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

Translate this into Chinese.

Even, we can define the instruction as a single phrase

Translate!

With certain fine-tuning effort, we can adapt LLMs to follow any of these instructions. From an efficient prompting perspective, there are computational advantages in simplifying instructions in prompting. For example, we can use simple instructions like “Translate!” to perform tasks that would typically require more complex and detailed instructions. This can make subsequent

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

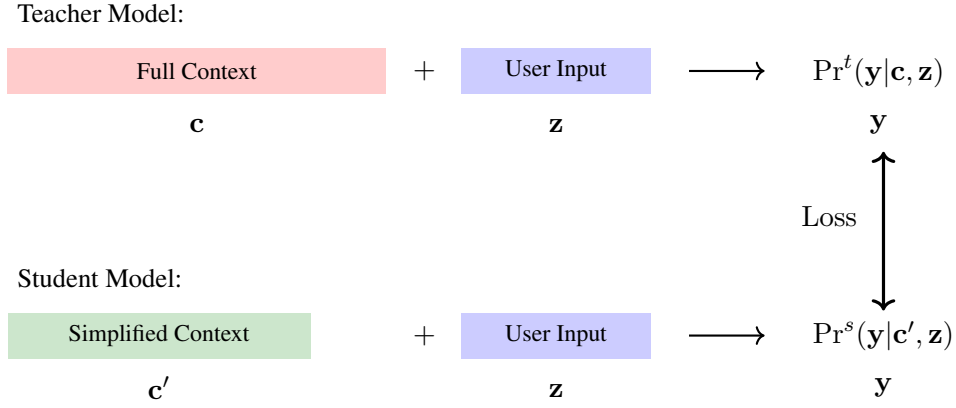


Fig. 3.4: Illustration of context distillation [Snell et al., 2022]. The teacher model is a standard LLM, which takes both the context and the user input as model input and produces a prediction as model output. Then, we simplify the context (e.g., simplifying the instruction in prompting) and use the student model to make predictions based on the simplified context and the user input. The student model is trained by minimizing the loss between the predictions produced by the two models.

prompting during inference much easier. On the other hand, fine-tuning LLMs with overly simplified instructions may be harmful to the generalization of the models. Since simplified instructions can lead to a loss of information, it is more likely that the LLMs will overfit the fine-tuning data and fail to generalize beyond those instructions. In scenarios involving both complex and simplified instructions for fine-tuning, this problem is more severe because the labeled data available for fine-tuning is usually limited, and accommodating a variety of instructions is costly.

An alternative way to adapt LLMs for simplified instructions is through knowledge distillation. As an example, we consider the context distillation method [Snell et al., 2022]. The goal of this method is to learn a student model that can make use of simplified instructions from a well-trained instruction-following teacher model. Figure 3.4 shows an illustration of this approach. Building the teacher model follows a standard fine-tuning process: we first collect a certain amount of data that includes instructions, user inputs, and correct responses, and then we continue to train a pre-trained model with this dataset. For building the student model, we need to construct a new dataset \mathcal{D}' where each sample is a tuple consisting of an instruction, a corresponding simplified instruction, and a user input, denoted by $\mathbf{x}' = (c, c', z)$. Knowledge distillation is performed by minimizing a loss function defined on the outputs of the teacher and student models

$$\hat{\theta} = \arg \min_{\theta} \sum_{\mathbf{x}' \in \mathcal{D}'} \text{Loss}(\Pr^t(\cdot|\cdot), \Pr_{\theta}^s(\cdot|\cdot), \mathbf{x}') \quad (3.13)$$

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

$$\text{Loss} = \sum_{\mathbf{y}} \Pr^t(\mathbf{y}|\mathbf{c}, \mathbf{z}) \log \Pr_{\theta}^s(\mathbf{y}|\mathbf{c}', \mathbf{z}) \quad (3.14)$$

But this function is computationally infeasible because it requires summing over an exponentially large number of outputs. A variant of this method is to train the student model using outputs generated by the teacher model. For each sample, we use the teacher model to produce an output

$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \log \Pr^t(\mathbf{y}|\mathbf{c}, \mathbf{z})$. Then we consider $\hat{\mathbf{y}}$ as the target for learning, and the loss function is given by

$$\text{Loss} = \log \Pr_{\theta}^s(\hat{\mathbf{y}}|\mathbf{c}', \mathbf{z}) \quad (3.15)$$

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

$$\text{Loss} = \text{KL}(\mathbf{P}^t \parallel \mathbf{P}_{\theta}^s) \quad (3.16)$$

where

$$\mathbf{P}^t = \Pr^t(\cdot|\mathbf{c}, \mathbf{z}) \quad (3.17)$$

$$\mathbf{P}_{\theta}^s = \Pr_{\theta}^s(\cdot|\mathbf{c}', \mathbf{z}) \quad (3.18)$$

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

3.3.2.2 Learning Soft Prompts for Parameter-efficient Fine-tuning

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

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

Specifically, let the input of a layer at depth l be denoted by $\mathbf{H}^l = \mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l$. The output of the layer can be expressed as

$$\mathbf{H}^{l+1} = \text{Layer}(\mathbf{H}^l) \quad (3.19)$$

In prefix fine-tuning, we extend the sequence $\mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l$ by adding a few vectors at the beginning, which we denote as $\mathbf{p}_0^l \mathbf{p}_1^l \dots \mathbf{p}_n^l$. Hence \mathbf{H}^l can be written in the form

$$\mathbf{H}^l = \underbrace{\mathbf{p}_0^l \mathbf{p}_1^l \dots \mathbf{p}_n^l}_{\text{trainable}} \underbrace{\mathbf{h}_0^l \mathbf{h}_1^l \dots \mathbf{h}_m^l}_{\text{previous layer output}} \quad (3.20)$$

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

$$\begin{aligned}\bar{\mathbf{H}}^{l+1} &= \text{Layer}(\mathbf{H}^l)[-m-1:] \\ &= \mathbf{h}_0^{l+1} \mathbf{h}_1^{l+1} \dots \mathbf{h}_m^{l+1}\end{aligned}\quad (3.21)$$

where $[-m-1:]$ denotes the slicing operation that extracts the last $m + 1$ elements of a sequence. Given $\bar{\mathbf{H}}^{l+1}$, the input of the next layer can be expressed in the same form of Eq. (3.20):

$$\begin{aligned}\mathbf{H}^{l+1} &= \mathbf{p}_0^{l+1} \mathbf{p}_1^{l+1} \dots \mathbf{p}_n^{l+1} \bar{\mathbf{H}}^{l+1} \\ &= \mathbf{p}_0^{l+1} \mathbf{p}_1^{l+1} \dots \mathbf{p}_n^{l+1} \mathbf{h}_0^{l+1} \mathbf{h}_1^{l+1} \dots \mathbf{h}_m^{l+1}\end{aligned}\quad (3.22)$$

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

Figure 3.5 shows an illustration of prefix fine-tuning for a translation task. Here, only the prefix vectors \mathbf{p}_0^l and \mathbf{p}_1^l are updated by receiving the error gradients from the output (i.e., the Chinese translation). By adjusting these vectors for the translation task, the model adapts accordingly. This makes \mathbf{p}_0^l and \mathbf{p}_1^l serve as prompts which activate the LLM to perform the task without needing explicit input prompts like “Translate the following sentence from English to Chinese”. At test time, we prepend the optimized \mathbf{p}_0^l and \mathbf{p}_1^l to the layer, and the LLM will then translate the input sentence. Note that prefix fine-tuning introduces additional $L \times n \times d$ parameters, where L is the number of layers, n is the number of prefixes, and d is the dimensionality of each prefix. However, this number is much smaller compared to the total number of parameters in the LLM, making the fine-tuning process highly efficient.

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

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

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

Note that a pseudo embedding needs not to correspond to any token in natural language. Instead these embeddings can be seen as “soft prompt embeddings” that serve to condition the LLMs. By training soft prompt embeddings on task-specific data, they learn to interact adaptively with the token embeddings $\mathbf{e}_0 \dots \mathbf{e}_m$ and guide the behavior of LLMs. Since prompt tuning does not change the underlying parameters of pre-trained LLMs, it is considered a lightweight and efficient method of fine-tuning, improving task-specific performance while maintaining their generalization capabilities. See Figure 3.6 for an illustration of prompt tuning.

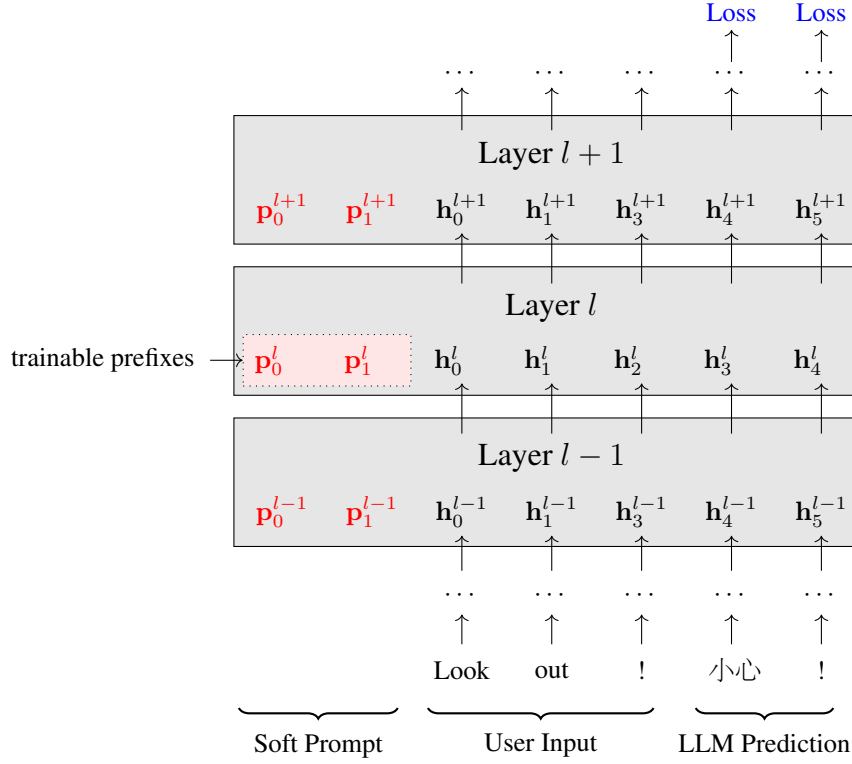
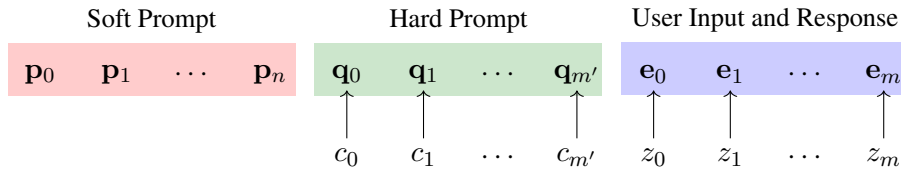


Fig. 3.5: Illustration of prefix fine-tuning for a translation task (Look out! \rightarrow 小心!). For each layer, we add two prefixes p_0^l and p_1^l at the beginning. The LLM is trained to minimize the loss on the predictions given the input. During this process, only the prefixes are optimized while the rest of the parameters remain fixed. Therefore, the model can adapt to the given task in a very efficient manner. At inference time, the LLM works with optimized prefixes, and can perform the task without the need of explicit hard prompts.

Since $p_0 p_1 \dots p_n$ is itself a sequence, we can employ sequence models to better represent it. For example, a Transformer model can encode this sequence, and the resulting representation can then be used as the input to the LLM. In other words, we can develop an additional model for encoding soft prompts. Another way to improve prompting is by combining soft and hard prompts, thereby taking advantage of both types [Liu et al., 2023b]. In the embedding sequence, we can arrange or intersperse these prompts. This would result in different prompt patterns. For example, a simple pattern that uses both two types of prompt is



where $c_0 \dots c_{m'}$ denotes the hard prompt and $q_0 \dots q_{m'}$ denotes the corresponding embedding sequence.

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

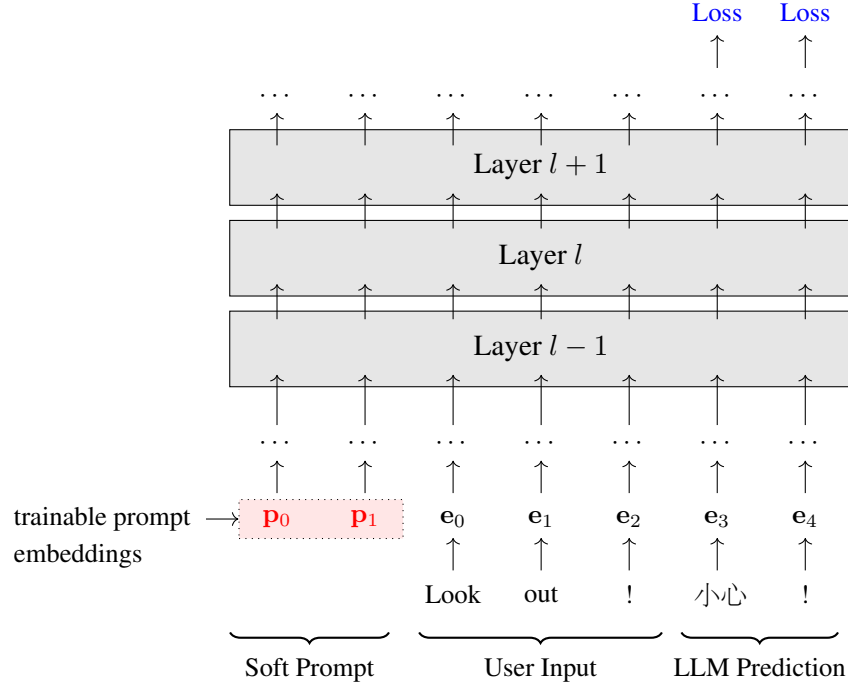


Fig. 3.6: Illustration of prompt tuning for a translation task (Look out! \rightarrow 小心!). Instead of using fixed textual prompts, soft prompts are learnable embeddings that are added at the beginning of the embedding sequence. During fine-tuning, only these prompt embeddings are optimized to efficiently adapt the LLM to the given task. Once optimized, the prompt embeddings are used to instruct the LLM to perform the task as new data arrives.

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

Broadly speaking, many parameter-efficient fine-tuning methods can be thought of as learning some sort of soft prompt [Lialin et al., 2023]. When we fine-tune a part of an LLM for a task, this process can essentially be seen as injecting task-related prompting information into that specific part of the model. Another widely-used approach to parameter-efficient fine-tuning is to add an adaptor layer between the existing model layers. This approach allows us to fine-tune only the adaptor layer on specific tasks without altering the underlying architecture or retraining the entire model. In this sense, adaptor layers can be viewed as soft prompts that encode prompting and task-related information and interact with the original LLM to help it adapt. To summarize, Figure 3.7 shows a comparison of different methods of using soft prompts in LLMs.

3.3.2.3 Learning Soft Prompts with Compression

Another approach to learning soft prompts is from the perspective of compression. As a simple example, consider the problem of approximating a long context using a continuous representation [Wingate et al., 2022]. Suppose we have a user input z and its context c (such as long instructions and demonstrations). Now we want to develop a compressed representation of the context, denoted

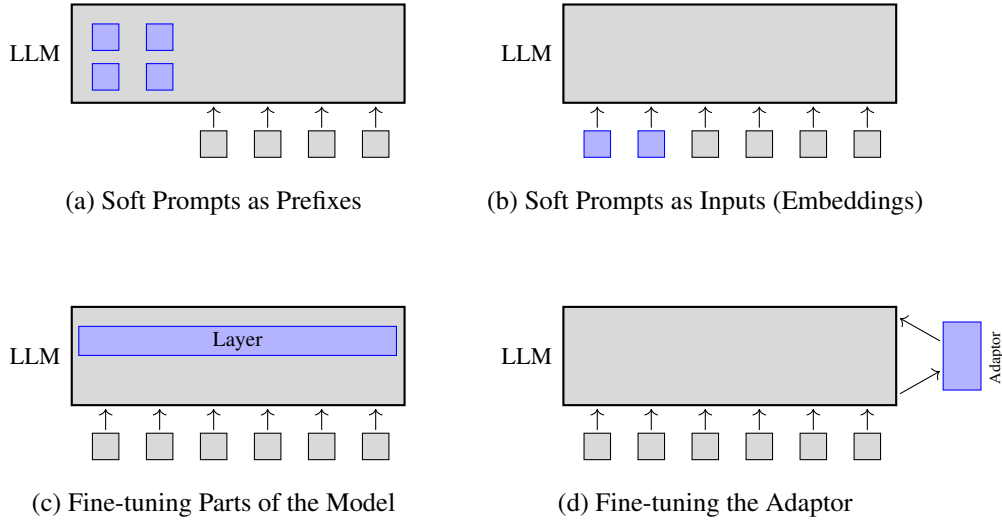


Fig. 3.7: Illustrations of using soft prompts in LLMs. Here tunable soft prompts are shown in blue, and components whose parameters are fixed during fine-tuning are shown in gray. In sub-figure (a), soft prompts are prefixes appended to each layer of the LLM. In sub-figure (b), soft prompts are used as input embeddings for the LLM. In sub-figures (c) and (d), soft prompts are broadly treated as components of the model that are fine-tuned for task adaptation.

by σ , such that the prediction based on \mathbf{z} and σ is as close as possible to the prediction based on \mathbf{z} and \mathbf{c} . This goal can be expressed in the form

$$\hat{\sigma} = \arg \min_{\sigma} s(\hat{\mathbf{y}}, \hat{\mathbf{y}}_{\sigma}) \quad (3.23)$$

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

One general framework for achieving this is knowledge distillation, where $\hat{\mathbf{y}}$ and $\hat{\mathbf{y}}_{\sigma}$ can be seen as the predictions of the teacher model and the student model, respectively. This formalization links our discussion to the context distillation problem discussed earlier. The training objective can be obtained by analogy with Eqs. (3.15) and (3.16). For example, a simple training objective is given by

$$\hat{\sigma} = \arg \max_{\sigma} \log \Pr(\hat{\mathbf{y}}|\sigma, \mathbf{z}) \quad (3.24)$$

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

$$\hat{\sigma} = \arg \min_{\sigma} \text{KL}(\Pr(\cdot|\mathbf{c}, \mathbf{z}) \parallel \Pr(\cdot|\sigma, \mathbf{z})) \quad (3.25)$$

The difference with the models in Eqs. (3.15) and (3.16) is that here the compressed context is represented as real-valued vectors (call them **prompt embeddings**), rather than as normal tokens. By applying the above methods, we distill the context from the token sequence \mathbf{c} into the embeddings σ . Note that the teacher model $\Pr(\cdot|\mathbf{c}, \mathbf{z})$ and the student model $\Pr(\cdot|\sigma, \mathbf{z})$ may not share the same architecture or model settings. In practice, we generally wish for the teacher model to be

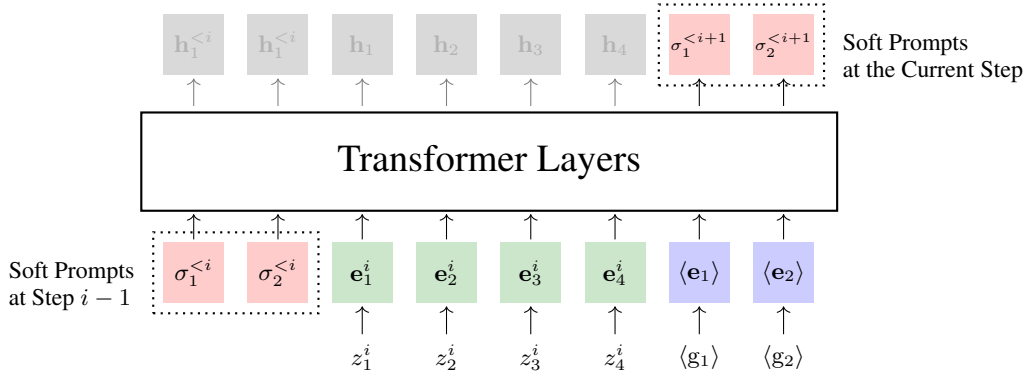


Fig. 3.8: Illustration of compressing a context segment into soft prompts ($\kappa = 2$ and $m_i = 4$). The input to the LLM includes the soft prompts from the previous step ($\sigma_1^{<i>}$ and $\sigma_2^{<i>}$), the tokens of the segment (z_1, z_2, z_3 , and z_4), and the summary tokens ($\langle g_1 \rangle$ and $\langle g_2 \rangle$). Given these, the LLM operates as usual. We then extract the outputs at the last Transformer layer that correspond to the summary tokens. These outputs can be viewed as the soft prompts that accumulated up to this segment.

stronger, while the student model should be smaller and more efficient.

While compressing full context into continuous representations is a straightforward approach to learning soft prompts, it requires a teacher model that can deal with long input sequences. In many cases, however, the context is so long that applying an LLM is too costly or infeasible. Modeling long input sequences can fall under the broad family of efficient methods for long-context LLMs. Many techniques have been developed to address this issue. For example, one can use a fixed-size KV cache to store the past information at each step during inference. Efficient Transformer architectures and long-context LLMs have been intensively discussed in this book. For more detailed discussions of these topics, interested readers can refer to Chapter 2.

There are also methods specifically designed to compress long context into soft prompts. Here we consider [Chevalier et al. \[2023\]](#)’s method as an example. The basic idea is that we learn soft prompts gradually by accumulating the fixed-size context representation over the context sequence. Given a long context, we first divide it into a number of segments $\mathbf{z}^1, \dots, \mathbf{z}^K$. We then process these segments in sequence, each time generating a representation of the context we have processed so far, denoted by $\sigma^{<i+1>}$. To do this, a few summary tokens $\langle g_1 \rangle, \dots, \langle g_\kappa \rangle$ are introduced. At each step, we take a segment $\mathbf{z}^i = z_1^i \dots z_{m_i}^i$, along with the previous context representation $\sigma^{<i>}$ and the summary tokens $\langle g_1 \rangle, \dots, \langle g_\kappa \rangle$ as input, and use an LLM to produce the corresponding hidden representation sequence at the last Transformer layer. An example of this process is illustrated in Figure 3.8.

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

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

3.3.3 Prompt Length Reduction

While soft prompts provide dense, hidden representations, they are not directly interpretable. The lack of interpretability can be a significant barrier for users trying to understand how their inputs influence LLM outputs. Moreover, although soft prompts are efficient for fine-tuning and deployment, they are inflexible and do not allow for easy adjustments without extensive fine-tuning or modification. This inflexibility can limit their utility in dynamic environments where prompt changes are frequently needed.

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

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

Question:

What are the best practices for using artificial intelligence in diagnosing cardiovascular diseases?

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

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

We can also paraphrase it as a shorter text.

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

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

LLMs have been fine-tuned and aligned to perform text simplification tasks, it is straightforward to use these models to simplify prompts. For example, we can prompt an LLM to simplify a text under certain constraints, such as limiting the length of the simplified text.

3.4 Summary

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

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

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

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

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

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

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

CHAPTER 4

Alignment

Alignment is not a new concept in NLP, but its meaning varies across different domains and over time. In traditional NLP, the term *alignment* typically refers to the tasks that link corresponding elements in two sets, such as aligning words between a Chinese sentence and an English sentence. As LLMs become increasingly important in NLP research, this term is more broadly used to refer to aligning model outputs with human expectations. The problem that alignment addresses is that the output of a model may not align with the specific goals or contexts intended by users. For example, pre-trained LLMs may not be able to follow user instructions because they were not trained to do so. Another example is that LLMs may generate harmful content or perpetuate biases inherent in their training data. This poses new challenges in ensuring that LLM outputs are not only accurate and relevant, but also ethically sound and non-discriminatory.

Simply pre-training LLMs can result in a variety of alignment problems. Our ultimate goal is to resolve or mitigate all these problems to ensure LLMs are both accurate and safe. There is an interesting issue here: since large language models are trained on vast amounts of data, we have reason to believe that if we have sufficient data covering a variety of tasks and aligned with human preferences, pre-training could make LLMs accurate and safe enough, perhaps even eliminating the need for alignment. However, the reality is that it is nearly impossible to gather data that encompasses all tasks or adequately represents human preferences. This makes it difficult to achieve model alignment through pre-training alone, or at least, at this stage, alignment remains a very necessary and critical step in the development of LLMs.

In this chapter, we will focus on alignment methods for LLMs. We will begin by discussing the general alignment tasks. Then we will consider two widely-used approaches, known as **instruction alignment** and **human preference alignment**, respectively. The former resorts to supervised fine-tuning techniques and guides the LLMs to generate outputs that adhere closely to user instructions. On the other hand, the latter typically relies on reinforcement learning techniques, where the LLMs are trained based on feedback from humans. While these methods are motivated by different goals, they are commonly used together to develop well-aligned LLMs.

4.1 An Overview of LLM Alignment

Alignment can be achieved in several different ways. We need different methods for LLM alignment because this problem is itself complicated and multifaceted, requiring a blend of technical considerations. Here we consider three widely-used approaches to aligning LLMs.

The first approach is to fine-tune LLMs with labeled data. This approach is straightforward as it simply extends the pre-existing training of a pre-trained LLM to adapt it to specific tasks. An example of this is **supervised fine-tuning (SFT)**, in which the LLM is further trained on a dataset comprising task-specific instructions paired with their expected outputs. The SFT dataset is generally much smaller compared to the original training set, but this data is highly specialized. The result of SFT is that the LLM can learn to execute tasks based on user instructions. For example, by fine-tuning the LLM with a set of question-answer pairs, the model can respond to specific questions, even if not directly covered in the SFT dataset. This method proves particularly

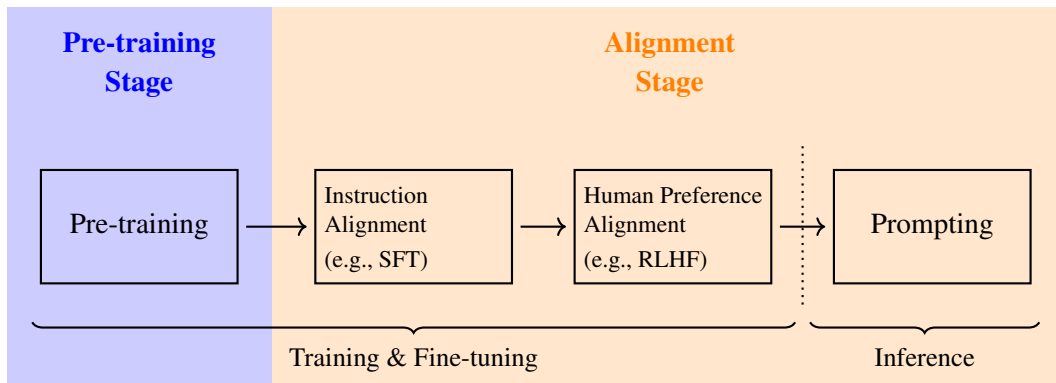


Fig. 4.1: Schematic illustration of the pre-train-then-align method for developing LLMs. In the pre-training stage, we train an LLM on vast amounts of data using next token prediction. Then, in the alignment stage, we align the LLM to user instructions, intents, and preferences. This includes instruction alignment, human preference alignment, and prompting.

useful when it is relatively easy to describe the input-output relationships and straightforward to annotate the data.

The second approach is to fine-tune LLMs using reward models. One difficulty in alignment is that human values and expectations are complex and hard to describe. In many cases, even for humans themselves, articulating what is ethically correct or culturally appropriate can be challenging. As a result, collecting or annotating fine-tuning data is not as straightforward as it is with SFT. Moreover, aligning LLMs is not just a task of fitting data, or in other words, the limited samples annotated by humans are often insufficient to comprehensively describe these behaviors. What we really need here is to teach the model how to determine which outputs are more in line with human preferences, for example, we not only want the outputs to be technically accurate but also to align with human expectations and values. One idea is to develop a reward model analogous to a human expert. This reward model would work by rewarding the LLM whenever it generates responses that align more closely with human preferences, much like how a teacher provides feedback to a student. To obtain such a reward model, we can train a scoring function from human preference data. The trained reward model is then used as a guide to adjust and refine the LLM. This frames the LLM alignment task as a reinforcement learning task. The resulting methods, such as **reinforcement learning from human feedback (RLHF)**, have been demonstrated to be particularly successful in adapting LLMs to follow the subtleties of human behavior and social norms.

The third approach is to perform alignment during inference rather than during training or fine-tuning. From this perspective, prompting in LLMs can also be seen as a form of alignment, but it does not involve training or fine-tuning. So we can dynamically adapt an LLM to various tasks at minimal cost. Another method to do alignment at inference time is to rescore the outputs of an LLM. For example, we could develop a scoring system to simulate human feedback on the outputs of the LLM (like a reward model) and prioritize those that receive more positive feedback.

The three methods mentioned above are typically used in sequence once the pre-training is complete: we first perform SFT, then RLHF, and then prompt the LLM in some way during inference. This roughly divides the development of LLMs into two stages — the pre-training stage and the alignment stage. Figure 4.1 shows an illustration of this. Since prompting techniques have been intensively discussed in the previous chapter, we will focus on fine-tuning-based alignment methods in the rest of this chapter.

4.2 Instruction Alignment

One feature of LLMs is that they can follow the prompts provided by users to perform various tasks. In many applications, a prompt consists of a simple instruction and user input, and we want the LLM to follow this instruction to perform the task correctly. This ability of LLMs is also called the instruction-following ability. For example, below is a prompt where we want the LLM to extract key points and provide a concise summary for a lengthy article.

Instruction

Summarize this text in three sentences.

Input

Daylight Savings Time (DST) - the process of moving clocks forward by one hour in the summer - was started in Germany in 1916 ...

Output

This task requires the LLM to understand the instruction “Summarize this text in three sentences” and perform the summarization accordingly. However, LLMs are typically trained for next-token prediction rather than for generating outputs that follow instructions. Applying a pre-trained LLM to the above example would likely result in the model continuing to write the input article instead of summarizing the main points. The goal of instruction alignment (or **instruction fine-tuning**) is to tune the LLM to accurately respond to user instructions and intentions. The rest of this section will discuss some issues related to instruction alignment, including fine-tuning LLMs to follow instructions, generating or collecting instruction data, and generalizing instruction alignment.

4.2.1 Supervised Fine-tuning

One straightforward approach to adapting LLMs to follow instructions is to fine-tune these models using annotated input-output pairs [Ouyang et al., 2022; Wei et al., 2022a]. Unlike standard language model training, here we do not wish to maximize the probability of generating a complete sequence, but rather maximize the probability of generating the rest of the sequence given its prefix. This approach makes instruction fine-tuning a bit different from pre-training. The SFT data is a collection of such input-output pairs (denoted by S), where each output is the correct response for the corresponding input instruction. For example, below is an SFT dataset

x (instruction + user input)	y (output)
Summarize the following article. Article: In recent years, solar energy has seen unprecedented growth, becoming the fastest-growing ...	{*summary*}
Extract the main financial figures from the following earnings report. Report: The company reported a revenue of \$10 million in the first quarter with a profit margin of 15% ...	Revenue: \$10 million, Profit Margin: 15%
Classify the following email as spam or not spam. Text: Congratulations! You’ve won a \$500 gift card. Click here to claim now.	Spam
Provide a solution to the following technical issue. Issue: my computer is running slow and often freezes.	First, check for ...

where the instructions are highlighted. This dataset contains instructions and the corresponding outputs for several different NLP problems, and so we can fine-tune an LLM to handle multiple tasks simultaneously.

Let $\mathbf{x} = x_0 \dots x_m$ be an input sequence (e.g., instruction + user input) and $\mathbf{y} = y_1 \dots y_n$ be the corresponding output sequence. In SFT, we aim to maximize the probability of the output \mathbf{y} given the input \mathbf{x} . Consider an LLM with pre-trained parameters $\hat{\theta}$. The fine-tuning objective can then be formulated as:

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x}) \quad (4.1)$$

where $\tilde{\theta}$ denotes the parameters optimized via fine-tuning, and $\hat{\theta}^+$ represents an adjustment to $\hat{\theta}$. Here we will omit the superscript $+$ and use θ to represent $\hat{\theta}^+$ to keep the notation uncluttered. But the reader should keep in mind that the fine-tuning starts from the pre-trained parameters rather than randomly initialized parameters.

The objective function $\log \Pr_{\theta}(y_i|\mathbf{x}, \mathbf{y}_{<i})$ is computed by summing the log-probabilities of the tokens in \mathbf{y} , conditional on the input \mathbf{x} and all the previous tokens $\mathbf{y}_{<i}$:

$$\log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr_{\theta}(y_i|\mathbf{x}, \mathbf{y}_{<i}) \quad (4.2)$$

This formulation is equivalent to minimizing the cross-entropy loss.

Note that minimizing the conditional log-probability $\log \Pr_{\theta}(\mathbf{y}|\mathbf{x})$ is not a standard language model training problem. If we concatenate \mathbf{x} and \mathbf{y} as a single sequence, a more general form of language modeling is based on the joint log-probability $\log \Pr_{\theta}(\mathbf{x}, \mathbf{y})$, that is, we minimize the loss over all tokens of the sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$. We can write the probability of this sequence using the chain rule

$$\begin{aligned} \log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}}) &= \log \Pr_{\theta}(\mathbf{x}, \mathbf{y}) \\ &= \underbrace{\log \Pr_{\theta}(\mathbf{x})}_{\text{set to 0}} + \underbrace{\log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}_{\text{loss computation}} \end{aligned} \quad (4.3)$$

There are two terms on the right-hand side of the equation. We can simply set the first term $\log \Pr_{\theta}(\mathbf{x})$ to 0, focusing solely on the second term $\log \Pr_{\theta}(\mathbf{y}|\mathbf{x})$ for loss computation. As a result, the training can be implemented using standard LLMs. For the sequence $\text{seq}_{\mathbf{x}, \mathbf{y}}$, we first run the forward pass as usual. Then, during the backward pass, we force the loss corresponding to \mathbf{x} to be zero. Figure 4.2 shows an illustration of this process.

By taking $\log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}})$ as the objective function, we can describe SFT using a regular form of language model training:

$$\tilde{\theta} = \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \Pr_{\theta}(\text{seq}_{\mathbf{x}, \mathbf{y}}) \quad (4.4)$$

The problem we considered above is fundamentally a **single-round prediction** problem, where the LLM generates a response based on a single input without any further interaction or feedback from the user. The input is processed, and the output is generated in one go. This is typical in

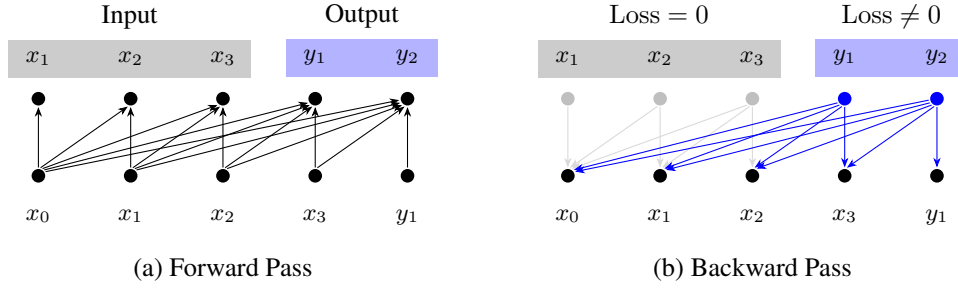


Fig. 4.2: Illustration of supervised fine-tuning for LLMs. We concatenate the input and the output into a single sequence. During the forward pass, we run the LLM as usual. During the backward pass, we compute the loss only for the output part and simply set the loss for the input part to 0.

scenarios where a single question is asked, and a single answer is provided, with no follow-up questions or clarifications. However, in practice, we sometimes have to handle multi-round prediction problems, for example, an LLM engages in a dialogue over multiple turns. In this setting, the LLM not only generates responses based on the initial input but also incorporates subsequent inputs that might refine or expand on earlier interactions. For example, we can use the LLM to act as a healthcare assistant chatbot and have a conversation with the user, like this

User I've been feeling very tired lately.

Chatbot I'm sorry to hear that. Besides feeling tired, have you noticed any other symptoms?

User Yes, I'm also experiencing headaches frequently.

Chatbot How long have these symptoms been going on?

User About a week now.

Chatbot It might be good to check in with a healthcare professional. Would you like help setting up an appointment?

User Yes, please. Can it be after work hours?

Chatbot Sure, I can arrange that. There are slots available next Wednesday and Thursday after 5 PM. Which day works better for you?

...

In this task, there are several rounds of conversation, each involving the generation of a response based on the user's request or question and the conversational history. Suppose we have K rounds of conversation, denoted by $\{\mathbf{x}^1, \mathbf{y}^1, \mathbf{x}^2, \mathbf{y}^2, \dots, \mathbf{x}^K, \mathbf{y}^K\}$. Here \mathbf{x}^k and \mathbf{y}^k denote the user request and the response, respectively, for each round k . The log-probability of generating the response can be written as $\log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^k)$. Our goal is then to maximize the sum of these log-probabilities

$$\tilde{\theta} = \arg \max_{\theta} \sum_{k=1}^K \log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^k) \quad (4.5)$$

A straightforward implementation of this involves calculating the conditional probability for each k . However, it requires running the LLM K times, each time with an increased conversational history to make predictions. A more efficient method is to perform loss computation of all responses in a single run of the LLM. To do this, we represent the conversation as a sequence $\text{seq}_{\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K} = [\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K]$ (or seq for short). The log-probability of this sequence is given by

$$\begin{aligned}
 \log \Pr_{\theta}(\text{seq}) &= \log \Pr_{\theta}(\mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K, \mathbf{y}^K) \\
 &= \underbrace{\log \Pr_{\theta}(\mathbf{x}^1)}_{\text{set to 0}} + \underbrace{\log \Pr_{\theta}(\mathbf{y}^1 | \mathbf{x}^1)}_{\text{loss computation}} + \dots + \\
 &\quad \underbrace{\log \Pr_{\theta}(\mathbf{x}^K | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{y}^{K-1})}_{\text{set to 0}} + \\
 &\quad \underbrace{\log \Pr_{\theta}(\mathbf{y}^K | \mathbf{x}^1, \mathbf{y}^1, \dots, \mathbf{x}^K)}_{\text{loss computation}}
 \end{aligned} \tag{4.6}$$

The trick here is that we ignore the loss for generating user inputs, as illustrated in Figure 4.3. Hence we only compute the probabilities of generating the responses given their conversational histories, in other words, the value on the right-hand side of Eq. (4.6) is actually equal to the value on the right-hand side of Eq. (4.5). As with Eq. (4.4), the training of this multi-round prediction model can be achieved by maximizing the log likelihood over a training dataset \mathcal{D} :

$$\tilde{\theta} = \arg \max_{\theta} \sum_{\text{seq} \in \mathcal{D}} \log \Pr_{\theta}(\text{seq}) \tag{4.7}$$

While implementing the SFT methods introduced above seems trivial as they are fundamentally the same as regular language model training, there are still issues that need to be considered in practice. For example,

- SFT requires labeled data. This makes SFT quite different from pre-training, where raw text is used as training data and is readily available. As in other supervised machine learning problems, data annotation and selection in SFT are not simple tasks. In general, we wish to develop SFT data that is both substantial in quantity and high in quality, and this data should be highly relevant to the tasks the LLM will perform. On the other hand, there is a need to fine-tune LLMs with less data to minimize computational and data construction costs. Often, the quality of LLMs is highly dependent on the data used in SFT. Thus, such data must be carefully developed and examined. As we will see in later subsections, SFT can be more efficient and effective through more advanced techniques for data construction.
- SFT is still computationally expensive for LLMs due to their large size. As a result, maintaining and updating such models is resource-intensive. For example, applying gradient updates to billions of parameters within an LLM requires significant computational power and memory. This often requires high-performance computing environments, which are costly to operate. To address these challenges, various optimization strategies, such as pruning, quantization, and the use of more efficient training algorithms, have been explored. In particular, there has been significant interest in parameter-efficient fine-tuning methods which are designed to maintain state-of-the-art performance without the need for extensive computational resources. We have seen in Chapter 3 that applying techniques like soft prompts can

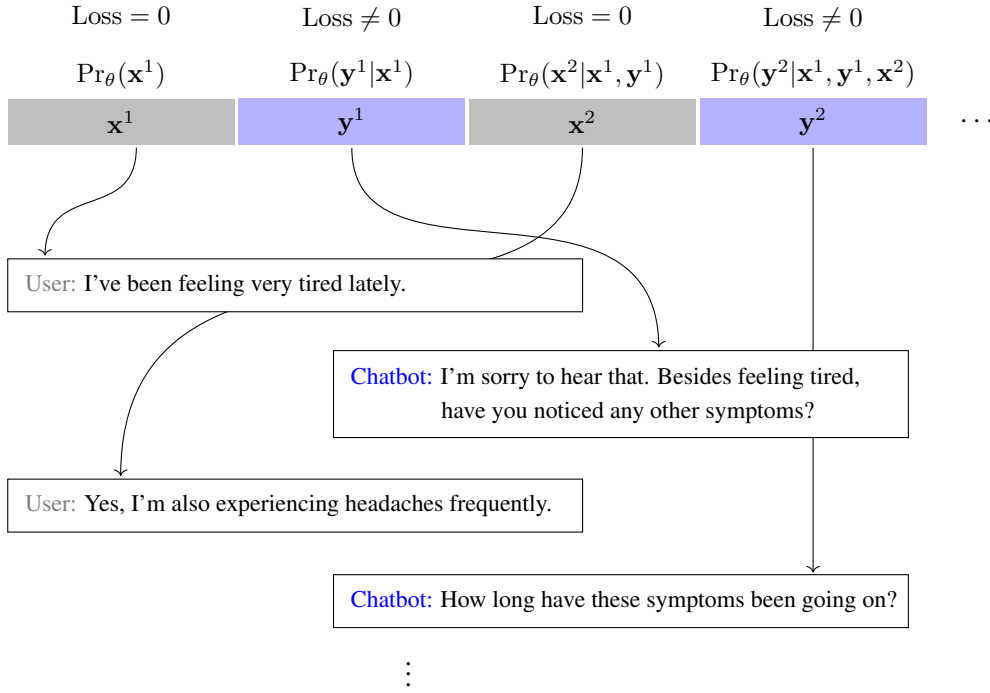


Fig. 4.3: Illustration of supervised fine-tuning for conversational models. Here the LLM acts as a chatbot to respond to each request based on the conversational history. The conversation progresses by alternating between the user and the chatbot. In SFT, we treat the entire conversation as a sequence, just like in standard LLMs, but compute the loss only for the responses of the LLM.

make the fine-tuning process more efficient. For further discussion on parameter-efficient methods, the reader can refer to related papers on this issue [Houlsby et al., 2019; Hu et al., 2022; Han et al., 2024].

- SFT can be regarded as a post-training step following pre-training. It is a separate training phase designed to preserve the advantages of the initial pre-training while incorporating new adjustments. This may seem paradoxical because updating a pre-trained LLM with further data potentially causes the model to forget some of its prior knowledge. Imagine a scenario where we have a large amount of SFT data and extensively fine-tune the LLM. In this case, the LLM could overfit the data, which in turn may reduce generalization performance or cause catastrophic forgetting. A common strategy to mitigate this issue is to employ regularization and early stopping techniques. Another practical approach is to use a smaller learning rate to gently adjust the weights of the LLM. In addition, fine-tuning with data from diverse sources and problem domains can also be beneficial. Nevertheless, in practice, the SFT step is often carefully examined and requires substantial engineering and experimental efforts to optimize.

4.2.2 Fine-tuning Data Acquisition

Fine-tuning data is so important that much recent work in LLM has focused on developing various datasets for instruction fine-tuning. As with most work in machine learning, there are generally two approaches to data acquisition — manual data generation and automatic data generation.

4.2.2.1 Manually Generated Data

One straightforward method is to recruit human annotators to create input-output pairs for the tasks of interest. Unlike data annotation in conventional NLP, such as text classification, where annotators simply assign labels to collected texts according to guidelines, creating fine-tuning data for LLMs requires more steps and effort, making it thus more challenging. Suppose we want to obtain fine-tuning data for the English-to-Chinese machine translation task. The first step is to write a prompt template to describe the task and format the problem clearly. For example,

Instruction	Translate the text from English to Chinese.
User Input	{*text*}
Output	<u>{*translation*}</u>

Then, we collect pairs of source and target texts (i.e., Chinese texts and the corresponding translations), and replace the variables {*text*} and {*translation*} to generate the fine-tuning samples. For example, given a pair of English and Chinese sentences

How's the weather today?	→	今天天气怎么样?
{*text*}		{*translation*}

we can generate a fine-tuning sample using the prompt template, like this

Instruction	Translate the text from English to Chinese.
User Input	How's the weather today?
Output	<u>今天天气怎么样?</u>

That is,

\mathbf{x}	=	Translate the text from English to Chinese.\n How's the weather today?
\mathbf{y}	=	今天天气怎么样?

We can use this (\mathbf{x}, \mathbf{y}) pair to fine-tune the LLM, as described in the previous subsection.

One difficulty here is that there are many, many different ways to write prompt templates for the same task, and different people may produce prompt templates with varying qualities and complexities. Sometimes, we may write prompt templates with overly complex or verbose instructions. Sometimes, we may not even know exactly what the target task is and how to describe it. A widely-adopted strategy is to create prompt templates for existing NLP tasks, given that there have been so many well-established NLP problems and benchmarks [Bach et al., 2022; Wang et al., 2022b; Mishra et al., 2022]. In this case, annotators can be given the original task description and many examples. Then, they can use their own ways to express how to prompt the LLM to perform the tasks. Note that, while such a method can ease the process of creating and writing prompts, we still need annotation frameworks and crowdsourcing systems to manage the work and conduct quality control. For example, we generally need to design annotation guidelines and

a unified format for writing prompt templates, especially when many annotators are contributing to the same task. One advantage of inducing prompts from existing NLP tasks is that, once the prompt templates have been developed, it is easy to generate prompts using the annotated samples in the original tasks. For example, given a bilingual dataset for English-to-Chinese translation, we can easily create a number of fine-tuning examples by filling the slots in the above template with the sentence pairs in this dataset.

Another approach is to directly use the naturally existing data available on the internet. A common example is by collecting question-and-answer pairs from QA websites to fine-tune LLMs for open-domain QA tasks [Joshi et al., 2017]. Many benchmarks in QA are built in this way because there are so many types of questions that it is impossible to think of them all by a small group of people. Instead, using data from those websites can ensure that the LLM fine-tuning data is at a good or acceptable level in terms of quantity and quality.

In addition to employing existing resources, another straightforward way to develop a fine-tuning dataset is to crowdsource the data. A simple approach is to allow users to input any question, after which responses are either manually given or automatically generated by an LLM and then manually annotated and corrected. It is thus possible to capture real user behavior and consequently gather inputs and outputs for a large number of “new” problems that traditional NLP tasks do not cover.

An issue related to the construction of the fine-tuning datasets is that we usually want the data to be as diverse as possible. Many studies have found that increasing the diversity of fine-tuning data can improve the robustness and generalization ability of LLMs. For this reason, there has been considerable interest in involving more diverse prompts and tasks in LLM fine-tuning datasets. We will provide further discussion on the generalization of fine-tuning in Section 4.2.4.

4.2.2.2 Automatically Generated Data

One limitation of manual data generation is that the quality and diversity largely depend on human experience and creativity. Therefore, if we want LLMs to handle a broad range of tasks, that is, to effectively execute any instruction, relying on human-annotated data for LLM fine-tuning is often inefficient. Moreover, the coverage of such data can be limited, and the data may even contain biases introduced by the annotators themselves. An alternative approach is to generate data automatically. For example, we can collect a number of questions through crowdsourcing, and employ a well-tuned LLM to generate answers to the questions. These question-answer pairs are then used as fine-tuning samples as usual. This method, though very simple, has been extensively applied to generate large-scale fine-tuning data for LLMs.

The above way of producing synthetic fine-tuning data is similar to those used in data augmentation for NLP. If we have an LLM, we can produce a prediction in response to any input. Repeating this process for different inputs allows us to create a sufficient number of fine-tuning samples. Such a method is particularly useful for fine-tuning new LLMs using a well-tuned LLM. However, one disadvantage of this approach is that it relies on human-crafted or collected inputs for data generation, which may turn out to be inappropriate for generalizing LLMs. In many LLM applications, a significant challenge arises from the broad range of users’ questions and requests, many of which are not covered in existing NLP tasks and datasets. In these cases, it becomes necessary to generate not only the predictions but also the inputs themselves.

Here we consider **self-instruct** as an example to illustrate how to generate LLM fine-tuning

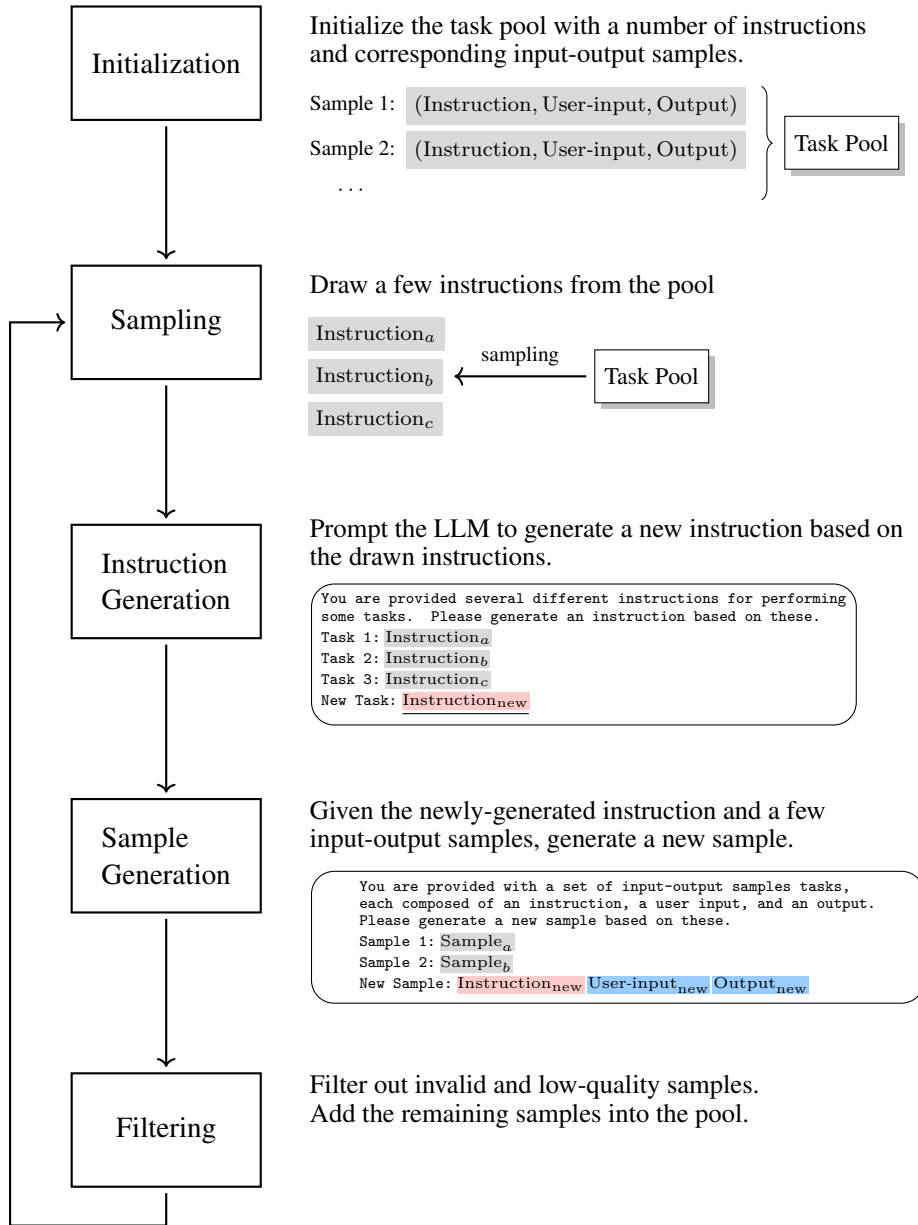


Fig. 4.4: Illustration of self-instruct [Wang et al., 2023b]. This method maintains a pool of instructions and corresponding input-output samples. Initially, the pool contains a number of hand-crafted instructions and samples. Each time, we draw a few instructions from the pool. An LLM is then prompted to generate new instructions and samples based on those drawn. Finally, the newly-generated instructions and samples are filtered and added to the pool.

samples [Wang et al., 2023d; Honovich et al., 2023]. The idea is that we can prompt an LLM to create a new instruction by learning from other instructions. Given this instruction, the LLM can then fill in other fields (such as the user input) and produce the predictions. Figure 4.4 shows a schematic illustration of self-instruct. Here we give a brief outline of the key steps involved.

- The self-instruct algorithm maintains a pool of tasks. Initially it contains a number of seed hand-crafted tasks, each with an instruction and input-output sample. As the algorithm proceeds, LLM-generated instructions and samples will be added to this pool.

- At each step, a small number of instructions are drawn from the instruction pool. For example, we can randomly select a few human-written instructions and a few LLM-generated instructions to ensure diversity.
- The selected instructions are then used as demonstration examples. Thus, the LLM can in-context learn from these examples and produce a new instruction. Below is an example template for prompting the LLM.

You are provided several different instructions for performing some tasks.
Please generate an instruction based on these.

Task 1: {instruction1}

Task 2: {instruction2}

Task 3: {instruction3}

Task 4: {instruction4}

New Task: _____

- Given the generated instruction, the LLM is then prompted to complete the sample by filling in the remaining input fields and generating the corresponding output. Below is a prompt template.

You are provided with a set of input-output samples, each composed of an instruction, a user input, and an output. Please generate a new sample based on these.

Sample 1: {instruction1}

Input: {user-input1}

Output: {output1}

Sample 2: {instruction2}

Input: {user-input2}

Output: {output2}

New Sample: {new-instruction}

- This newly-generated sample is examined by some heuristic rules (such as filtering out samples or instructions that are similar to those already in the pool). If it passes, the sample and instruction are added to the pool.

This generation process can be repeated many times to obtain a sufficient number of fine-tuning samples. Note that, above, we just show simple prompt templates for generating instruction and fine-tuning samples. Of course, we can develop better templates to generate more diverse and accurate instruction and fine-tuning samples. For example, for certain tasks like text classification,

the LLM may tend to produce biased predictions, for example, most generated samples belong to a single class. In such cases, we can adjust the order of generation of different fields. More specifically, we can specify the output (i.e., the class) with some prior, and prompt the LLM to generate user input given both the instruction and the output. This method resembles **input inversion**, where the LLM generates the input based on the specified output [Longpre et al., 2023].

Using LLM-generated instructions and fine-tuning samples has been a common method for developing LLMs, especially given that manually developing such data is so expensive that most research groups cannot afford it. In several well-tuned LLMs, their fine-tuning datasets include a certain amount of synthetic data, which has proved useful [Ouyang et al., 2022; Taori et al., 2023; Chiang et al., 2023]. There have been further studies on synthetic data generation for LLM fine-tuning. For example, one can generate more diverse instructions by introducing evolutionary algorithms [Xu et al., 2024], or use synthetic data as supervision signals in a more advanced fine-tuning process [Chen et al., 2024b]. More recently, there has also been considerable interest in using synthetic data in the pre-training stage [Gunasekar et al., 2023; Allal et al., 2024].

In many applications, a real-world scenario is that, given a task, we can collect or annotate a relatively small amount of fine-tuning data, for example, we can recruit experts to create questions for QA tasks in a specific domain. But the quantity and diversity of this data are in general not sufficient. In this case, we can use self-instruct techniques to generate more diverse question-answer pairs, and thus augment the fine-tuning data. This provides a way of bootstrapping the LLM starting from a seed set of fine-tuning samples. Note that using self-generated data is a common practice and has long been applied in NLP. For example, this approach has been successfully used in parsing and machine translation [Charniak, 1997; Sennrich et al., 2016].

4.2.3 Fine-tuning with Less Data

With the increasing prominence of instruction fine-tuning, there has been a surge in demand for large-scale, high-quality fine-tuning data. For example, the FLAN fine-tuning dataset, which is compiled from 1,836 tasks, contains 15 million samples [Longpre et al., 2023]. Fine-tuning LLMs with such large datasets is typically a computationally expensive task, especially given that updating the large number of parameters in LLMs is resource-intensive. One approach for mitigating this issue is to explore efficient model training methods, for example, one can use parameter-efficient methods to update only a small portion of the model. However, many fine-tuning datasets contain a large amount of synthetic data, where errors and biases are still inevitable.

Another approach to efficient fine-tuning is to consider only the most relevant and impactful examples for fine-tuning. We can thus reduce the amount of data that needs to be processed while still maintaining the quality of the model updates. There are several methods to achieve this. For example, Zhou et al. [2023a] built an instruction-following dataset containing only 1,000 samples by carefully crafting the prompts and collecting samples from a variety of NLP tasks. They showed that the LLaMa 65B model fine-tuned with this dataset could be competitive with or even better than models with much more fine-tuning effort. This suggests that LLMs can be adapted to respond to diverse tasks without necessarily needing fine-tuning on all types of instruction-following data. Chen et al. [2024a] developed a system based on the GPT-3.5 model to assess the quality of each instruction-following sample. Therefore, they could select high-quality samples from existing datasets, showing better fine-tuning performance with fewer fine-tuning samples. Researchers have also developed methods to either select or filter out data using

heuristics [Zhao et al., 2024; Ge et al., 2024], or to prioritize data that more significantly influences the fine-tuning process [Xia et al., 2024]. In fact, most of these methods can be seen as instances of larger families of data selection and filtering methods. And it is often the case that using higher quality (but maybe less) data is beneficial for training NLP models.

The discoveries in instruction fine-tuning somewhat differ from traditional views in NLP: the ability of models to handle complex problems can be activated with a small amount of annotated data, rather than requiring massive amounts of supervised data for extensive training. One possible explanation is that the ability of generating correct responses given instructions has been learned during pre-training, but such instruction-response mappings are not with high probabilities during inference. Fine-tuning can slightly adjust the models to get them to follow instructions, requiring significantly less training effort than pre-training. This is closely related to what is known as the **superficial alignment hypothesis**, which suggests that learning primarily occurs during pre-training, and the subsequent fine-tuning or alignment phase does not significantly contribute to the underlying knowledge base of an LLM [Zhou et al., 2023a]. Since the core abilities and knowledge of the model are already established from pre-training, effective fine-tuning for alignment with user needs can be achieved with relatively small training fine-tuning effort. This implies the possibility of fine-tuning LLMs with very little data. In another direction, it may not be necessary to restrict fine-tuning to paired instruction-response data. For example, Hewitt et al. [2024] found that instruction-following can be implicitly achieved by fine-tuning LLMs only on responses, without corresponding instructions.

A concept related to the discussion here is sample efficiency. A machine learning method is called **sample efficient** if it can learn effectively from a small number of training examples. In this sense, instruction fine-tuning is sample efficient compared with pre-training. From the perspective of machine learning, sample-efficient methods can be seen as efficient ways to sample the space of data, and are advantageous as they make optimal use of scarce data. Therefore, sampling-based learning techniques, such as many reinforcement learning algorithms, can benefit from these sample efficient approaches. For example, in human preference alignment, we can either efficiently sample preference data via reward models [Liu et al., 2024b] or improve the sampling efficiency in policy learning [Wang et al., 2024].

4.2.4 Instruction Generalization

In many machine learning and NLP problems, training a model to generalize is a fundamental goal. For example, in text classification, we expect our model to correctly classify new texts that were not seen during training. However, generalization poses additional challenges in instruction fine-tuning. We expect instruction-fine-tuned LLMs to not only generate appropriate responses for different inputs within a task but also to accurately perform various tasks as described by different instructions. To illustrate this issue, consider an LLM $\Pr(\mathbf{y}|\mathbf{c}, \mathbf{z})$, where \mathbf{c} is an instruction, \mathbf{z} is a user input, and \mathbf{y} is the corresponding model output (i.e., the response). Suppose that the performance of this model is evaluated in terms of a metric, written as $\text{Performance}(\Pr(\mathbf{y}|\mathbf{c}, \mathbf{z}))$ or $P(\mathbf{c}, \mathbf{z}, \mathbf{y})$ for short. Informally, when we say this model can generalize within a given task (indicated by the instruction \mathbf{c}^*), we mean that there may be a value ϵ such that the average performance on new inputs is above this value:

$$\frac{1}{|\mathcal{Z}|} \sum_{\mathbf{z}' \in \mathcal{Z}} P(\mathbf{c}^*, \mathbf{z}', \mathbf{y}') > \epsilon \quad (4.8)$$

where \mathcal{Z} is the set of new inputs, and \mathbf{z}' and \mathbf{y}' are an input in this set and the corresponding output, respectively.

Likewise, we can say that this model can generalize across tasks if the average performance over all instruction-input pairs is above some ϵ :

$$\frac{1}{|\mathcal{D}|} \sum_{(\mathbf{c}', \mathbf{z}') \in \mathcal{D}} P(\mathbf{c}', \mathbf{z}', \mathbf{y}') > \epsilon \quad (4.9)$$

where \mathcal{D} is the set of new instruction-input pairs.

Here, we need to deal with variations in two dimensions: instruction and user input. This makes the generalization problem very complex, because, intuitively, a model needs to learn from a vast number of tasks and different input-output pairs associated with each task to achieve good generalization. As we have discussed several times in this book, achieving such generalization incurs much lower cost than pre-training. In general, fine-tuning LLMs with instruction-response data to some extent can lead to models yielding instruction following on new tasks. Nevertheless, it is typically believed that certain efforts are still needed to adapt LLMs to make them understand and execute instructions broadly.

One way to generalize instruction fine-tuning is to increase the diversity of the fine-tuning data. In earlier studies on instruction fine-tuning, researchers developed many datasets, covering a wide variety of NLP tasks and different instructions for each task [Wang et al., 2022b; Sanh et al., 2022; Longpre et al., 2023]. By transforming these tasks into a unified format, one can fine-tune an LLM with a sufficiently large number of samples, for example, there have been several instruction fine-tuning datasets that involve over 100 NLP tasks and 1M samples. However, these early datasets mostly focus on existing academic problems, but not those that users want to deal with in real-world applications. Much recent work has shifted focus to addressing new and more practical problems. For example, there has been considerable interest in constructing datasets that contain large and complicated demonstrations and responses from SOTA models to real user queries [Wang et al., 2023c; Teknium, 2023].

Perhaps the use of large and diverse fine-tuning datasets has its origins in attempts to scale LLMs in different dimensions. Indeed, scaling laws have been used broadly to motivate the development of a wide range of different instruction-fine-tuned LLMs. And it is reasonable to scale instruction fine-tuning to make an LLM follow broad instructions. From the perspective of LLM alignment, however, scaling instruction fine-tuning might not be efficient to achieve generalization.

One problem is that instruction fine-tuning relies on supervised learning that learns to generalize and perform tasks based on instruction-response mappings. However, such an approach does not capture subtle or complex human preferences (e.g., tone, style, or subjective quality) because these are hard to encode as explicit instruction-response data. Moreover, the generalization performance is bounded by the diversity and quality of the instruction-response dataset. Given these limitations, we would instead like to employ preference models as an additional fine-tuning step following instruction fine-tuning, so the LLMs can generalize further (see Section 4.3).

Another view is that some instruction-response mappings may already be learned during pre-training, and so the pre-trained LLMs have encoded such mappings. However, since we often do not know exactly what data is used in the pre-training, it is hard to judge whether we need to learn such mappings in the fine-tuning. A related question is whether out-of-distribution generalization

is primarily achieved during pre-training or fine-tuning. While directly answering this question is beyond the scope of this chapter, it has been shown that pre-training on large and diverse datasets is effective in improving out-of-distribution performance [Hendrycks et al., 2020; Radford et al., 2021; Gunasekar et al., 2023]. This raises an interesting problem: if an LLM has been well pre-trained at scale, fine-tuning may not be as essential for out-of-distribution generalization, since the model may have already encountered sufficient distributional variation. This prompts researchers to fine-tune LLMs with modest effort or to explore new methods to achieve instruction-following. As discussed in the previous sub-section, for example, instruction following can be yielded by fine-tuning on a small number of carefully selected instruction-response pairs [Zhou et al., 2023a], or even by using methods that are not explicitly designed to do so [Kung and Peng, 2023].

The above discussion provides two different strategies: one requires scaling up fine-tuning datasets for larger diversity, the other requires small but necessary fine-tuning datasets for efficient LLM adaptation. However, in practice, involving diverse instructions often helps. In many cases, we need to adapt our LLM for specific purposes. But the LLM, which has possibly encoded broad instruction-following mappings during pre-training, might tend to behave as a general-purpose instruction executor even with modest fine-tuning. An interesting phenomenon is that when fine-tuning on math data, the resulting LLM might not specialize in math outputs. Instead, this model might respond normally to general instructions, for example, it could generate poetry if instructed to do so [Hewitt, 2024]. This is not a bad thing, but it shows that LLMs may not easily change their nature of following general instructions. In this case, additional adaptations with more diverse data may help adjust the way the LLM follows instructions, particularly for those tasks we aim to address.

4.2.5 Using Weak Models to Improve Strong Models

So far we have explored a variety of instruction fine-tuning methods based on labeled data. One of the limitations of many such methods is that they require the data to be annotated by humans or generated by strong LLMs, which can provide accurate supervision signals in fine-tuning. However, in many cases, the LLM we have in hand is already strong (or at least is advantageous in specific aspects of problem solving), and thus it is not easy to find a superior model for supervision. Even for human experts, when the problem becomes complex, providing correct and detailed answers might be difficult, or sometimes infeasible. For example, when faced with an extremely long document, the experts would find it challenging to identify any inconsistencies, subtle biases, or missing key points without conducting an exhaustive and time-consuming review.

One may ask at this point: can we use weak LLMs to supervise strong LLMs? This seems to be a significant challenge, but it may reflect a future scenario where we need to supervise AI systems that are smarter than humans or any other AI systems [Burns et al., 2023b]. The problem of using smaller, less complex models to improve the training of larger, more complex models is also called the **weak-to-strong generalization** problem. While there have not been mature approaches to weak-to-strong generalization, using smaller models to assist stronger models has indeed proven useful in several areas of LLMs.

For instruction fine-tuning, one of the simplest ways of applying weak LLMs is to use these models to generate synthetic fine-tuning data. Suppose we have a collection of inputs X , where each input includes an instruction and a user input if necessary. For each $\mathbf{x} \in X$, we use a weak LLM $\text{Pr}^w(\cdot)$ to generate a prediction $\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \text{Pr}^w(\mathbf{y}|\mathbf{x})$. Then, the strong LLM $\text{Pr}_\theta^s(\cdot)$ can

be trained on these generated predictions (see Eq. (4.1)):

$$\tilde{\theta} = \arg \max_{\theta} \sum_{\mathbf{x} \in X} \log \Pr_{\theta}^s(\hat{\mathbf{y}}|\mathbf{x}) \quad (4.10)$$

where θ is the model parameters.

The above form transforms the fine-tuning problem into a knowledge distillation problem, in other words, we distill knowledge from the weak model to the strong model. Consequently, we can employ various knowledge distillation methods to achieve this goal. However, explaining weak-to-strong fine-tuning from the perspective of knowledge distillation is not straightforward. A major concern is that the strong model may merely imitate or overfit the errors of the weak model and fail to generalize. For example, the fine-tuned strong model still cannot solve difficult problems that the weak model cannot accurately predict. Fortunately, preliminary experiments in this line of research have shown positive and promising results. For example, [Burns et al. \[2023a\]](#) found that fine-tuning the strong pre-trained GPT-4 model with GPT-2-level supervision could improve generalization across several NLP tasks. To measure how the weak model improves the generalization of the strong model, we define the following terms:

- **Weak Performance** (P_{weak}). This is the test-set performance of the weak model, which can be regarded as the baseline performance.
- **Weak-to-strong Performance** ($P_{\text{weak} \rightarrow \text{strong}}$). This is the test-set performance of the strong model that is fine-tuned with the weak model.
- **Strong Ceiling Performance** (P_{ceiling}). This is the test-set performance of the strong model that is fine-tuned with ground truth data. For example, we fine-tune the strong model with human-annotated predictions and take the resulting model as a ceiling.

Then, the **performance gap recovered (PGR)** can be defined as

$$\text{PGR} = \max \left\{ 0, \frac{P_{\text{weak} \rightarrow \text{strong}} - P_{\text{weak}}}{P_{\text{ceiling}} - P_{\text{weak}}} \right\} \quad (4.11)$$

This metric measures how much of the performance gap between the ceiling model and the weak model can be recovered by the weak-to-strong model. A PGR of 1 indicates that the weak-to-strong fine-tuning can completely close the performance gap, whereas a PGR of 0 indicates no improvement. In [Burns et al. \[2023a\]](#)'s work, it is shown that PGR can be around 0.8 on 22 NLP classification tasks. It should be noted that, while the potential of weak-to-strong fine-tuning is promising, achieving substantial weak-to-strong generalization remains a challenging goal that needs further investigation [[Aschenbrenner, 2024](#)].

Fine-tuning LLMs with weak supervision is just one choice for using small models to improve large models. Although this section primarily focuses on fine-tuning LLMs, we also mention other methods here to give a more complete discussion (see Figure 4.5 for illustrations of these methods).

- Instead of using small models to generate synthetic data, it is also straightforward to incorporate knowledge distillation loss based on these models. For example, a simple loss

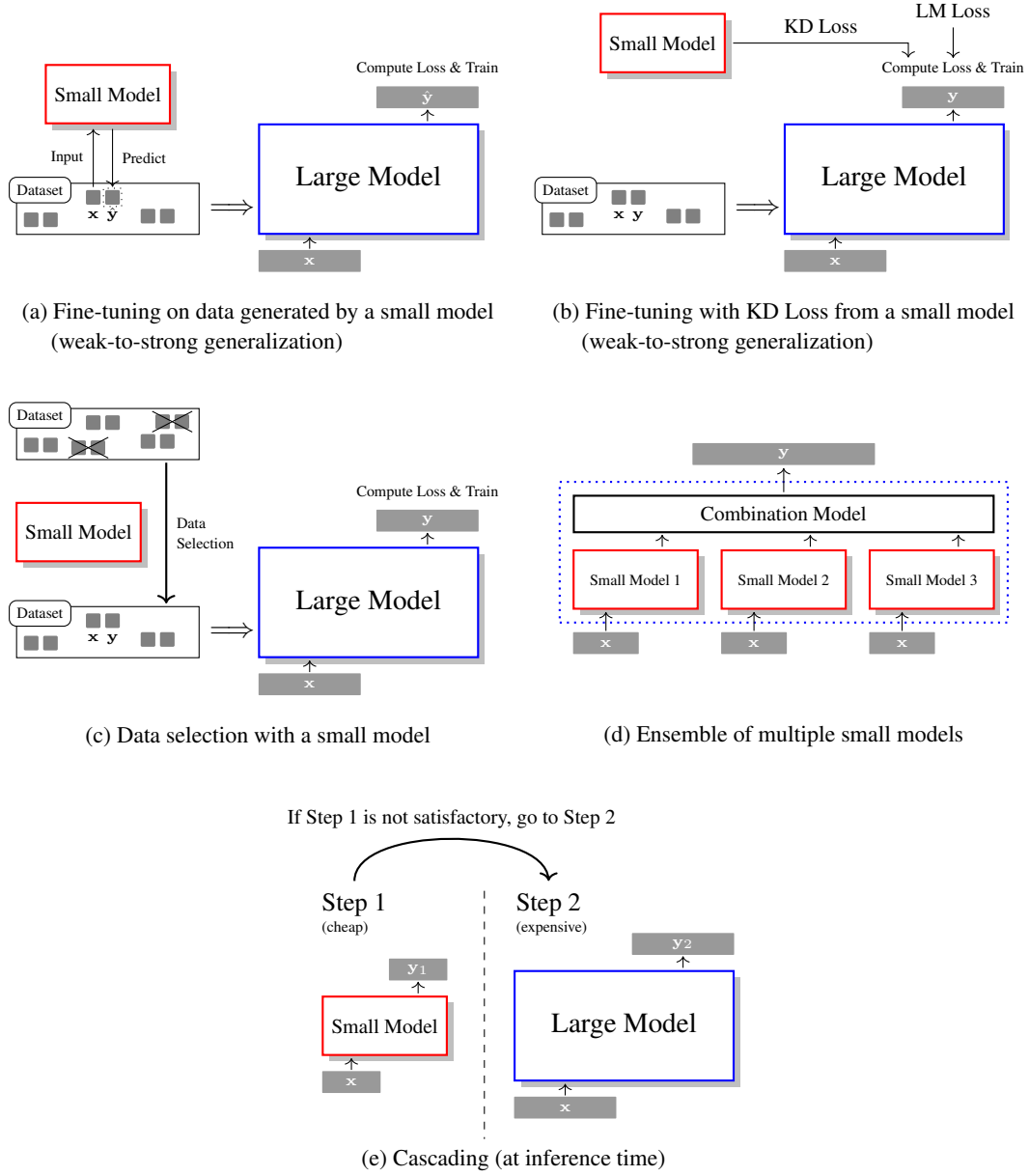


Fig. 4.5: Illustrations of using small models to improve large models in LLMs. One approach involves using smaller models for the fine-tuning or pre-training of larger models. This includes generating synthetic data (a), incorporating auxiliary loss (b), and selecting appropriate data (c). Another approach involves combining small models and large models. This includes learning a strong model by aggregating multiple small models (d), and cascading small models with large models (e).

function that measures the difference between the small and large models can be defined as:

$$\text{Loss}_{\text{kd}} = \text{KL}(\text{Pr}^w(\cdot|\mathbf{x}) \parallel \text{Pr}_\theta^s(\cdot|\mathbf{x})) \quad (4.12)$$

Then, we can add this loss to the original loss of language modeling, and yield the following training objective

$$\tilde{\theta} = \arg \max_{\theta} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \log \text{Pr}_\theta^s(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Loss}_{\text{kd}} \quad (4.13)$$

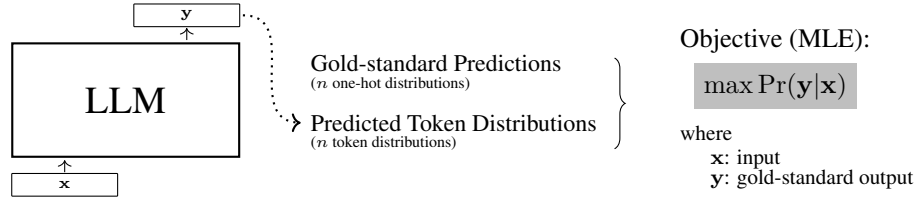
where \mathcal{D} is the set of input and output pairs, and λ is the coefficient of the interpolation. This method can be employed in either the pre-training or fine-tuning phase. We can adjust λ to control how much the small model influences the training. For example, we can gradually decrease λ to make the training rely more on the original language modeling loss as the large model becomes more capable.

- Another approach to involving small models in LLM pre-training and fine-tuning is to use them to do data selection or filtering. Given a sequence, we can compute the likelihood or cross-entropy using a small model. These quantities can then be used as criteria for selecting or filtering data. For example, sequences with low likelihood or high cross-entropy might be excluded from the training set, as they are less aligned with the small model's learned distribution. Conversely, sequences with high likelihood or low cross-entropy can be prioritized, ensuring that the training focuses on more relevant or high-quality data.
- Ensemble learning is a simple and effective way to build a strong model by combining multiple weak models. Applying this technique to LLMs is straightforward. We can aggregate distributions predicted by multiple small models or specialized submodels, and derive the final prediction from the aggregated results. This aggregation can be done using methods such as majority voting, weighted averaging, or stacking.
- Small models can also be employed at inference time to improve overall efficiency. Suppose we have a large model that is slow but more accurate, and a small model that is fast but less accurate. In model cascading, the small model first processes the input data, quickly generating preliminary results. If these results meet certain pre-defined criteria, they can be directly used. However, if the initial results are not sufficiently good, the input is then passed to the larger, more accurate model to produce a better result. This approach significantly reduces computational costs and latency, as the small model can effectively handle many inputs without access to the large model.

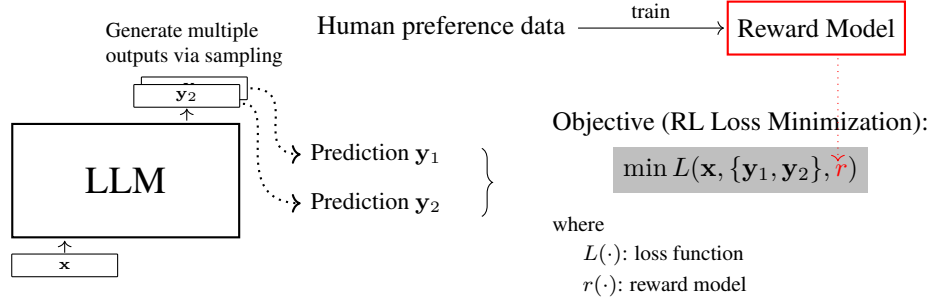
4.3 Human Preference Alignment: RLHF

So far in this chapter, we have focused on fine-tuning LLMs using input-output paired labeled data. This approach allows us to adapt LLMs for instruction-following via supervised learning. In many applications, however, LLMs are required not only to follow instructions but also to act in ways that are more aligned with human values and preferences. Consider a scenario where a user asks an LLM how to hack into a computer system. If the LLM is not appropriately aligned, it may respond by providing details on how to perform this illegal activity. Instead, a more desirable response might be to advise the user against engaging in illegal activities and offer a general overview of the consequences of such actions. The difficulty in achieving this is that the ethical nuances and contextual considerations required for an LLM to respond appropriately in such scenarios are not always straightforward to encode into a fine-tuning dataset. What's even more challenging is that, often, humans themselves cannot precisely express their own preferences.

In this section, we discuss an alternative LLM fine-tuning method, called reinforcement learning from human feedback or RLHF for short [Christiano et al., 2017; Stiennon et al., 2020]. The basic idea behind RLHF is that LLMs can learn from comparisons of model outputs using reward



(a) Supervised fine-tuning (maximizing the prediction probability given the input)



(b) Reinforcement Learning from Human Feedback

Fig. 4.6: Supervised fine-tuning vs. reinforcement learning from human feedback. In supervised fine-tuning, we optimize the LLM by maximizing the probability of the prediction given the input. In reinforcement learning from human feedback, we first train a reward model on human preference data (on each pair of predictions, evaluators are asked to choose which one they prefer). Then, we use this reward model to supervise the LLM during fine-tuning.

models (see Figure 4.6). To do this, we can recruit human experts who indicate their preferences between pairs of outputs generated by the LLM. This preference data is used to train a reward model that can predict the perceived quality of LLM outputs. Once trained, the reward model provides feedback by assigning scores to new outputs that the LLM generates in response to the inputs. The LLM uses these scores to update its parameters through reinforcement learning algorithms. In the rest of this section, we will first introduce the basic knowledge of reinforcement learning to facilitate the discussion, and then discuss methods for training reward models and aligning LLMs with these models.

4.3.1 Basics of Reinforcement Learning

We begin by looking at some basic concepts of reinforcement learning. Note that the notation used here slightly differs from that used in the previous sections and chapters because we want to make our description more consistent with those in the reinforcement learning literature. Nevertheless, we will show how this notation corresponds to the language modeling notation. The reader who is already familiar with reinforcement learning techniques may skip or skim this subsection.

A general reinforcement learning framework describes how an agent interacts with a dynamic environment. This interaction is modeled as a sequence of actions taken by the agent in response to the state of the environment. At each time step, the agent observes the current state, chooses an action based on its policy, performs the action, and then receives feedback from the environment in the form of a reward and a new state. This sequence of observe-act-receive feedback is repeated

until the agent achieves its goal.

A reinforcement learning system involves several components:

- **Agent.** This is the learner or decision-maker in reinforcement learning. In the context of LLMs, it can be seen as the LLM itself.
- **Environment.** This includes everything external to the agent with which the agent interacts. But the environment in LLMs is less about a physical or virtual space and more about the framework within which the agent (e.g., an LLM) receives feedback and learns.
- **State (s).** A state represents the current situation of the environment. Given a sequence of tokens for language modeling, a state at a time step can be viewed as the tokens we observed so far, that is, the context tokens we take to predict the next token. For example, we can define $(\mathbf{x}, \mathbf{y}_{<t})$ as the state when predicting the next token at the time step t .
- **Action (a).** Actions represent possible decisions the agent can make. We can see them as possible predicted tokens in the vocabulary.
- **Reward (R).** The reward is the feedback from the environment that evaluates the success of an action. For example, $r(s, a, s')$ denotes the reward the agent receives for taking the action a at the state s and moving to the next state s' . If the state-action sequence is given, we can denote the reward at the time step t as $r_t = r(s_t, a_t, s_{t+1})$. Also note that if the decision-making process is deterministic, we can omit s_{t+1} because it can be determined by s_t and a_t . In such cases, we can use $r(s_t, a_t)$ as shorthand for $r(s_t, a_t, s_{t+1})$.
- **Policy (π).** For an LLM, a policy is defined as the probability distribution over the tokens that the LLM predicts, given the preceding context tokens. Formally, this can be expressed as

$$\pi(a|s) = \Pr(y_t|\mathbf{x}, \mathbf{y}_{<t}) \quad (4.14)$$

where a corresponds to the token y_t , and s corresponds to the context $(\mathbf{x}, \mathbf{y}_{<t})$. Figure 4.7 illustrates how an LLM can be treated as a policy in the reinforcement learning framework.

- **Value Function (V and Q).** A **state-value function** (or value function, for short) assesses the expected discounted return (i.e., accumulated rewards) for an agent starting from a particular state s and following a specific policy π . It is defined as:

$$\begin{aligned} V(s) &= \mathbb{E} \left[r(s_0, a_0, s_1) + \gamma r(s_1, a_1, s_2) + \gamma^2 r(s_2, a_2, s_3) + \cdots \mid s_0 = s, \pi \right] \\ &= \mathbb{E} \left[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s, \pi \right] \\ &= \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi \right] \end{aligned} \quad (4.15)$$

where $\gamma \in [0, 1]$ is the discount factor that adjusts the importance of future rewards, $s_0 = s$ indicates that the agent starts with the state s , and the expectation \mathbb{E} is performed over all possible trajectories (i.e., state-action sequences). Similarly, an **action-value function** (or

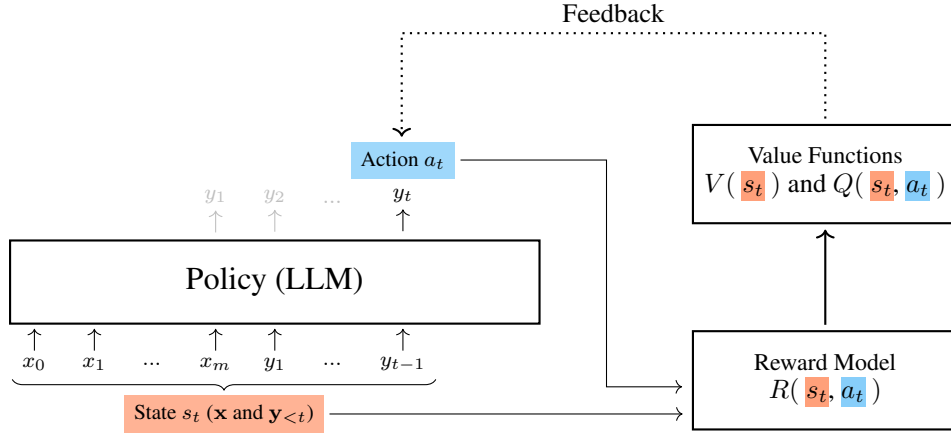


Fig. 4.7: LLM as policy in reinforcement learning. At each step t , the LLM predicts a token y_t given the model input \mathbf{x} and the previously-generated tokens $\mathbf{y}_{<t}$. This process can be framed as a reinforcement learning problem, where y_t serves as the action, $(\mathbf{x}, \mathbf{y}_{<t})$ as the state, and the predicted distribution $\Pr(y_t|\mathbf{x}, \mathbf{y}_{<t})$ as the policy. Once y_t is predicted, the LLM inputs both $(\mathbf{x}, \mathbf{y}_{<t})$ and y_t to the reward model, which evaluates how effectively the chosen token contributes to achieving the desired textual outcome. This evaluation generates reward scores which are used to compute the value functions $V(s_t)$ and $Q(s_t, a_t)$. These functions then provide feedback to the LLM and guide the policy training.

Q-value function) measures the expected return starting from a state s taking an action a and thereafter following a policy π , given by

$$Q(s, a) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right] \quad (4.16)$$

where $a_0 = a$ indicates that the action taken at the initial state is a .

The goal of reinforcement learning is to learn a policy that maximizes the **cumulative reward** (or **return**) the agent receives over the long run. Given a state-action sequence $\tau = \{(s_1, a_1), \dots, (s_T, a_T)\}$ ¹, the cumulative reward over this sequence can be written as

$$R(\tau) = \sum_{t=1}^T r_t \quad (4.17)$$

The expectation of this cumulative reward over a space of state-action sequences is given in the form

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim \mathcal{D}} [R(\tau) \mid \pi_\theta] \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \sum_{t=1}^T r_t \end{aligned} \quad (4.18)$$

¹We assume the state-action sequence begins with s_1 and a_1 , rather than s_0 and a_0 , to align with the notation commonly used in this chapter, where the prediction \mathbf{y} typically starts from y_1 . Of course, it is also common to denote a state-action sequence as $\{(s_0, a_0), \dots, (s_T, a_T)\}$ or $\{(s_0, a_0), \dots, (s_{T-1}, a_{T-1})\}$ in the literature. But this variation in notation does not affect the discussion of the models presented here.

where $\tau \sim \mathcal{D}$ indicates that τ is drawn from the state-action sequence space \mathcal{D} , and the subscript θ indicates the parameters of the policy. $J(\theta)$ is also called the **performance function**.

Then the training objective is to maximize $J(\theta)$:

$$\tilde{\theta} = \arg \max_{\theta} J(\theta) \quad (4.19)$$

Now, we have a simple reinforcement learning approach: 1) we sample a number of state-action sequences; then, 2) we evaluate each sequence using the performance function; then, 3) we update the model to maximize this performance function. If we take Eq. (4.18) and use gradient descent to optimize the policy, this approach would constitute a form of policy gradient methods [Williams, 1992].

Note that in many NLP problems, such as machine translation, rewards are typically sparse. For instance, a reward is only received at the end of a complete sentence. This means that $r_t = 0$ for all $t < T$, and r_t is non-zero only when $t = T$. Ideally, one might prefer feedback to be immediate and frequent (dense), and thus the training of the policy can be easier and more efficient. While several methods have been proposed to address sparse rewards, such as reward shaping, we will continue in our discussion to assume a sparse reward setup, where the reward is available only upon completing the prediction.

The model described in Eqs. (4.17-4.19) establishes a basic form of reinforcement learning, and many variants and improvements of this model have been developed. Before showing those more sophisticated models, let us take a moment to interpret the objective function $J(\theta)$ from the perspective of policy gradient. In gradient descent, we need to compute the gradient of $J(\theta)$ with respect to θ :

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta} &= \frac{\partial \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) R(\tau)}{\partial \theta} \\ &= \sum_{\tau \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) \frac{\partial \Pr_{\theta}(\tau) / \partial \theta}{\Pr_{\theta}(\tau)} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_{\theta}(\tau) \frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \end{aligned} \quad (4.20)$$

In some cases, we will assume that every sequence in \mathcal{D} is equally probable (i.e., $\Pr_{\theta}(\tau) = 1/|\mathcal{D}|$). In this case we can simplify Eq. (4.20) and need only consider the terms $\frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta}$ and $R(\tau)$:

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \sum_{\tau \in \mathcal{D}} \frac{\partial \log \Pr_{\theta}(\tau)}{\partial \theta} R(\tau) \quad (4.21)$$

One advantage of this result is that $R(\tau)$ does not need to be differentiable, which means that we can use any type of reward function in reinforcement learning.

By treating the generation of the sequence τ as a Markov decision process, we can further

derive $\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta}$, and obtain:

$$\begin{aligned} \frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} &= \frac{\partial}{\partial \theta} \log \prod_{t=1}^T \pi_\theta(a_t|s_t) \Pr(s_{t+1}|s_t, a_t) \\ &= \frac{\partial}{\partial \theta} \sum_{t=1}^T \underbrace{\log \pi_\theta(a_t|s_t)}_{\text{policy}} + \frac{\partial}{\partial \theta} \sum_{t=1}^T \underbrace{\log \Pr(s_{t+1}|s_t, a_t)}_{\text{dynamics}} \end{aligned} \quad (4.22)$$

where the gradient is decomposed into two parts: the policy gradient and the dynamics gradient. The policy component, $\log \pi_\theta(a_t|s_t)$, determines the log-probability of taking action a_t given state s_t , and it is parameterized by θ . The dynamics component, $\log \Pr(s_{t+1}|s_t, a_t)$, represents the log-probability of transitioning to state s_{t+1} from state s_t after taking action a_t . In typical reinforcement learning settings, the dynamics are not directly influenced by the policy parameters θ , and thus, their derivatives are often zero. In this case, therefore, Eq. (4.22) can be simplified to:

$$\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{t=1}^T \log \pi_\theta(a_t|s_t) \quad (4.23)$$

In other words, we only concentrate on optimizing the policy without concerning ourselves with the underlying dynamics.

Substituting Eq. (4.23) into Eq. (4.21), and expanding $R(\tau)$, we then obtain

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left(\sum_{t=1}^T \log \pi_\theta(a_t|s_t) \sum_{t=1}^T r_t \right) \quad (4.24)$$

While this policy gradient approach is straightforward, it suffers from the problem that the variance of the estimated gradients can be very high, making the learning process noisy and inefficient. One reason for this high variance problem is that rewards can vary greatly across different steps or scenarios. Imagine that in a sequence of action decisions, the reward model tends to assign small rewards to good actions (e.g., $R_t = 2$) and large penalties to poor actions (e.g., $R_t = -50$). Such varying reward scales for good and poor actions can result in a very low total reward for the entire sequence, even if it includes good actions.

One simple method for reducing the variance of the gradient is to set a baseline b and subtract it from $\sum_{t=1}^T r_t$, resulting in $\sum_{t=1}^T r_t - b$.² Here, the baseline can be interpreted as a reference point. By centering the rewards around this baseline, we remove systematic biases in the reward signal, making the updates more stable and less sensitive to extreme fluctuations in individual rewards.

²In fact, the use of a baseline b does not change the variance of the total rewards $\sum_{t=1}^T r_t$. However, it is important to note that while introducing a baseline does not alter the overall variance of the rewards, it helps reduce the variance of the gradient estimates. This is because subtracting the baseline from the total rewards effectively reduces fluctuations around their mean, which makes the gradient estimates more stable. In general, the operation $\sum_{t=1}^T r_t - b$ centers the rewards around zero (e.g., b is defined as the expected value of $\sum_{t=1}^T r_t$), which can lead to reduced variance in the product $\sum_{t=1}^T \log \pi_\theta(a_t|s_t) (\sum_{t=1}^T r_t - b)$.

This policy gradient model with a baseline can be given by

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left(\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \right) \left(\sum_{t=1}^T r_t - b \right) \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left(\sum_{k=1}^T r_k - b \right) \right] \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left(\sum_{k=1}^{t-1} r_k + \sum_{k=t}^T r_k - b \right) \right] \tag{4.25}
\end{aligned}$$

Here we write $\sum_{k=1}^T r_k$ as the sum of two terms $\sum_{k=1}^{t-1} r_k$ and $\sum_{k=t}^T r_k$ to distinguish between the rewards accrued before and after the action at time step t . Note that in Markov decision processes, the future is independent of the past given the present. Therefore, the action taken at time step t cannot influence the rewards received before t , or in other words, the rewards prior to t are already “fixed” by the time the action at t is chosen. The term $\sum_{k=1}^{t-1} r_k$ does not contribute to the gradient and can be omitted, leading to a simplified version of Eq. (4.25)

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left[\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) \left(\sum_{k=t}^T r_k - b \right) \right] \tag{4.26}$$

Also note that removing $\sum_{k=t}^T r_k$ can further reduce the variance of the gradient.

There are many ways to define the baseline b . Here we consider the value function of the state s_t , that is, the estimated value of being in state s_t : $V(s_t) = \mathbb{E}(r_t + r_{t+1} + \dots + r_T)$. Hence we have

$$\begin{aligned}
A(s_t, a_t) &= \sum_{k=t}^T r_k - b \\
&= \sum_{k=t}^T r_k - V(s_t) \tag{4.27}
\end{aligned}$$

where $\sum_{k=t}^T r_k$ represents the actual return received, and $V(s_t)$ represents the expected return. $A(s_t, a_t)$ (or A_t for short) is called the **advantage** at time step t , which quantifies the relative benefit of the action a_t compared to the expected value of following the policy from the state s_t onward.

By using the advantage function $A(s_t, a_t)$, the gradient of $J(\theta)$ can be written in the form

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \left(\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \right) \tag{4.28}$$

This optimization objective corresponds to the **advantage actor-critic (A2C)** method in reinforcement learning [Mnih et al., 2016]. In this method, the actor aims at learning a policy. It updates the policy parameters using Eq. (4.28) to help focus more on actions that are likely to improve performance. The critic, on the other hand, updates its estimation of the value function, which is used to calculate the advantage function $A(s_t, a_t)$, thus serving as the evaluator of the

policy being learned by the actor.

In the A2C method, $A(s_t, a_t)$ is typically expressed as the difference of the action-value function $Q(s_t, a_t)$ and the state-value function $V(s_t)$

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \quad (4.29)$$

At first glance, this model may seem challenging to develop because it requires two separate sub-models to calculate $Q(s_t, a_t)$ and $V(s_t)$ respectively. Fortunately, considering that $Q(s_t, a_t)$ can be defined as the return $r_t + V(s_{t+1})$, we can rewrite Eq. (4.29) as

$$A(s_t, a_t) = r_t + V(s_{t+1}) - V(s_t) \quad (4.30)$$

or alternatively, introduce the discount factor γ to obtain a more general form

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (4.31)$$

$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$ is also called the **temporal difference (TD)** error. What we need is to train a critic network for the value function $V(s_t)$, and then use it to compute the advantage function³.

Up to this point, we have spent considerable space discussing the basics of reinforcement learning, especially on how to derive the optimization objective for the A2C method. However, reinforcement learning is a vast field, and many technical details cannot be covered here. The interested reader can refer to reinforcement learning books for more details [Sutton and Barto, 2018; Szepesvári, 2010]. Nevertheless, we now have the necessary knowledge to further discuss RLHF. In the subsequent subsections, we will return to the discussion on LLM alignment, demonstrating how to use the A2C method for aligning with human preferences.

4.3.2 Training Reward Models

We have shown that reward models play a very important role in the general reinforcement learning framework and form the basis for computing value functions. We now consider the problem of training these reward models.

In RLHF, a reward model is a neural network that maps a pair of input and output token sequences to a scalar. Given an input \mathbf{x} and an output \mathbf{y} , the reward can be expressed as

$$r = \text{Reward}(\mathbf{x}, \mathbf{y}) \quad (4.33)$$

where $\text{Reward}(\cdot)$ is the reward model. r can be interpreted as a measure of how well the output \mathbf{y} aligns with the desired behavior given the input \mathbf{x} . As discussed in the previous subsection, both \mathbf{x}

³The training loss for the value network (or critic network) in A2C is generally formulated as the mean squared error between the computed return $r_t + \gamma V(s_{t+1})$ and the predicted state value $V(s_t)$. Suppose that the value network is parameterized by ω . The loss function is given by

$$\mathcal{L}_v(\omega) = \frac{1}{M} \sum (r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t))^2 \quad (4.32)$$

where M is the number of training samples, for example, for a sequence of T tokens, we can set $M = T$.

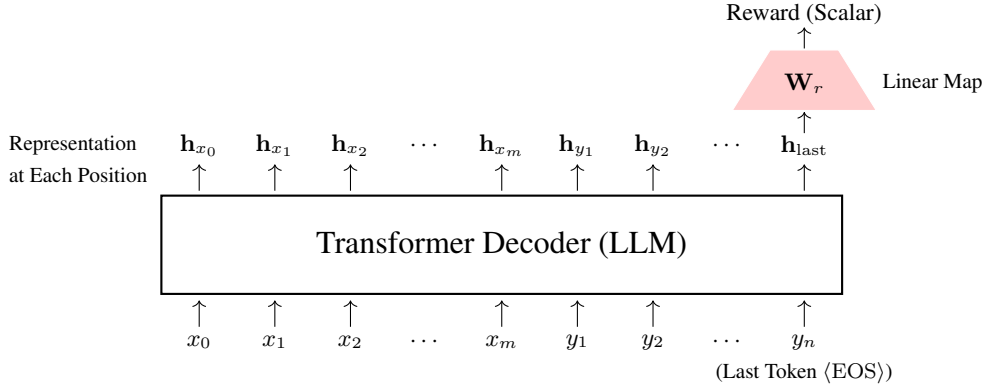


Fig. 4.8: Architecture of the reward model based on Transformer. The main component of this model is still an LLM. We use the Transformer decoder as the sequence representation model. We extract the representation of the last position of the decoder as the representation of the entire sequence $[\mathbf{x}, \mathbf{y}]$. We then map this representation to a scalar through a linear transformation, which serves as the reward score for $[\mathbf{x}, \mathbf{y}]$.

and \mathbf{y} are assumed to complete texts. This means that the reward model evaluates the relationship between inputs and outputs that provide full semantic content. For example, when applying the reward model, it assigns a value of 0 (or another predetermined value) at each position t in the output sequence $\mathbf{y} = y_1 \dots y_n$. Only at the final position, when $t = n$, does the reward model generate the actual reward score. To keep the notation uncluttered, we will use $r(\mathbf{x}, \mathbf{y})$ to denote the reward model $\text{Reward}(\mathbf{x}, \mathbf{y})$ from here on.

There are many ways to implement the reward model. One simple approach is to build the reward model based on a pre-trained LLM. More specifically, we can concatenate \mathbf{x} and \mathbf{y} to form a single token sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$. We run a pre-trained LLM on this sequence, as usual, and at each position, we obtain a representation from the top-most Transformer layer. Then, we take the representation at the last position (denoted by \mathbf{h}_{last}) and map it to a scalar via linear transformation:

$$r(\mathbf{x}, \mathbf{y}) = \mathbf{h}_{\text{last}} \mathbf{W}_r \quad (4.34)$$

where \mathbf{h}_{last} is a d -dimensional vector, and \mathbf{W}_r is a $d \times 1$ linear mapping matrix. This architecture of the reward model is illustrated in Figure 4.8.

To train the reward model, the first step is to collect human feedback on a set of generated outputs. Given an input \mathbf{x} , we use the LLM to produce multiple candidate outputs $\{\mathbf{y}_1, \dots, \mathbf{y}_N\}$. Human feedback can be obtained in several ways:

- **Pairwise Comparison (Pairwise Ranking).** Given two different outputs, human experts select which one is better.
- **Rating.** Human experts provide a score or rating to each output. This score is often a continuous or discrete numerical value, such as a score on a scale (e.g., 1-5 stars, or 1-10 points). In some cases, the rating might be binary, indicating a “yes/no” or “positive/negative” preference.

- **Listwise Ranking.** Human experts are asked to rank or order the given set of possible outputs.

Here we consider pairwise comparison feedback as it is one of the simplest and most common forms of human feedback used in RLHF. In this setting, each time, two outputs $(\mathbf{y}_a, \mathbf{y}_b)$ are randomly drawn from the candidate pool $\{\mathbf{y}_1, \dots, \mathbf{y}_N\}$. Human experts are then presented with these pairs and asked to decide which output they prefer based on specific criteria, such as clarity, relevance, and accuracy. The human feedback can be encoded as a binary label, $\mathbf{y}_a \succ \mathbf{y}_b$ for a preference for \mathbf{y}_a , and $\mathbf{y}_b \succ \mathbf{y}_a$ for a preference for \mathbf{y}_b .

One simple and widely used model for describing such pairwise comparisons is the **Bradley-Terry model** [Bradley and Terry, 1952]. It is a probabilistic model that estimates the probability that one item is preferred over another. Adapting this model to the notation used here, we can write the probability that \mathbf{y}_a is preferred over \mathbf{y}_b in the form

$$\begin{aligned} \Pr(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) &= \frac{e^{r(\mathbf{x}, \mathbf{y}_a)}}{e^{r(\mathbf{x}, \mathbf{y}_a)} + e^{r(\mathbf{x}, \mathbf{y}_b)}} \\ &= \frac{e^{r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)}}{e^{r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)} + 1} \\ &= \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)) \end{aligned} \quad (4.35)$$

When training the reward model, we want to maximize this preference probability. A loss function based on the Bradley-Terry model is given by

$$\mathcal{L}_r(\phi) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \quad (4.36)$$

where $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ is drawn from a human-annotated dataset \mathcal{D}_r consisting of preference pairs of outputs and their corresponding inputs. ϕ represents the parameters of the reward model, which includes both the parameters of the Transformer decoder and the linear mapping matrix \mathbf{W}_r . In practice, assuming $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ is uniformly sampled from \mathcal{D}_r , we can replace the expectation with a summation

$$\mathcal{L}_r(\phi) = -\frac{1}{|\mathcal{D}_r|} \sum_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \in \mathcal{D}_r} \log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) \quad (4.37)$$

The goal of training the reward model is to find the optimal parameters $\hat{\phi}$ that minimize this loss function, given by

$$\hat{\phi} = \arg \min_{\phi} \mathcal{L}_r(\phi) \quad (4.38)$$

Since the reward model itself is also an LLM, we can directly reuse the Transformer training procedure to optimize the reward model. The difference from training a standard LLM is that we only need to replace the cross-entropy loss with the pairwise comparison loss as described in Eq. (4.37). After the training of the reward model, we can apply the trained reward model $r_{\hat{\phi}}(\cdot)$ to supervise the target LLM for alignment.

It is worth noting that although we train the reward model to perform pairwise ranking, we apply it to score each input-output pair independently during the alignment process. The pairwise

ranking objective ensures that the reward model is sensitive to subtle differences between outputs, but we rely on the continuous scores produced by the reward model to guide the optimization of the LLM. An advantage of this approach is that we can choose from or combine various ranking loss functions, and still apply the resulting reward models in the same way as we have done in this subsection. This consistency ensures a unified framework for aligning the LLM, regardless of the specific ranking loss used during reward model training.

4.3.3 Training LLMs

Having obtained the reward model, we then train the policy (i.e., the LLM) via the A2C method. Recall from Section 4.3.1 that a state-action sequence or trajectory τ can be evaluated by the utility function

$$U(\tau; \theta) = \sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \quad (4.39)$$

where $A(s_t, a_t)$ is the advantage of taking the action a_t given the state s_t . An estimate of $A(s_t, a_t)$ is defined as the TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$, where the value function $V(s_t)$ is trained with the reward model.

Given this utility function, the A2C-based loss function can be written in the form

$$\begin{aligned} \mathcal{L}(\theta) &= -\mathbb{E}_{\tau \sim \mathcal{D}} [U(\tau; \theta)] \\ &= -\mathbb{E}_{\tau \sim \mathcal{D}} \left[\sum_{t=1}^T \log \pi_{\theta}(a_t | s_t) A(s_t, a_t) \right] \end{aligned} \quad (4.40)$$

where \mathcal{D} is a space of state-action sequences. As usual, the goal of training the policy is to minimize this loss function

$$\tilde{\theta} = \arg \min_{\theta} \mathcal{L}(\theta) \quad (4.41)$$

If we map the problem back to the language modeling problem and adopt the notation from LLMs, the loss function can be written as:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}} [U(\mathbf{x}, \mathbf{y}; \theta)] \quad (4.42)$$

where

$$U(\mathbf{x}, \mathbf{y}; \theta) = \sum_{t=1}^T \log \pi_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (4.43)$$

Here $\pi_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) = \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})$ is the LLM parameterized by θ .

In general, we do not have a human annotated input-output dataset \mathcal{D} in RLHF, but a dataset containing inputs only. The outputs, in this case, are typically the predictions made by the LLM.

The loss function is then defined as

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} [U(\mathbf{x}, \mathbf{y}; \theta)] \quad (4.44)$$

where \mathcal{D} denotes the input-only dataset, and $\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})$ denotes that the output \mathbf{y} is sampled by the policy $\pi_{\theta}(\cdot|\mathbf{x})$.

The above formulation provides a basic form of the A2C method for LLMs. Improved versions of this model are more commonly used in RLHF. In the following discussion, we will still use the reinforcement learning notation to simplify the presentation and will get back the language modeling notation later.

One common improvement of policy gradient methods is to use **importance sampling** to refine the estimation of $U(\tau; \theta)$. This can be written as

$$U(\tau; \theta) = \sum_{t=1}^T \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} A(s_t, a_t) \quad (4.45)$$

Here we replace the log-probability $\log \pi_{\theta}(a_t|s_t)$ with the ratio $\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$. θ_{ref} denotes the parameters of the previous policy (such as an initial model from which we start the training). So $\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$, also called the **ratio function**, can be interpreted as the log-probability ratio between the current policy π_{θ} and the previous policy $\pi_{\theta_{\text{ref}}}$ (call it the reference policy). By using the ratio function we reweight the observed rewards based on the likelihood of the actions under the current policy versus the reference policy. When $\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} > 1$, the action a_t is more favored by the current policy compared to the reference policy. By contrast, when $\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} < 1$, the action a_t is less favored by the current policy⁴.

⁴Consider a more general case where we wish to evaluate the policy using its expected reward (also see Eq. (4.18))

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)] \quad (4.46)$$

Here $\tau \sim \pi_{\theta}$ means that the sequence τ is generated by the policy π_{θ} . Alternatively, we can write $J(\theta)$ in another form

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] \quad (4.47)$$

It is not difficult to find that the right-hand sides of these equations are essentially the same since $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] = \sum_{\tau} \Pr_{\theta_{\text{ref}}}(\tau) \frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) = \sum_{\tau} \Pr_{\theta}(\tau) R(\tau) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)]$

Note that this equivalence holds only when the expectation is performed over the entire sequence space. In practice, however, we often only sample a relatively small number of sequences using a policy in policy learning. As a result, the sampling method itself matters. Eq. (4.47) offers an interesting manner to separate the sampling and reward computation processes: we first use a baseline policy (with θ_{ref}) to sample a number of sequences, and then use the target policy (with θ) to compute the expected reward. In this way, we separate the policy used for collecting the data, and the policy used for computing the gradient. This approach avoids the need to directly sample from the policy we are evaluating, which can be beneficial in cases where generating sequences from the target policy is expensive or difficult.

In reinforcement learning, $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \right]$ is often called a **surrogate objective**.

Eq. (4.47) can also be interpreted from a policy gradient perspective. For $\mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \right]$, the gradient at $\theta = \theta_{\text{ref}}$ is given by

$$\frac{\partial}{\partial \theta} \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\Pr_{\theta}(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \right] \Big|_{\theta=\theta_{\text{ref}}} = \mathbb{E}_{\tau \sim \pi_{\theta_{\text{ref}}}} \left[\frac{\partial \Pr_{\theta}(\tau) |_{\theta=\theta_{\text{ref}}}}{\partial \theta} R(\tau) \right] \quad (4.48)$$

The right-hand side is a standard form used in policy gradient methods, meaning that we compute the direction of

A problem with the model presented in Eq. (4.47) (as well as in Eq. (4.39)) is that the variance in the gradient estimates is often high, making the learning process unstable. To mitigate this issue, techniques such as clipping are often employed to bound the importance weights and prevent large updates. A clipped version of the utility function (also called the clipped surrogate objective function) is given by

$$U_{\text{clip}}(\tau; \theta) = \sum_{t=1}^T \text{Clip}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) A(s_t, a_t) \quad (4.49)$$

$$\text{Clip}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\right) = \min\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, \text{bound}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon\right)\right) \quad (4.50)$$

Here the function $\text{bound}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon\right)$ constrains the ratio function to the range $[1 - \epsilon, 1 + \epsilon]$.

A further improvement to the above model is to consider **trust regions** in optimization [Schulman et al., 2015]. In reinforcement learning, a large update to the policy can lead to instability, where the agent may start performing worse after an update. A reasonable idea is to optimize the model in the trust region, which refers to a region around the current parameter estimate where the model is well-behaved. One approach to incorporating trust regions is to impose a constraint on the size of the policy update, ensuring that the current policy does not deviate too significantly from the reference policy. This can be achieved by adding a penalty based on some form of divergence between the current and reference policies to the objective function. A simple form of such a penalty is given by the difference in the log-probability of the sequence τ under the current policy versus the reference policy:

$$\text{Penalty} = \log \pi_{\theta}(\tau) - \log \pi_{\theta_{\text{ref}}}(\tau) \quad (4.51)$$

In practice, this penalty can be approximated by considering only the policy probabilities and ignoring the dynamics. This gives

$$\text{Penalty} = \sum_{t=1}^T \log \pi_{\theta}(a_t|s_t) - \sum_{t=1}^T \log \pi_{\theta_{\text{ref}}}(a_t|s_t) \quad (4.52)$$

By including this penalty in the optimization objective, we encourage the current policy to remain close to the reference policy, limiting very large updates that could destabilize learning.

We can incorporate this penalty into the clipped surrogate objective function, and obtain

$$U_{\text{ppo-clip}}(\tau; \theta) = U_{\text{clip}}(\tau; \theta) - \beta \text{Penalty} \quad (4.53)$$

where β is the weight of the penalty. This training method is called **proximal policy optimization (PPO)**, which is one of the most popular reinforcement learning methods used in LLMs and many other fields [Schulman et al., 2017].

Now we can write the objective of training LLMs in the form of PPO.

the parameter update at the point $\theta = \theta_{\text{ref}}$ on the optimization surface.

$$U(\mathbf{x}, \mathbf{y}; \theta) = U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta) - \beta \text{Penalty} \quad (4.54)$$

where

$$U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta) = \sum_{t=1}^T \text{Clip}\left(\frac{\pi_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\pi_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})}\right) A(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (4.55)$$

$$\begin{aligned} \text{Penalty} &= \log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \\ &= \sum_{t=1}^T \log \Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t}) - \sum_{t=1}^T \log \Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t}) \end{aligned} \quad (4.56)$$

Although the notation here appears a bit tedious, the idea of PPO is simple: we develop an objective by combining the clipped likelihood ratio of the target and reference policies with an advantage function, and then impose a penalty that ensures policy updates are not too large. The PPO-based RLHF is illustrated in Figure 4.9.

To summarize, implementing RLHF requires building four models, all based on the Transformer decoder architecture.

- **Reward Model** ($r_{\phi}(\cdot)$ where ϕ denotes the parameters). The reward model learns from human preference data to predict the reward for each pair of input and output token sequences. It is a Transformer decoder followed by a linear layer that maps a sequence (the concatenation of the input and output) to a real-valued reward score.
- **Value Model** or **Value Function** ($V_{\omega}(\cdot)$ where ω denotes the parameters). The value function receives reward scores from the reward model and is trained to predict the expected sum of rewards that can be obtained starting from a state. It is generally based on the same architecture as the reward model.
- **Reference Model** ($\pi_{\theta_{\text{ref}}}(\cdot) = \Pr_{\theta_{\text{ref}}}(\cdot)$ where θ_{ref} denotes the parameters). The reference model is the baseline LLM that serves as a starting point for policy training. In RLHF, it represents the previous version of the model or a model trained without human feedback. It is used to perform sampling over the space of outputs and contribute to the loss computation for policy training.
- **Target Model** or **Policy** ($\pi_{\theta}(\cdot) = \Pr_{\theta}(\cdot)$ where θ denotes the parameters). This policy governs how the LLM decides the most appropriate next token given its context. It is trained under the supervision of both the reward model and the value model.

In practice, these models need to be trained in a certain order. First, we need to initialize them using some other models. For example, the reward model and the value model can be initialized with a pre-trained LLM, while the reference model and the target model can be initialized with a model that has been instruction fine-tuned. Note that, at this point, the reference model is ready for use and will not be further updated. Second, we need to collect human preference data and train the reward model on this data. Third, both the value model and the policy are trained simultaneously using the reward model. At each position in an output token sequence, we update the value model

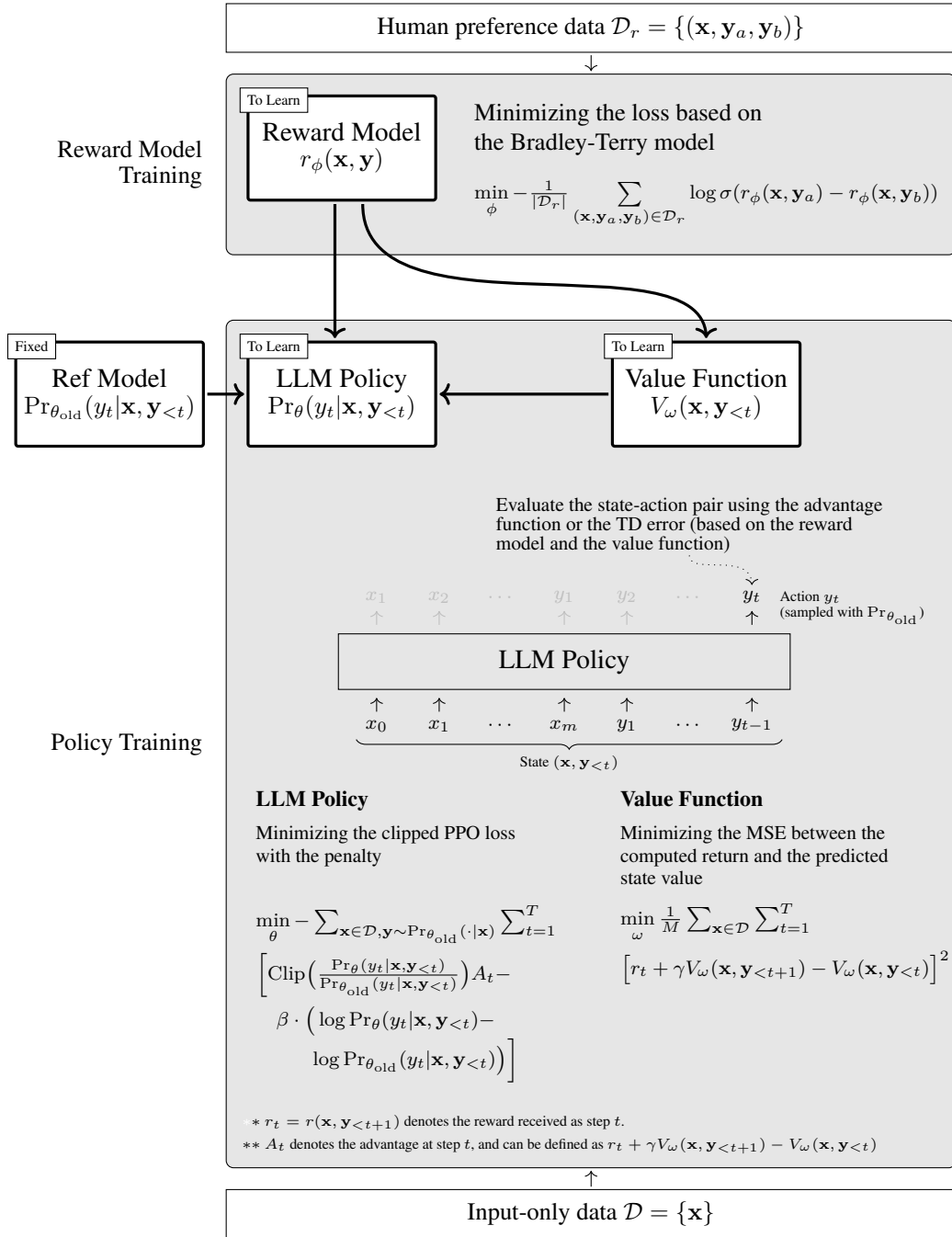


Fig. 4.9: Illustration of RLHF. The first step is to collect human preference data and train the reward model using this data. Once the reward model is optimized, along with the reference model, we proceed to train both the policy and the value function. At each prediction step, we compute the sum of the PPO-based loss and update the parameters of the policy. This requires access to the reward model, the reference model, and the value function at hand. At the same time, we update the parameters of the value function by minimizing the MSE loss.

by minimizing the MSE error of value prediction, and the policy is updated by minimizing the PPO loss.

4.4 Improved Human Preference Alignment

In the previous section, we reviewed the basic concepts of reinforcement learning and the general framework of RLHF. In this section, we will discuss some refinements of RLHF and alternative methods to achieve human preference alignment.

4.4.1 Better Reward Modeling

In Section 4.3.2, we highlighted the task of learning from human preferences as well as the use of pairwise ranking loss for training reward models. Here we consider more methods for reward modeling. Our discussion will be relatively general, and since the reward model is widely used in many reinforcement learning problems, it will be easy for us to apply the methods discussed here to RLHF and related applications.

4.4.1.1 Supervision Signals

The training of reward models can broadly be seen as a ranking problem, where the model learns to assign scores to outputs so that their order reflects the preferences indicated by humans. There are several methods to train a reward model from the perspective of ranking.

One approach is to extend pairwise ranking to listwise ranking. For each sample in a dataset, we can use the LLM to generate multiple outputs, and ask human experts to order these outputs. For example, given a set of four outputs $\{y_1, y_2, y_3, y_4\}$, one possible order of them can be $y_2 \succ y_3 \succ y_1 \succ y_4$. A very simple method to model the ordering of the list is to accumulate the pairwise comparison loss. For example, we can define the listwise loss by accumulating the loss over all pairs of outputs:

$$\mathcal{L}_{\text{list}} = -\mathbb{E}_{(\mathbf{x}, Y) \sim \mathcal{D}_r} \left[\frac{1}{N(N-1)} \sum_{\substack{y_a \in Y, y_b \in Y \\ y_a \neq y_b}} \log \Pr(y_a \succ y_b | \mathbf{x}) \right] \quad (4.57)$$

where Y is a list of outputs, and N is the number of outputs in the list. $\Pr(y_a \succ y_b | \mathbf{x})$ can be defined using the Bradley-Terry model, that is, $\Pr(y_a \succ y_b | \mathbf{x}) = \text{Sigmoid}(r(\mathbf{x}, y_a) - r(\mathbf{x}, y_b))$. Here we omit the ϕ superscript on the $\Pr(\cdot)$ to keep the notation uncluttered.

An extension to the Bradley-Terry model for listwise ranking could involve a ranking mechanism that takes into account the entire list of outputs rather than just pairwise comparisons. One such model is the **Plackett-Luce model**, which generalizes the Bradley-Terry model to handle multiple items in a ranking [Plackett, 1975]. In the Plackett-Luce model, for each item in a list, we define a “worth” for this item that reflects its relative strength of being chosen over other items. For the reward modeling problem here, the worth of y in the list Y can be defined as

$$\alpha(y) = \exp(r(\mathbf{x}, y)) \quad (4.58)$$

Then the probability of selecting \mathbf{y} from Y is given by

$$\begin{aligned}\Pr(\mathbf{y} \text{ is selected} | \mathbf{x}, Y) &= \frac{\alpha(\mathbf{y})}{\sum_{\mathbf{y}' \in Y} \alpha(\mathbf{y}')} \\ &= \frac{\exp(r(\mathbf{x}, \mathbf{y}))}{\sum_{\mathbf{y}' \in Y} \exp(r(\mathbf{x}, \mathbf{y}'))}\end{aligned}\quad (4.59)$$

Suppose \mathring{Y} is an ordered list $\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \dots \succ \mathbf{y}_{j_N}$. The overall log-probability of this ordered list can be defined as the sum of the conditional log-probabilities at each stage of selection, given by

$$\begin{aligned}\log \Pr(\mathring{Y} | \mathbf{x}) &= \log \Pr(\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \dots \succ \mathbf{y}_{j_N} | \mathbf{x}) \\ &= \log \Pr(\mathbf{y}_{j_1} | \mathbf{x}, \{\mathbf{y}_{j_1}, \mathbf{y}_{j_2}, \dots, \mathbf{y}_{j_N}\}) + \\ &\quad \log \Pr(\mathbf{y}_{j_2} | \mathbf{x}, \{\mathbf{y}_{j_2}, \dots, \mathbf{y}_{j_N}\}) + \\ &\quad \dots + \\ &\quad \log \Pr(\mathbf{y}_{j_N} | \mathbf{x}, \{\mathbf{y}_{j_N}\}) \\ &= \sum_{k=1}^N \log \Pr(\mathbf{y}_{j_k} | \mathbf{x}, \mathring{Y}_{\geq k})\end{aligned}\quad (4.60)$$

where $\mathring{Y}_{\geq k}$ represents the subset of the list of outputs that remain unselected at the k -th stage, i.e., $\mathring{Y}_{\geq k} = \{\mathbf{y}_{j_k}, \dots, \mathbf{y}_{j_N}\}$. Given the log-probability $\log \Pr(\mathring{Y} | \mathbf{x})$, we can define the loss function based on the Plackett-Luce model by

$$\mathcal{L}_{\text{pl}} = -\mathbb{E}_{(\mathbf{x}, \mathring{Y}) \sim \mathcal{D}_r} [\log \Pr(\mathring{Y} | \mathbf{x})] \quad (4.61)$$

There are also many other pairwise and listwise methods for modeling rankings, such as RankNet [Burges et al., 2005] and ListNet [Cao et al., 2007]. All these methods can be categorized into a large family of learning-to-rank approaches, and most of them are applicable to the problem of modeling human preferences. However, discussing these methods is beyond the scope of this chapter. Interested readers can refer to books on this topic for more details [Liu, 2009; Li, 2011].

In addition to pairwise and listwise ranking, using pointwise methods to train reward models offers an alternative way to capture human preferences. Unlike methods that focus on the relative rankings between different outputs, pointwise methods treat each output independently. For example, human experts might assign a score to an individual output, such as a rating on a five-point scale. The objective is to adjust the reward model so that its outputs align with these scores. A simple way to achieve pointwise training is through regression techniques where the reward of each output is treated as a target variable. Let $\varphi(\mathbf{x}, \mathbf{y})$ be the score assigned to \mathbf{y} given \mathbf{x} by humans. Pointwise reward models can be trained by minimizing a loss function, often based on mean squared error or other regression losses, between the predicted reward $r(\mathbf{x}, \mathbf{y})$ and the actual human feedback $\varphi(\mathbf{x}, \mathbf{y})$. For example, the loss function could be

$$\mathcal{L}_{\text{point}} = -\mathbb{E}[\varphi(\mathbf{x}, \mathbf{y}) - r(\mathbf{x}, \mathbf{y})]^2 \quad (4.62)$$

While pointwise methods are conceptually simpler and can directly guide the reward model to

predict scores, they might not always be the best choice in RLHF. A problem is that these methods may struggle with high variance in human feedback, especially when different experts provide inconsistent scores for similar outputs. Because they focus on fitting to absolute scores rather than relative differences, inconsistencies in scoring can lead to poor model performance. Moreover, fitting to specific scored outputs might discourage generalization, particularly given that training data is often very limited in RLHF. In contrast, methods that consider relative preferences can promote the learning of more generalized patterns of success and failure. Nevertheless, there are scenarios where pointwise methods might still be suitable. For example, in tasks where training data is abundant and the costs of obtaining accurate, consistent annotations are low, pointwise methods can prove effective.

In fact, to make the supervision signal for training the reward model more robust, we can also introduce additional regularization terms into training. For example, if we consider the first term $U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta)$ in Eq. (4.54) as a type of generalized reward, then the second term (i.e., the penalty term) can be viewed as a form of regularization for the reward model, except that here the goal is to train the policy rather than the reward model. Another example is that [Eisenstein et al. \[2023\]](#) develop a regularization term based on the squared sum of rewards, and add it to the pairwise comparison loss in RLHF:

$$\begin{aligned}\mathcal{L}_{\text{reg}} &= \mathcal{L}_{\text{pair}} + (-\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)]^2) \\ &= -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_{\phi}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \\ &\quad - \mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)]^2\end{aligned}\tag{4.63}$$

Optimizing with this regularization term can help mitigate the underdetermination of reward models⁵.

4.4.1.2 Sparse Rewards vs. Dense Rewards

As discussed in Section 4.3, the rewards in RLHF are very sparse: they are observed only at the end of sequences, rather than continuously throughout the generation process. Dealing with sparse rewards has long been a concern in reinforcement learning, and has been one of the challenges in many practical applications. For example, in robotics, it often needs to shape the reward function to ease optimization rather than relying solely on end-of-sequence rewards. Various methods have been developed to address this issue. One common approach is reward shaping, where the original function is modified to include intermediate rewards, thereby providing more immediate feedback. Also, one can adopt curriculum learning to sequentially structure tasks in a way that the complexity gradually increases. This can help models to master simpler tasks first, which prepares them for more complex challenges as their skills develop. There are many such methods that can mitigate the impact of sparse rewards, such as Monte Carlo methods and intrinsic motivation. Most of these methods are general and the discussion of them can be found in the broader literature on reinforcement learning, such as [Sutton and Barto \[2018\]](#)’s book.

Although we do not discuss methods for mitigating sparse rewards in detail here, an interesting question arises: why are sparse rewards so successful in RLHF? Recall from Section 4.3.1 that the supervision signal received at each time step t is not the reward for the current action, but

⁵A model is called underdetermined if there are multiple alternative sets of parameters that can achieve the same objective.

rather some form of the accumulated rewards from t until the last time step. Such supervision signals are dense over the sequence, because the reward obtained at the end of the sequence can be transferred back to that time step, regardless of which time step it is. In other words, the sparse rewards are transformed into the dense supervision signals. Furthermore, from the perspective of reward shaping, [Ng et al. \[1999\]](#) show that the reward at t can be defined as

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t, s_{t+1}) + f(s_t, a_t, s_{t+1}) \quad (4.64)$$

where $r'(\cdot)$ is the transformed reward function, $r(\cdot)$ is the original reward function, and $f(\cdot)$ is the shaping reward function. To ensure the optimality of the policy under the transformed reward function, the shaping reward function can be given in the form

$$f(s_t, a_t, s_{t+1}) = \gamma\Phi(s_{t+1}) - \Phi(s_t) \quad (4.65)$$

where $\Phi(s)$ is called the potential value of the state s . If we define $\Phi(s)$ as the common value function as in [Eq. \(4.15\)](#) and substitute [Eq. \(4.65\)](#) into [Eq. \(4.64\)](#), we obtain

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1}) - V(s_t) \quad (4.66)$$

It is interesting to see that this function is exactly the same as the advantage function used in PPO. This relates advantage-based methods to reward shaping: the advantage is essentially a shaped reward.

On the other hand, one of the reasons for adopting end-of-sequence rewards lies in the nature of the RLHF tasks. Unlike traditional reinforcement learning environments where the agent interacts with a dynamic environment, RLHF tasks often involve complex decision-making based on linguistic or other high-level cognitive processes. These processes do not lend themselves easily to frequent and meaningful intermediate rewards because the quality and appropriateness of the actions can only be fully evaluated after observing their impact in the larger context of the entire sequence or task. In this case, the reward signals based on human feedback, though very sparse, are typically very informative and accurate. Consequently, this sparsity, together with the high informativeness and accuracy of the human feedback, can make the learning both robust and efficient.

4.4.1.3 Fine-grained Rewards

For many applications, our objective will be more complex than merely evaluating an entire text. For example, in sentiment analysis, we often do not just determine the sentiment of a text, but need to analyze the sentiment in more detail by associating it with specific aspects of a topic discussed in the text. Consider the sentence "The camera of the phone is excellent, but the battery life is disappointing." In this example, we would need to separately analyze the sentiments expressed about the camera and the battery. Such analysis, known as aspect-based sentiment analysis, helps provide a finer-grained understanding of the customer review compared to general sentiment analysis.

For the problem of reward modeling, we often need to model different parts of a sequence as well. A simple and straightforward way to do this is to divide a sequence into different segments and then compute the reward for each segment [[Wu et al., 2023b](#)]. Suppose that an output token

sequence \mathbf{y} can be divided into n_s segments $\{\bar{\mathbf{y}}_1, \dots, \bar{\mathbf{y}}_{n_s}\}$ by some criterion. We can use the reward model to evaluate each of these segments. By taking \mathbf{x} , \mathbf{y} and $\bar{\mathbf{y}}_k$ as input to the reward model, the reward score for the k -th segment is given by

$$r^k = r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \quad (4.67)$$

Then the reward score for the entire output sequence is given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \quad (4.68)$$

Here $r(\mathbf{x}, \mathbf{y})$ can be used to train the policy as usual.

A problem with this model is that training reward models at the segment level is not as straightforward as learning from human preferences on entire texts, as it is difficult to obtain segment-level human preference data. For rating-like problems (e.g., we rate a segment according to its level of misinformation), one simple approach is to assign a rating score to each segment and train the reward model using pointwise methods. For example, we can use a strong LLM to rate the sequences $\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_{k-1}$ and $\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_k$, and obtain the scores $s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_{k-1})$ and $s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_k)$. We can then define the score of the segment $\bar{\mathbf{y}}_k$ as the difference between $s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_k)$ and $s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_{k-1})$

$$s(\bar{\mathbf{y}}_k) = s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_k) - s(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_{k-1}) \quad (4.69)$$

Using these segment-level scores, we can train the reward model with a regression loss function

$$\mathcal{L}_{\text{rating}} = -\mathbb{E}_{\bar{\mathbf{y}}_k} [s(\bar{\mathbf{y}}_k) - r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k)]^2 \quad (4.70)$$

Sometimes, alignment can be treated as a classification problem, for example, we assess whether a segment has ethical issues. In this case, the segment can be labeled as ethical or unethical, either by humans or using additional classifiers. Given the label of the segment, we can train the reward model using some classification loss function. For example, suppose that $r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) = 1$ if the segment is classified as unethical, and $r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) = -1$ otherwise⁶. The hinge loss of training binary classification models is given by

$$\mathcal{L}_{\text{hinge}} = \max(0, 1 - r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \cdot \hat{r}) \quad (4.71)$$

where $\hat{r} \in \{1, -1\}$ denotes the ground truth label.

The remaining issue here is how to split \mathbf{y} into segments. One approach is to define a fixed-length segmentation, where \mathbf{y} is divided into equal-length chunks. However, this may not always be ideal, as the content of the sequence may not align well with fixed boundaries. An alternative approach is to segment \mathbf{y} based on specific linguistic or semantic cues, such as sentence boundaries, topic shifts, or other meaningful structures in the text. Such a segmentation can be achieved by using linguistic segmentation systems or prompting LLMs to identify natural breaks in the sequence. Another approach is to use dynamic segmentation methods based on the complexity of

⁶To allow the reward model to output categories, we can replace the linear layer described in Section 4.3.2 with a Softmax layer.

the sequence. For example, segments could be defined where there is a significant change in the reward score, which might correspond to shifts in the task being modeled.

4.4.1.4 Combination of Reward Models

A reward model can be viewed as a proxy for the environment. Since the true environment is often too complex or unknown, developing a perfect proxy for the environment is generally not possible. As a result, over-aligning LLMs with this imperfect proxy might lead to decreased performance, known as the **overoptimization problem** [Stiennon et al., 2020; Gao et al., 2023a]⁷. We can also explain this through Goodhart’s law, which states: *when a measure becomes a target, it ceases to be a good measure* [Goodhart, 1984].

Addressing the overoptimization problem is not easy, and there is no mature solution yet. The ideal approach might be to develop an oracle reward model that can perfectly capture the true objectives of the task and prevent the agent from “tricking”. However, creating such a model is extremely difficult due to the complexity of the real-world environment, as well as the challenge of defining all the relevant factors that contribute to the desired outcome. Instead, a more practical approach is to combine multiple reward models, thereby alleviating the misalignment between the training objective and the true objective that arises from using a single, specific reward model [Coste et al., 2024].

Given a set of reward models, combining them is straightforward, and in some cases, we can simply treat this problem as an ensemble learning problem. A simple yet common approach is to average the outputs of these models to obtain a more precise reward estimation:

$$r_{\text{combine}} = \frac{1}{K} \sum_{k=1}^K w_k \cdot r_k(\mathbf{x}, \mathbf{y}) \quad (4.72)$$

where $r_k(\cdot)$ is the k -th reward model in the ensemble, w_k is the weight of $r_k(\cdot)$, and K is the number of reward models. This combined reward can then be used to supervise the training of a policy. In fact, there are many ways to combine different models, for example, one can make predictions using Bayesian model averaging or develop a fusion network to learn to combine the predictions from different models. Alternatively, one can frame this task as a multi-objective optimization problem, and use multiple reward models to train the policy simultaneously. These methods have been intensively discussed in the literature on optimization and machine learning [Miettinen, 1999; Bishop, 2006].

In addition to model combination methods, another important issue is how to collect or construct multiple different reward models. One of the simplest approaches is to employ ensemble learning techniques, such as developing diverse reward models from different subsets of a given dataset or from various data sources. For RLHF, it is also possible to construct reward models based on considerations of different aspects of alignment. For example, we can develop a reward model to evaluate the factual accuracy of the output and another reward model to evaluate the

⁷This problem is also called **reward hacking** or **reward gaming** [Krakovna et al., 2020; Skalse et al., 2022; Pan et al., 2022], which refers to the phenomenon where the agent attempts to trick the reward model but fails to align its actions with the true intended objectives of the task. Imagine a student who is assigned homework and is rewarded with points or praise for completing it. The student might then find ways to finish the homework with minimal effort to maximize the reward, such as copying and pasting solutions from the internet or previous assignments, rather than solving the problems themselves.

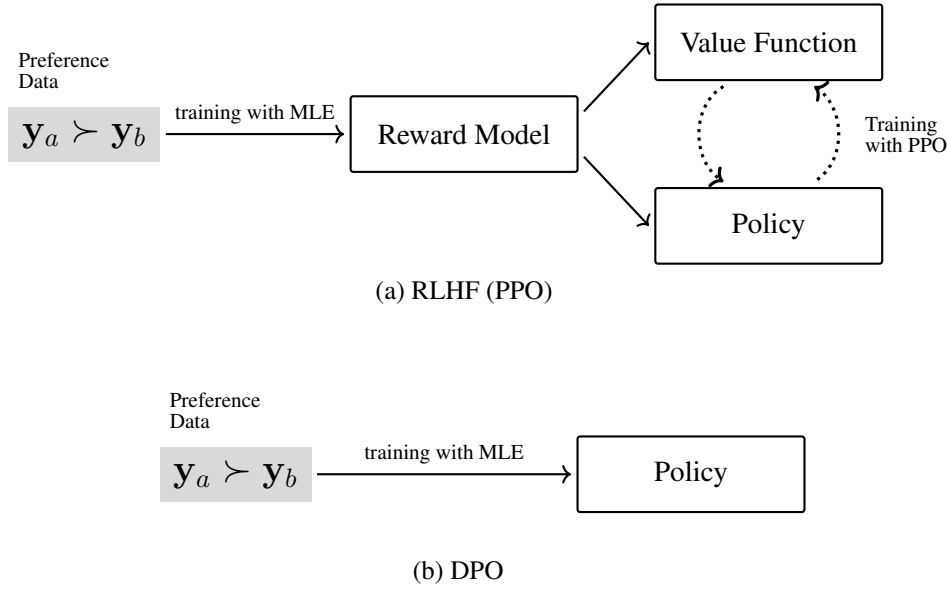


Fig. 4.10: Standard RLHF (PPO) vs. DPO. In RLHF, the human preference data is used to train a reward model, which is then employed in training the policy as well as the value function. In DPO, the use of human preference data is more direct, and the policy is trained on this data without the need for reward model training.

completeness of the output. These two models are complementary to each other, and can be combined to improve the overall evaluation of the output. Another approach is to employ different off-the-shelf LLMs as reward models. This approach is simple and practical, as there have been a lot of well-developed LLMs and we just need to use them with no or little modification. An interesting issue, though not closely related to the discussion here, arises: can an LLM that aligns with other LLMs outperform those LLMs? Probably not at first glance. In part, this is because the target LLM merely imitates other LLMs based on limited supervision and thus cannot capture well the nuances of the behaviors of these supervisors. However, given the strong generalization ability of LLMs, this approach can, in fact, be quite beneficial. For example, using open-sourced or commercial LLMs as reward models has demonstrated strong performance in aligning LLMs, even achieving state-of-the-art results on several popular tasks [Lambert et al., 2024].

4.4.2 Direct Preference Optimization

Although learning reward models is a standard step in reinforcement learning, it makes the entire training process much more complex than supervised training. Training a reliable reward model is itself not an easy task, and a poorly trained reward model can greatly affect the outcome of policy learning. We now consider an alternative alignment method, called **direct preference optimization (DPO)**, which simplifies the training framework by eliminating the need to explicitly model rewards [Rafailov et al., 2024]. This method directly optimizes the policy based on user preferences, rather than developing a separate reward model. As a result, we can achieve human preference alignment in a supervised learning-like fashion. Figure 4.10 shows a comparison of the standard RLHF method and the DPO method.

Before deriving the DPO objective, let us first review the objective of policy training used in RLHF. As discussed in Section 4.3.3, the policy is typically trained by optimizing a loss function

with a penalty term. The DPO method assumes a simple loss function where the quality of the output \mathbf{y} given the input \mathbf{x} is evaluated by the reward model $r(\mathbf{x}, \mathbf{y})$. The training objective is thus given by

$$\tilde{\theta} = \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[\underbrace{-r(\mathbf{x}, \mathbf{y})}_{\text{loss}} + \beta \underbrace{(\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}))}_{\text{penalty}} \right] \quad (4.73)$$

Note that in this optimization problem, only the term $\pi_{\theta}(\mathbf{y}|\mathbf{x})$ depends on the target policy $\pi_{\theta}(\cdot)$. Both the reward model $r(\mathbf{x}, \mathbf{y})$ and the reference model $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$ are assumed to be fixed given \mathbf{x} and \mathbf{y} . This is a strong assumption compared with PPO, but as will be shown later, it simplifies the problem and crucial for deriving the DPO objective.

Since θ is the variable we want to optimize, we rearrange the right-hand side of Eq. (4.73) to isolate $\pi_{\theta}(\mathbf{y}|\mathbf{x})$ as an independent term:

$$\begin{aligned} \tilde{\theta} &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} [\beta \log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \beta \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) - r(\mathbf{x}, \mathbf{y})] \\ &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} [\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - (\log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) + \frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))] \\ &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[\underbrace{\log \pi_{\theta}(\mathbf{y}|\mathbf{x})}_{\text{dependent on } \theta} - \underbrace{\log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))}_{\text{not dependent on } \theta} \right] \quad (4.74) \end{aligned}$$

This equation defines the objective function as the difference between the log-probability distribution function of y and another function of y . This form of the objective function seems not “ideal”, as we usually prefer to see the difference between two distributions, so that we can interpret this difference as some kind of divergence between the distributions. A simple idea is to convert the second term (i.e., $\log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))$) into a log-probability distribution over the domain of \mathbf{y} . If we treat $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))$ as an unnormalized probability of y , we can convert it into a normalized probability by dividing it by a normalization factor:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})) \quad (4.75)$$

Hence we can define a probability distribution by

$$\pi^*(\mathbf{y}|\mathbf{x}) = \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y}))}{Z(\mathbf{x})} \quad (4.76)$$

We then rewrite Eq. (4.74) as

$$\begin{aligned}
\tilde{\theta} &= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}{Z(\mathbf{x})} \right. \\
&\quad \left. - \log Z(\mathbf{x}) \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi^*(\mathbf{y}|\mathbf{x}) - \log Z(\mathbf{x}) \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[\mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} \left[\log \pi_{\theta}(\mathbf{y}|\mathbf{x}) - \log \pi^*(\mathbf{y}|\mathbf{x}) \right] \right. \\
&\quad \left. - \mathbb{E}_{\mathbf{y} \sim \pi_{\theta}(\cdot|\mathbf{x})} [\log Z(\mathbf{x})] \right] \\
&= \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} \left[\underbrace{\text{KL}(\pi_{\theta}(\cdot|\mathbf{x}) \parallel \pi^*(\cdot|\mathbf{x}))}_{\text{KL divergence}} - \underbrace{\log Z(\mathbf{x})}_{\text{constant wrt. } \theta} \right] \tag{4.77}
\end{aligned}$$

Since $\log Z(\mathbf{x})$ is independent of θ , it does not affect the result of the $\arg \min_{\theta}$ operation, and can be removed from the objective. Now we obtain a new training objective which finds the optimal policy π_{θ} by minimizing the KL divergence between $\pi_{\theta}(\cdot|\mathbf{x})$ and $\pi^*(\cdot|\mathbf{x})$

$$\tilde{\theta} = \arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \mathcal{D}} [\text{KL}(\pi_{\theta}(\cdot|\mathbf{x}) \parallel \pi^*(\cdot|\mathbf{x}))] \tag{4.78}$$

Clearly, the solution to this optimization problem is given by

$$\begin{aligned}
\pi_{\theta}(\mathbf{y}|\mathbf{x}) &= \pi^*(\mathbf{y}|\mathbf{x}) \\
&= \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \exp\left(\frac{1}{\beta} r(\mathbf{x}, \mathbf{y})\right)}{Z(\mathbf{x})} \tag{4.79}
\end{aligned}$$

Given this equation, we can express the reward $r(\mathbf{x}, \mathbf{y})$ using the target model $\pi_{\theta}(\mathbf{y}|\mathbf{x})$, the reference model $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$, and the normalization factor $Z(\mathbf{x})$:

$$r(\mathbf{x}, \mathbf{y}) = \beta \left(\log \frac{\pi_{\theta}(\mathbf{y}|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})} + \log Z(\mathbf{x}) \right) \tag{4.80}$$

This is interesting because we initially seek to learn the policy $\pi_{\theta}(\cdot)$ using the reward model $r(\mathbf{x}, \mathbf{y})$, but eventually obtain a representation of the reward model based on the policy. Given the reward model defined in Eq. (4.80), we can apply it to the Bradley-Terry model to calculate the preference probability (also see Section 4.3.2):

$$\begin{aligned}
\Pr_{\theta}(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) &= \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)) \\
&= \text{Sigmoid} \left(\beta \left(\log \frac{\pi_{\theta}(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} + \log Z(\mathbf{x}) \right) - \right. \\
&\quad \left. \beta \left(\log \frac{\pi_{\theta}(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})} + \log Z(\mathbf{x}) \right) \right) \\
&= \text{Sigmoid} \left(\beta \log \frac{\pi_{\theta}(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} - \beta \log \frac{\pi_{\theta}(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})} \right) \tag{4.81}
\end{aligned}$$

This formula is elegant because it converts the difference in rewards into the difference in ratio functions, and we do not need to calculate the value of $Z(\mathbf{x})$. A direct result is that we no longer need a reward model, but only need the target policy and reference model to calculate the probability of preferences. Finally, we can train the target policy by minimizing the following DPO loss function

$$\mathcal{L}_{\text{dpo}}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_{\theta}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \quad (4.82)$$

The form of this loss function is very similar to that used in training reward models in RLHF (see Eq. (4.36)). But it should be noted that the loss function here depends on the parameters of the policy (i.e., θ) rather than the parameters of the reward model (i.e., ϕ).

The main advantage of DPO lies in its simplicity and efficiency. The DPO objective is very straightforward — it directly optimizes for preference-based feedback, rather than relying on separately developed reward models. Moreover, DPO is generally more sample-efficient, as it learns from a fixed dataset without the need for the computationally expensive sampling process used in PPO. This makes DPO a popular method for human preference alignment, especially when developing and applying reward models via reinforcement learning is challenging.

DPO can broadly be viewed as an **offline reinforcement learning** method, where the training data is pre-collected and fixed, and there is no exploration. In contrast, online reinforcement learning methods like PPO, which require exploring new states through interaction with the environment (using the reward model as a proxy), also have their unique advantages. One of the benefits of online reinforcement learning is that it allows the agent to continuously adapt to changes in the environment by learning from real-time feedback. This means that, unlike offline methods, online methods are not constrained by the static nature of pre-collected data and can discover new problem-solving strategies. In addition, exploration can help the agent cover a wider range of state-action pairs, thus improving generalization. This could be an important advantage for LLMs, as generalization is considered a critical aspect in applying such large models.

4.4.3 Automatic Preference Data Generation

Although learning from human preferences is an effective and popular method for aligning LLMs, annotating preference data is costly. Using human feedback does not only faces the problem of limited scalability, but it may also introduce bias because human feedback is inherently subjective. As a result, one can turn to AI feedback methods to address these scalability and consistency issues without the limitations associated with human annotators.

As with data generation for instruction fine-tuning, generating preference data using LLMs is straightforward. Given a set of inputs, we first use an LLM to generate pairs of outputs. Then, we prompt the LLM to label the preference between each pair of outputs, along with its corresponding input. Below is an example of prompting the LLM to generate a preference label for a pair of consumer service responses.

Consider a customer service scenario where a customer poses a request. You will review two responses to this request. Please indicate which response is preferred. Note that a good response should be courteous, clear, and concise. It should address the customer’s concern directly, provide helpful information or a solution, and maintain a positive tone.

Request:

Hello, I noticed that my order hasn’t arrived yet, though it was scheduled to arrive several days ago. Could you please update me on its status? Thank you!

Response A:

I’m very sorry for the delay and understand how disappointing this can be. We’re doing our best to sort this out quickly for you.

Response B:

Hey, stuff happens! Your package will get there when it gets there, no need to stress.

Response A is preferred.

Once we collect such preference labels, we can use them, along with the output pair and input, to train the reward model. Of course, we can consider demonstrating a few examples or using advanced prompting techniques, such as CoT, to improve labeling performance. For example, we can include in the prompt an example showing how and why one of the two responses is preferred based on a CoT rationale.

In addition to preference labels, we can also obtain the probability associated with each label [Lee et al., 2023]. A simple method is to extract the probabilities for the label tokens, such as “A” and “B”, from the probabilities output by the LLM. We can then use the Softmax function or other normalization techniques to re-normalize these probabilities into a distribution over the labels. These probabilities of preferred labels can serve as pointwise supervision signals for training the reward model, as discussed in Section 4.4.1.

For data generation, although it is easy to scale up, it is often necessary to ensure the data is accurate and diverse. Here, the data quality and diversity issues involve not only the labeling of preferences but also the inputs and outputs of the model. Therefore, we often need to use a variety of techniques to obtain large-scale, high-quality data. For example, one can generate diverse model outputs and annotations by using different LLMs, prompts, in-context demonstrations, and so on [Cui et al., 2024]. Dubois et al. [2024] report that the variability in pairwise preference data is important for training LLMs from either human or AI feedback.

While learning from AI feedback is highly scalable and generally objective, this method is more suited to well-defined tasks where objective performance metrics are available. By contrast, learning from human feedback is more advantageous when aligning AI systems with human values, preferences, and complex real-world tasks that require understanding of subtle or subjective context. These methods can be combined to train LLMs that benefit from both human insights and the scalability of AI feedback.

4.4.4 Step-by-step Alignment

So far, our discussion of alignment has primarily focused on the use of reward models for evaluating entire input-output sequence pairs. These methods can be easily adapted to scenarios where the correctness of an output can be examined by checking whether the desired result is included. For example, in the task of calculating a mathematical expression, a reward model can provide positive feedback if the answer is correct, and negative feedback if the answer is wrong. However, in many problems that require complex reasoning, simply examining the correctness of the final result is insufficient for learning. Imagine a student who is only given the final answer to a challenging math problem. Knowing whether the final answer is right or wrong does not help the student figure out where they went wrong and how to calculate the correct answer. A better approach would be to guide the student with a step-by-step breakdown of the problem-solving process and encourage understanding of the underlying concepts and logic behind these steps.

In Chapter 3, we studied CoT methods to prompt LLMs to explicitly write out intermediate steps or the reasoning process needed to reach a conclusion or solve a problem. We saw that breaking down a problem into smaller parts could make it easier to understand the solution path and increase the accuracy of the output. These methods can be naturally extended to the alignment of LLMs, that is, we supervise the model during the intermediate steps of reasoning. Consider a reasoning task where an LLM produces a sequence of reasoning steps $\mathbf{y} = \{\bar{\mathbf{y}}_1, \dots, \bar{\mathbf{y}}_{n_s}\}$ for the given input. The result of the reasoning is assumed to be included in the last step $\bar{\mathbf{y}}_{n_s}$, and can be easily verified. For this reasoning problem, Uesato et al. [2022] categorize LLM fine-tuning approaches into two classes:

- **Outcome-based Approaches.** Supervision occurs only when the end result is verified. This is a standard method for learning from human feedback we have discussed in this chapter. For example, the LLM is optimized to maximize some form of the reward $r(\mathbf{x}, \mathbf{y})$.
- **Process-based Approaches.** Supervision is involved in all intermediate steps in addition to the last step. To do this, we need to develop a model to give a supervision signal at each step, and develop loss functions that can make use of such supervision signals.

Figure 4.11 shows two LLM outputs for an example math problem. Although the LLM gives the correct final answer in both cases, it makes mistakes during the problem-solving process in the second output. Outcome-based approaches overlook these mistakes and give positive feedback for the entire solution. By contrast, process-based approaches can take these mistakes into account and provide additional guidance on the detailed reasoning steps.

An important issue for process-based approaches is that we need to get step-level feedback during a (potentially) long reasoning path. We can collect or generate reasoning paths corresponding to problems from existing datasets. Human experts then annotate each step in these paths for correctness. These annotations can be used to directly train LLMs or as rewards in reward modeling. However, in practice, richer annotations are often introduced [Lightman et al., 2024]. In addition to the *correct* and *incorrect* labels, a step can also be labeled as *neutral* to indicate that while the step may be technically correct, it might still be problematic within the overall reasoning process. Furthermore, to improve the efficiency of data annotation, techniques such as active learning can be employed. Identifying obvious errors usually does not significantly contribute to

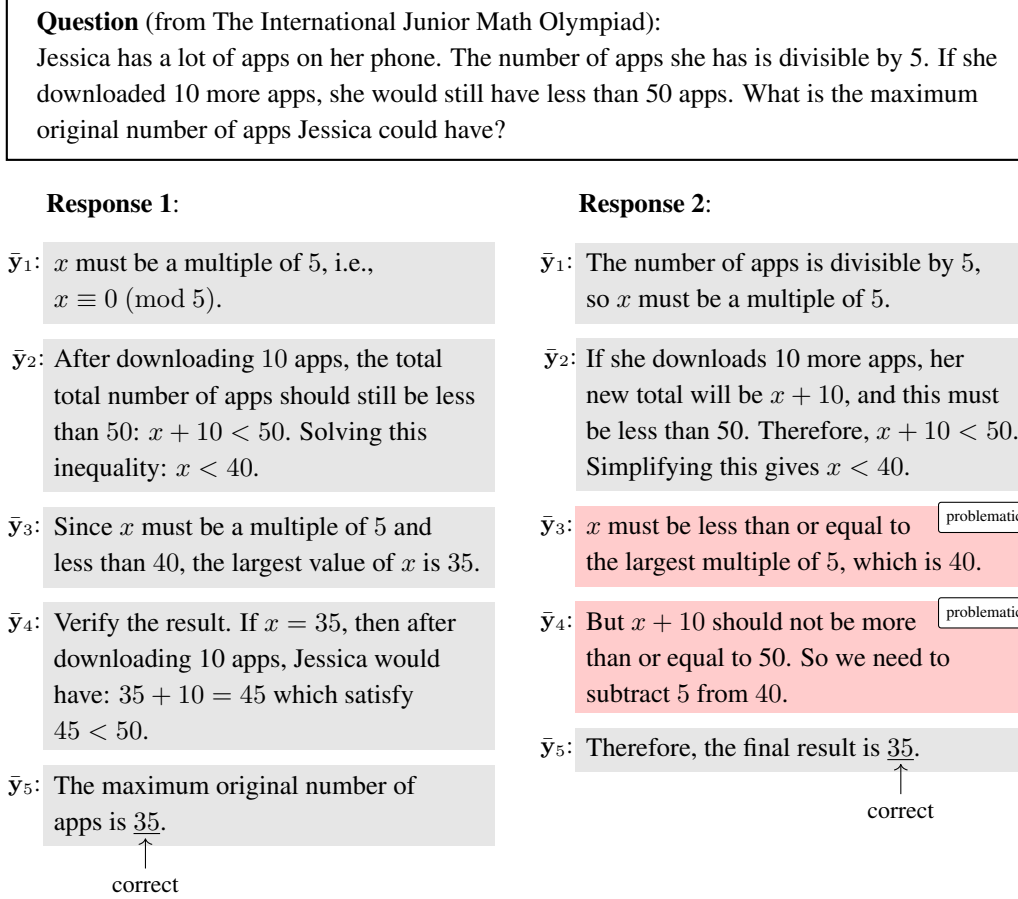


Fig. 4.11: Two LLM responses to a math problem. In response 1, both the final result and all the reasoning steps are correct. In response 2, the final result is correct, but there are mistakes in the reasoning process (highlighted in red). For outcome-based approaches, both responses are considered correct. For process-based approaches, the mistakes in response 2 can be considered in reward modeling.

learning from reasoning mistakes. Instead, annotating steps that the model confidently considers correct but are actually problematic is often more effective.

Given a set of step-level annotated reasoning paths and corresponding inputs, we can train a reward model to provide feedback for supervising policy learning. The reward model can be treated as a classification model, and so its architecture can be a Transformer decoder with a Softmax layer stacked on top. At step k , the reward model takes both the problem description (denoted by \mathbf{x}) and the reasoning steps generated so far (denoted by $\bar{\mathbf{y}}_{\leq k}$) as input and outputs a probability distribution over the label set $\{correct, incorrect\}$ or $\{correct, incorrect, neutral\}$. Then the learned reward model is used to evaluate reasoning paths by assessing the correctness of each step. A simple method to model correctness is to count the number of steps that are classified as *correct*, given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \delta(correct, C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})) \quad (4.83)$$

where $C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ denotes the label with the maximum probability. We can also use log-probabilities

of classification to define the reward of the entire path

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \log \Pr(\text{correct} | \mathbf{x}, \bar{\mathbf{y}}_{\leq k}) \quad (4.84)$$

where $\Pr(\text{correct} | \mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ denotes the probability of the *correct* label generated by the reward model. The reward score $r(\mathbf{x}, \mathbf{y})$ can then be used to train the policy in RLHF as usual.

While we restrict our discussion to math problems, the approaches described here are general and can be applied to a wide variety of tasks that involve multi-step reasoning and decision-making. Moreover, we can consider various aspects when assessing the quality of a step, rather than just its correctness. For example, in dialogue systems, responses must not only be accurate but also contextually appropriate across multiple turns of conversation. If a model provides a correct response but fails to maintain coherence in the context of the ongoing dialogue, step-level feedback could help the model identify and correct such discrepancies. Also note that the process-based approaches are related to the fine-grained reward modeling approaches discussed in Section 4.4.1.3. All these approaches essentially aim to provide more detailed supervision to LLMs by breaking their outputs into smaller, more manageable steps. However, process-based feedback focuses more on evaluating the correctness of a step based on its preceding steps, while the approaches in Section 4.4.1.3 emphasize evaluating each step independently.

The idea of aligning LLMs step by step has great application potential, especially considering the recent shift towards more complex reasoning tasks in the use of LLMs. For example, both the GPT-o1 and GPT-o3 models are designed with more advanced reasoning techniques (such as long internal CoT) to solve challenging problems like scientific and mathematical reasoning [OpenAI, 2024]. These tasks often rely on long and complex reasoning paths, and therefore, it seems essential to introduce detailed supervision signals in the reasoning process. Moreover, from a practical perspective, effective supervision on long reasoning paths not only improves reasoning performance, but it also helps the model eliminate redundant or unnecessary reasoning steps, thereby reducing reasoning complexity and improving efficiency.

4.4.5 Inference-time Alignment

In this section we explored a variety of methods to align models with human preferences and annotations. However, one of the significant limitations of many such methods is that LLMs must be fine-tuned. For RLHF and its variants, training LLMs with reward models can be computationally expensive and unstable, leading to increased complexity and costs when applying these approaches. In this case, we can consider aligning models at inference time, thus avoiding the additional complexity and effort involved.

One simple way to achieve inference-time alignment is to use the reward model to select the best one from N alternative outputs generated by the LLM, a method known as **Best-of- N sampling (BoN sampling)**. We can consider BoN sampling as a form of reranking. In fact, reranking methods have been widely used in NLP tasks, such as machine translation, for a long time. They are typically applied in situations where training complex models is costly. In such cases, directly reranking the outputs allows for the incorporation of these complex models at a very low cost⁸.

⁸Reranking methods can also help us explore what are known as model errors and search errors, although these

In the BoN sampling process, the LLM takes the input sequence \mathbf{x} and generates N different output sequences $\{\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_N\}$:

$$\{\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_N\} = \arg\text{TopN}_{\mathbf{y}} [\Pr(\mathbf{y}|\mathbf{x})] \quad (4.85)$$

where the $\arg\text{TopN}$ operation returns the top- N outputs that maximize the function $\Pr(\mathbf{y}|\mathbf{x})$. These outputs can be generated in a variety of ways, depending on the search algorithm used by the model (e.g., sampling or beam search). Once the N -best output candidates are generated, the reward model is used to evaluate and select the best one:

$$\hat{\mathbf{y}}_{\text{best}} = \max\{r(\mathbf{x}, \hat{\mathbf{y}}_1), \dots, r(\mathbf{x}, \hat{\mathbf{y}}_N)\} \quad (4.86)$$

It is worth noting that the result of BoN sampling is also influenced by the diversity of the N -best list. This is a common issue with most reranking methods. Typically, we wish the N -best output candidates to have relatively high quality but be sufficiently different from each other. In many text generation systems, the N -best outputs are very similar, often differing by just one or two words. The diversity issue is even more challenging in LLMs, as the N -best outputs generated by an LLM can be different in their wordings, yet their semantic meanings are often quite similar. In practice, one can adjust the model hyperparameters and/or adopt different LLMs to generate more diverse output candidates for reranking. Nevertheless, as with many practical systems, we need to make a trade-off between selecting high-quality candidates and ensuring sufficient variation in the generated outputs.

BoN sampling can be used for training LLMs as well. A closely related method is **rejection sampling**. In this method, we first select the “best” outputs from the N -best lists via the reward model, and then take these selected outputs to fine-tune the LLM. In this way, we can introduce human preferences into the training of LLMs via a much simpler approach compared to RLHF. Many LLMs have adopted rejection sampling for fine-tuning [Nakano et al., 2021; Touvron et al., 2023b].

4.5 Summary

In this chapter, we have explored a range of techniques for aligning LLMs. In particular, we have discussed fine-tuning methods that enable LLMs to follow instructions and align them with human preferences. One of the benefits of fine-tuning LLMs is computation efficiency. Unlike pre-training based on large-scale neural network optimization, fine-tuning is a post-training step and so is less computationally expensive. Moreover, it is better suited to address problems that are not easily solved in pre-training, such as human value alignment. The widespread attention to the alignment issue has also led to a surge of research papers on this topic, which has posed challenges in writing this chapter, as it is difficult to cover all the latest techniques. However, we have tried to provide a relatively detailed introduction to the fundamental approaches to alignment, such as

issues are not often discussed in the context of LLMs. For example, suppose we have an old model and a new, more powerful model. We can use the new model to select the best output from the N -best list of the old model as the oracle output. The performance difference between the oracle output and the top-1 output of the original N -best list reflects the performance gain brought by the new model. If the performance gain is significant, we can say that the old model has more model errors. If the gain is small, it may indicate that the issue lies in search errors, as the best candidates were not found.

instruction fine-tuning and RLHF.

While we have focused on LLM alignment techniques in this chapter, the term *AI alignment* is a wide-ranging concept. It generally refers to the process of ensuring that the behavior of an AI system aligns with human values, goals, and expectations. The idea of AI alignment can be traced back to the early days of AI. A widely cited description of AI alignment comes from an article by the mathematician and computer scientist Norbert Wiener [Wiener, 1960]. The quote is as follows

If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere ... we had better be quite sure that the purpose put into the machine is the purpose which we really desire.

At that time, AI alignment was a distant concern for researchers. But today, it greatly influences the design of various AI systems. For example, in robotics, alignment is critical to ensuring that autonomous robots safely interact with humans and their environments. In autonomous driving, cars must not only follow traffic laws but also make complex, real-time decisions that prioritize human safety, avoid accidents, and navigate ethical dilemmas.

In current AI research, alignment is usually achieved by developing a surrogate objective that is analogous to the real goal and steering the AI system towards this objective. However, designing the objective of AI alignment is very difficult. One reason is that human values are diverse and often context-dependent, making it difficult to distill them into a single, universally applicable objective function. Also, the complexity of real-world environments, where values and goals often conflict or evolve over time, further complicates alignment efforts. Even if we could define an appropriate objective, AI systems may find unintended ways to achieve it, leading to “misaligned” outcomes that still technically satisfy the objective but in a harmful or counterproductive way.

These challenges have motivated and are motivating AI research towards more aligned systems, either through developing new mechanisms for perceiving the world or more efficient and generalizable methods to adapt these systems to given tasks. More importantly, as AI systems become more powerful and intelligent, especially given that recent advances in LLMs have shown remarkable capabilities in dealing with many challenging problems, the need for AI alignment has become more urgent. Researchers have started to be concerned with AI safety and warn the community that they need to develop and release AI systems with great caution to prevent these systems from being misaligned [Russell, 2019; Bengio et al., 2024].

CHAPTER 5

Inference

Once we have pre-trained and fine-tuned an LLM, we can apply it to make predictions on new data. This process is called inference, in which the LLM computes the probabilities of different possible outputs given an input, and selects the output that maximizes the probability. The inference problem is generally expressed in the following form:

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{x}) \quad (5.1)$$

where \mathbf{x} is the input sequence, \mathbf{y} is a possible output sequence, and $\hat{\mathbf{y}}$ is the best output sequence.

This is perhaps one of the most widely adopted formulas in NLP, and dates back to the early days of speech recognition and machine translation systems based on probabilistic models. Although for some applications, such as predicting a token using a very small language model, solving this optimization problem seems trivial, for most situations the computational challenges arise from both calculating $\Pr(\mathbf{y}|\mathbf{x})$ and performing the $\arg \max$ operation. The problems we therefore wish to address in this chapter involve: 1) computing the prediction probability efficiently given a trained LLM, and 2) devising an efficient (suboptimal) search for $\hat{\mathbf{y}}$.

At a high level, these are fundamental issues in artificial intelligence, which have been extensively studied. So many well-established techniques can be directly applied, for example, one can use greedy search algorithms to implement an efficient inference system. On the other hand, model-specific optimizations, such as efficient attention models for Transformers, can be considered to further improve efficiency. But, in many practical applications, we still need to make a trade-off between accuracy and efficiency, by carefully combining various techniques.

The importance of the inference problem in LLMs also lies in the fact that many application scenarios require processing extremely long sequences. Recent studies have found that injecting additional prompts and contextual information, such as long chain-of-thought prompts, during inference can significantly improve the performance of LLMs. This provides a new approach to scaling LLMs: better results can be achieved by increasing the compute at inference time. For instance, through inference-time scaling, [OpenAI \[2024\]](#)'s o1 and [Deepseek \[2025\]](#)'s R1 systems have demonstrated impressive performance on complex reasoning and programming tasks. This, in turn, has encouraged the NLP field to focus more on the issue of efficient inference.

In this chapter, we will introduce basic concepts and algorithms of LLM inference, including prefilling-decoding frameworks, search (decoding) algorithms, and evaluation metrics of inference performance. We will then present methods for improving the efficiency of LLM inference, covering a range of techniques for speeding up the system and compressing the model. Finally, we will discuss inference-time scaling, which is considered an important application of inference optimization.

5.1 Prefilling and Decoding

In this section, we present the prefilling-decoding framework, which is the most commonly used for interpreting and implementing LLM inference processes. We first introduce the notation and background knowledge, and then describe the details of the framework, such as the decoding algorithms for LLM inference.

5.1.1 Preliminaries

Although we have described LLMs many times in this book, we begin by briefly defining the notation to facilitate the subsequent discussion, and to make this chapter self-contained.

- x:** The input token sequence. It is conceptually equivalent to a “prompt”, which includes instructions, user inputs, and any additional context intended as input to the LLM. **x** comprises $m + 1$ tokens, denoted by $x_0 \dots x_m$, where x_0 is the start symbol $\langle \text{SOS} \rangle$.
- y:** The output token sequence, also called the response to the input. **y** comprises n tokens, denoted by $y_1 \dots y_n$.
- $y_{<i}$:** The output tokens that precede position i , that is, $y_{<i} = y_1 \dots y_{i-1}$.
- $\Pr(\mathbf{y}|\mathbf{x})$:** The probability of generating **y** given **x** using the LLM. If the LLM is parameterized by θ , we can write it as $\Pr_\theta(\mathbf{y}|\mathbf{x})$.
- $[\mathbf{x}, \mathbf{y}]$:** The concatenated token sequence of **x** and **y**. That is, $[\mathbf{x}, \mathbf{y}] = x_0 \dots x_m y_1 \dots y_n$. Occasionally, we use the notation $\text{seq}_{\mathbf{x}, \mathbf{y}}$ to represent $[\mathbf{x}, \mathbf{y}]$.
- $\Pr([\mathbf{x}, \mathbf{y}])$:** The probability of generating the token sequence $[\mathbf{x}, \mathbf{y}]$ using the LLM.

As described in Eq. (5.1), the goal of LLM inference is to maximize $\Pr(\mathbf{y}|\mathbf{x})$. Modeling this conditional probability is common in NLP. At first glance, it seems to be a sequence-to-sequence problem, where we transform a sequence into another using encoding-decoding models. However, we are not discussing sequence-to-sequence problems or encoding-decoding architectures. Instead, as discussed in earlier chapters, this modeling problem can be addressed by using decoder-only models. To do this, we can interpret the log-scale probability $\log \Pr(\mathbf{y}|\mathbf{x})$ as the difference between $\log \Pr([\mathbf{x}, \mathbf{y}])$ and $\log \Pr(\mathbf{x})$

$$\log \Pr(\mathbf{y}|\mathbf{x}) = \log \Pr([\mathbf{x}, \mathbf{y}]) - \log \Pr(\mathbf{x}) \quad (5.2)$$

where $\log \Pr([\mathbf{x}, \mathbf{y}])$ and $\log \Pr(\mathbf{x})$ can be obtained by running the LLM on the sequences $[\mathbf{x}, \mathbf{y}]$ and **x**, respectively. For example, we can calculate the probability of generating **x** using the chain

rule

$$\begin{aligned}
\log \Pr(\mathbf{x}) &= \log \Pr(x_0 \dots x_m) \\
&= \log [\Pr(x_0) \Pr(x_1|x_0) \cdots \Pr(x_m|x_0 \dots x_{m-1})] \\
&= \underbrace{\log \Pr(x_0)}_{=0} + \sum_{j=1}^m \log \Pr(x_j|\mathbf{x}_{<j}) \\
&= \sum_{j=1}^m \log \Pr(x_j|\mathbf{x}_{<j})
\end{aligned} \tag{5.3}$$

In other words, we calculate the token prediction log-probability at each position of \mathbf{x} , and sum all these log-probabilities.

In common implementations of LLMs, however, we do not need to compute the log-probability of the input sequence, but use the LLM to directly compute the log-probability of the output sequence in the following form

$$\log \Pr(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{5.4}$$

where $[\mathbf{x}, \mathbf{y}_{<i}]$ represents the context for predicting y_i . We use $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ to denote $\Pr(y_i|[\mathbf{x}, \mathbf{y}_{<i}])$, following the commonly used notation in the literature.

Now, we have two sub-problems in addressing the inference issue described in Eq. (5.1):

- **Model Computation:** we model $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ and compute it in an efficient manner.
- **Search:** we find the optimal (or sub-optimal) output sequence in terms of $\log \Pr(\mathbf{y}|\mathbf{x})$.

The second sub-problem is a classic issue in NLP. We will show in Section 5.1.3 that there are several well-studied algorithms that can be applied to efficiently search the space of possible output sequences. The first sub-problem requires a language model to produce a distribution over a vocabulary V given a sequence of context tokens. We can do this by training a Transformer decoder, which outputs the distribution

$$\Pr(\cdot|\mathbf{x}, \mathbf{y}_{<i}) = \text{Softmax}(\mathbf{H}\mathbf{W}^o)_{m+i} \tag{5.5}$$

$$\mathbf{H} = \text{Dec}([\mathbf{x}, \mathbf{y}_{<i}]) \tag{5.6}$$

Here $\text{Dec}(\cdot)$ produces a sequence of representations, each corresponding to a position of the input sequence. So, if we input $[\mathbf{x}, \mathbf{y}_{<i}]$ to the LLM, \mathbf{H} is an $i' \times d$ matrix, where d is the dimensionality of each representation, and $i' = m + i$ is the number of context tokens. We can then use a Softmax layer to transform these representations into distributions of tokens. $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$ is the linear mapping matrix of the Softmax layer, and $\mathbf{H}\mathbf{W}^o$ transforms the d -dimensional representations in \mathbf{H} into the $|V|$ -dimensional representations. The use of the subscript $m + i$ indicates that the Softmax function is performed only on the representation at position $m + i$. See Figure 5.1 for an illustration of this architecture.

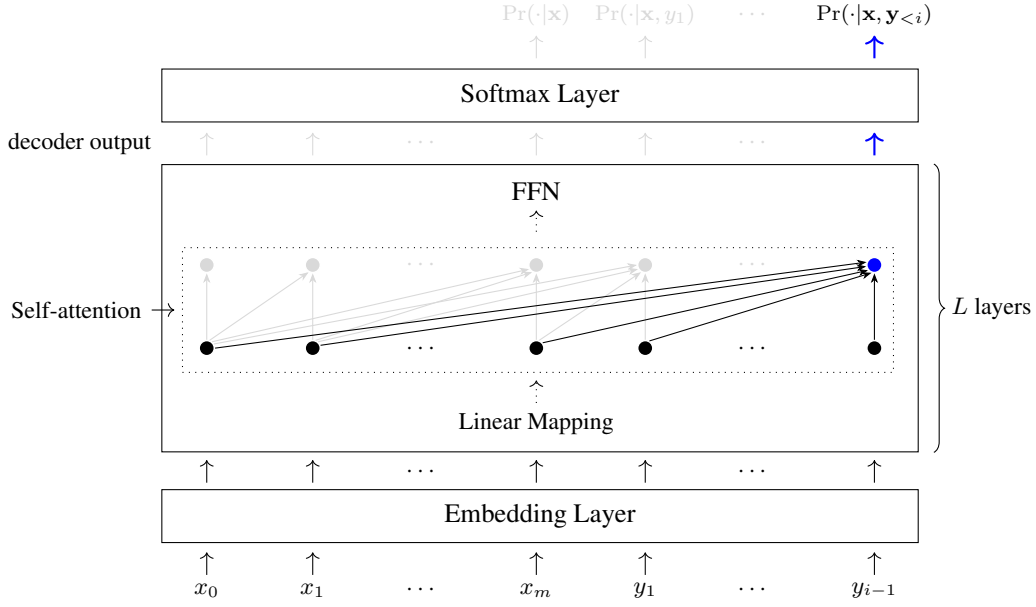


Fig. 5.1: The decoder-only architecture for LLMs. The decoder consists of an embedding layer and a stack of Transformer layers. In each Transformer layer, the input passes through a linear mapping, a self-attention network, and an FFN. The output of the decoder is a sequence of representations that are taken as input to a Softmax layer, which generates a distribution of tokens for each position.

$\text{Dec}(\cdot)$ is a Transformer decoding network that consists of an embedding network and a number of stacked self-attention and FFN networks. We will not discuss Transformers in detail here, as readers can easily learn about these models from the literature. However, it is worth pointing out that the difficulty of inference is in part from the use of the self-attention mechanism in Transformers. Recall that a general form of single-head self-attention is given by

$$\text{Att}_{\text{qkv}}(\mathbf{q}_{i'}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{q}_{i'} \mathbf{K}^T}{\sqrt{d}}\right) \mathbf{V} \quad (5.7)$$

where $\mathbf{q}_{i'} \in \mathbb{R}^d$ is the query at the position i' (i.e., position of y_i), and \mathbf{K} and $\mathbf{V} \in \mathbb{R}^{i' \times d}$ are the keys and values up to i' , respectively.

At each step during inference, we call the self-attention function $\text{Att}_{\text{qkv}}(\cdot)$, followed by an FFN, to generate a d -dimensional representation that integrates information from both the current token and its left context. This process is repeated through L layers of self-attention and FFN, forming a stack of Transformer layers. The output of the L -th layer in this stack is the final representation.

Each time, the model attends position i' to all previous positions, which results in $2i'$ vector products (i' times for $\mathbf{q}_{i'} \mathbf{K}^T$ and i' times for the product of $\text{Softmax}(\frac{\mathbf{q}_{i'} \mathbf{K}^T}{\sqrt{d}})$ and \mathbf{V}). Hence, generating a sequence of length len has a time complexity of $O(L \times len^2)$ for the self-attention network. Clearly, the inference of this model is slow for long sequences due to its quadratic time complexity with respect to sequence length. Therefore, many improvements to Transformers and alternative models have focused on efficient methods that are faster than this quadratic time complexity, such as sparse attention mechanisms and linear-time models. A detailed discussion of efficient Transformers can be found in the previous chapters, and this section will focus on the

standard Transformer architecture.

Note that in self-attention, the queries, keys, and values of a layer are linear mappings from the same input (i.e., the output of the previous layer). Once a new key-value pair is generated, it is repeatedly used in subsequent inference steps. Rather than regenerating these key-value pairs during inference, a more desirable way is to store them in a structure, called the **key-value cache**, or the **KV cache**. Thus, (\mathbf{K}, \mathbf{V}) can straightforwardly be considered a KV cache. This cache is updated as follows

$$\mathbf{K} = \text{Append}(\mathbf{K}, \mathbf{k}_{i'}) \quad (5.8)$$

$$\mathbf{V} = \text{Append}(\mathbf{V}, \mathbf{v}_{i'}) \quad (5.9)$$

where $(\mathbf{k}_{i'}, \mathbf{v}_{i'})$ is the newly generated key-value pair at position i' , and $\text{Append}(\mathbf{a}, \mathbf{b})$ denotes a function that appends a row vector \mathbf{b} to a matrix \mathbf{a} . Figure 5.2 shows how a Transformer decoder works with a KV cache.

Finally, the process of computing $\log \Pr(\mathbf{y}|\mathbf{x})$ is summarized as follows:

1. We concatenate \mathbf{x} and \mathbf{y} into a sequence $[\mathbf{x}, \mathbf{y}]$. For each position i' of this sequence, we perform the following steps.
 - (a) We compute the embedding of the token at position i' , and feed the resulting embedding as an initial representation into the stack of Transformer layers.
 - (b) In each Transformer layer, we pass the input representation through the self-attention network first and then through an FFN. In the self-attention network, the input representation is transformed into $\mathbf{q}_{i'}$, $\mathbf{k}_{i'}$, and $\mathbf{v}_{i'}$. Then, we update the KV cache (\mathbf{K}, \mathbf{V}) using $\mathbf{k}_{i'}$ and $\mathbf{v}_{i'}$ (see Eqs. (5.8-5.9)). Then, we compute the output of the attention model by attending $\mathbf{q}_{i'}$ to (\mathbf{K}, \mathbf{V}) (see Eq. (5.7)).
 - (c) If $i' > m$ (i.e., $i = i' - m \geq 0$), we take the output of the Transformer stack and compute the token prediction probability $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ via the Softmax layer (see Eq. (5.5)).
2. When reaching the end of the sequence, we obtain $\log \Pr(\mathbf{y}|\mathbf{x})$ by summing $\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ over $i \in [1, n]$ (see Eq. (5.4)).

5.1.2 A Two-phase Framework

As we have seen, language modeling is a standard autoregressive process, where each token is generated one at a time, conditioned on the previous tokens. For Transformers, this requires the model to maintain a KV cache that stores past representations, and attend the newly generated representation to this KV cache. If we think of the model $\Pr(\mathbf{y}|\mathbf{x})$ from the perspective of computing the KV cache, it is natural to divide inference into two phases:

- **Prefilling.** The prefilling phase computes the KV cache for the input sequence \mathbf{x} . It is called prefilling because the model prepares and stores the key-value pairs for each token

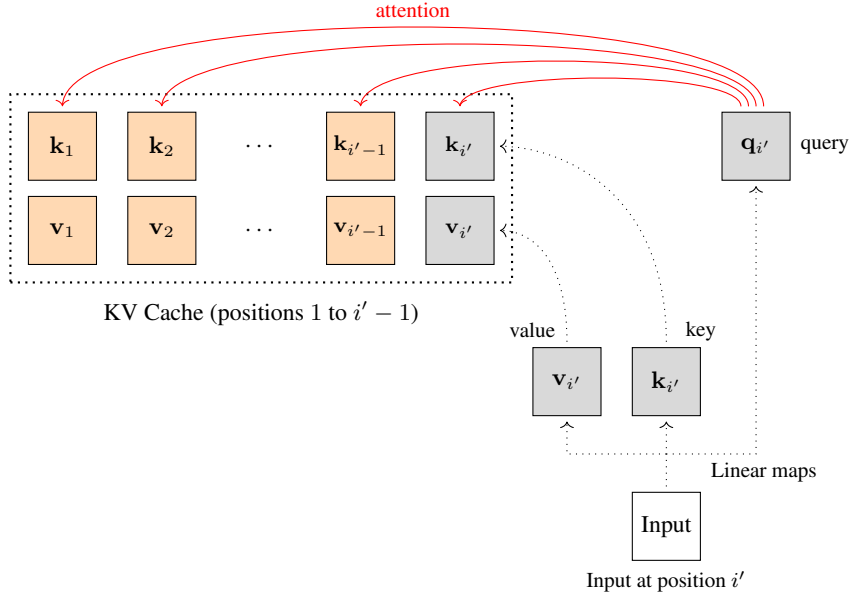
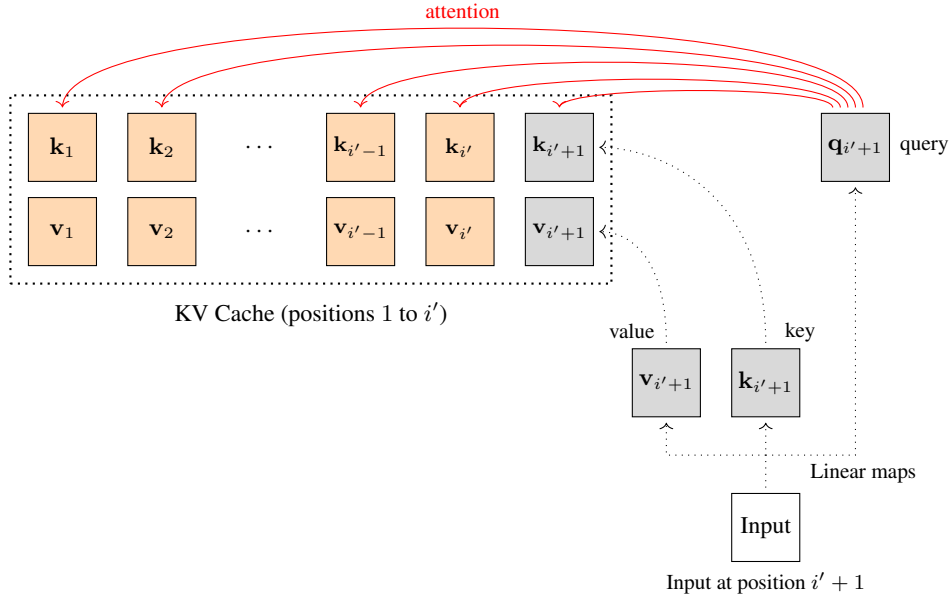
(a) Updating the KV Cache at Position i' (b) Updating the KV Cache at Position $i' + 1$

Fig. 5.2: Illustration of the KV cache. We update the KV cache at a position, perform the attention operation, and then move to the next position to repeat the process.

in the input before the actual inference begins. The process of prefilling in an LLM can be expressed as

$$\text{cache} = \text{Dec}_{\text{kv}}(\mathbf{x}) \quad (5.10)$$

where $\text{Dec}_{\text{kv}}(\cdot)$ is the decoding network (i.e., the same as $\text{Dec}(\cdot)$), but it returns the KV

cache in self-attention instead of the output representations. cache is a list, given by

$$\text{cache} = \{\text{cache}^1, \dots, \text{cache}^L\} \quad (5.11)$$

where cache^l represents the key-value pairs for the l -th layer.

- **Decoding.** The decoding phase continues generating tokens based on the KV cache, as illustrated in Figure 5.2. When a new token is input into the decoder, we update the KV cache in each layer by adding the new key-value pair. The updated cache is then used for self-attention computation. The token generation stops when some stopping criterion is met, such as when the generated token is the end symbol. The goal of decoding is to find the best predicted sequence, which is given by

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y}} \Pr(\mathbf{y}|\text{cache}) \quad (5.12)$$

Here we use $\Pr(\mathbf{y}|\text{cache})$ instead of $\Pr(\mathbf{y}|\mathbf{x})$ to emphasize that the decoding process actually relies on the KV cache rather than \mathbf{x} .

The prefilling and decoding processes are illustrated in Figure 5.3. Note that both these processes are autoregressive. However, as shown in Table 5.1, they differ in several aspects, which lead to very different implementations in practice.

In essence, while the underlying model of prefilling is based on token prediction, it can be considered an encoding process. This is because our goal is not to generate tokens, but to build a context representation (i.e., the KV cache) for the subsequent steps in the decoding phase. In this sense, it is similar to BERT, where we encode the input sequence into a sequence of contextualized token representations. On the other hand, unlike BERT which generates bidirectional sequence representations, prefilling is based on standard language modeling tasks, and is thus unidirectional. Note that, since the entire sequence \mathbf{x} is input into the model all at once, all queries can be packed together and the self-attention operation is performed on \mathbf{x} in parallel. Let \mathbf{Q} be the queries that are packed into one matrix. The self-attention model in prefilling can be defined as

$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}} + \text{Mask}\right)\mathbf{V} \quad (5.13)$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{d \times (m+1)}$. $\text{Mask} \in \mathbb{R}^{(m+1) \times (m+1)}$ is a mask that ensures that each token only attends to itself and the tokens that precede it in the sequence. It is represented by setting the values in the mask corresponding to future tokens to a large negative number, for example, for the query \mathbf{q}_i and the key \mathbf{k}_j , we set the value of the entry (i, j) to $-\infty$ if $i < j$. One advantage of processing the sequence with a single self-attention computation is that we can make better use of the parallel computing capabilities of modern GPUs, and so speed up prefilling. In general, the prefilling process is considered compute-bound. This is because merging multiple computational operations into one operation reduces the number of data transfers and the performance bottleneck usually comes from the computational capacity rather than memory bandwidth.

Decoding is a standard left-to-right text generation process. The token sequence is generated autoregressively by predicting one token at a time based on the KV cache. Each time a new token is generated, we need to attend it to previous tokens, following Eq. (5.7). Therefore, the decoding

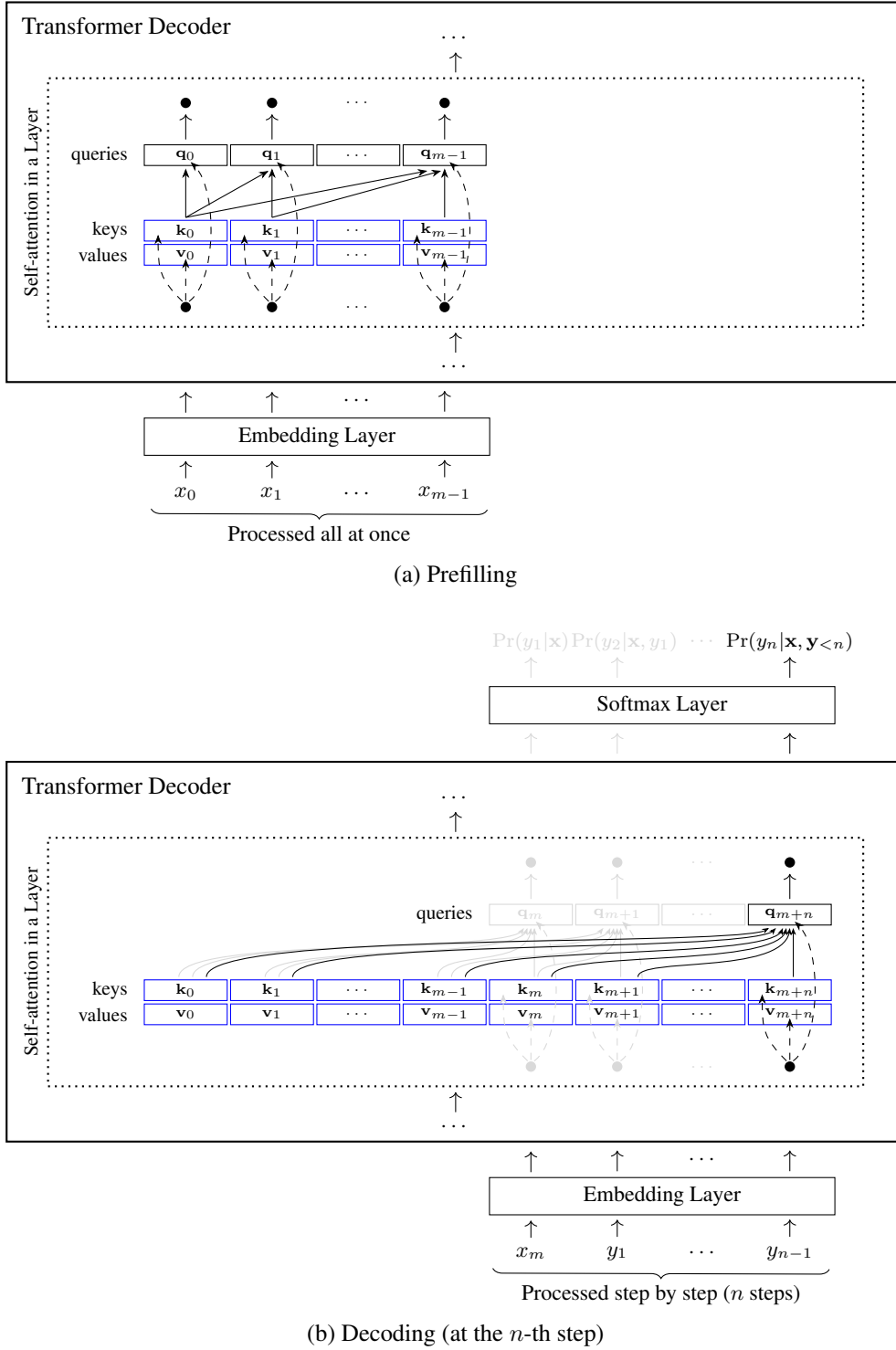


Fig. 5.3: Illustration of the prefilling and decoding processes. In prefilling, the entire input sequence is processed together and the KV cache is filled. In decoding, the LLM generates the output sequence step by step based on the prefilled KV cache.

process is memory-bound due to its frequent access to the KV cache. The cost of decoding grows significantly as more tokens are generated. In most cases, decoding is computationally more expensive than prefilling. Note that this is not just because, in decoding, the LLM generates tokens one by one and repeatedly updates the KV cache. As we will see in the following subsection,

	Prefilling	Decoding
Goal	Set up initial context \mathbf{x} .	Continue generating tokens \mathbf{y} after the initial input.
All-at-once Visibility	Tokens in \mathbf{x} are presented all at once.	Tokens in \mathbf{y} are presented sequentially, that is, predicting a token requires waiting for the previous tokens to be predicted first.
Context Use	Build the context or encoded representation of the input.	Use the cached key-value pairs (from prefilling) to generate further tokens.
Resource Limitation	Compute-bound	Memory-bound
Computational Cost	High	Very High

Table 5.1: Prefilling vs Decoding.

we may need to explore multiple different token sequences during decoding, which makes the problem more complex and increases its cost further.

5.1.3 Decoding Algorithms

So far our discussion of LLM inference has primarily focused on the model computation problem, that is, how to compute $\Pr(\mathbf{y}|\mathbf{x})$. Now we turn to the discussion of the search problem. The problem can be stated as: given an LLM $\Pr(\mathbf{y}|\mathbf{x})$, how do we efficiently search for the best output sequence $\hat{\mathbf{y}}$ given the input sequence \mathbf{x} (or the generated KV cache)? Naively, we can consider all of the output sequences, compute the prediction probability for each, and then select the output sequence having the highest probability. This method can guarantee the globally optimal solution, but direct exhaustive search is impractical for LLMs as the number of possible output sequences grows exponentially with the length of \mathbf{y} .

In practice, various heuristic search algorithms, such as greedy search and sampling-based search, are commonly employed to approximate the solution. Each of these methods offers trade-offs between search quality and computational efficiency. The search problem, therefore, becomes a balancing act between exploration and exploitation, where the goal is to find an efficient strategy that produces high-quality outputs without exploring the entire space.

Before giving a more detailed discussion of these methods, let us first informally define what a search space is and how it is represented. In LLM inference, we define a hypothesis as a tuple of input and output sequences. Since \mathbf{x} is fixed during inference, we can simply consider each hypothesis as an output sequence. The search space, denoted by \mathcal{Y} , is then the set of all possible hypotheses (i.e., output sequences) that the model can generate. The search problem for LLM inference can be re-expressed as

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \mathcal{Y}} \Pr(\mathbf{y}|\mathbf{x}) \quad (5.14)$$

In NLP, \mathcal{Y} is commonly represented in a tree data structure to facilitate search. Figure 5.4 shows an example of the search tree resulting from a small vocabulary. In this example, a node

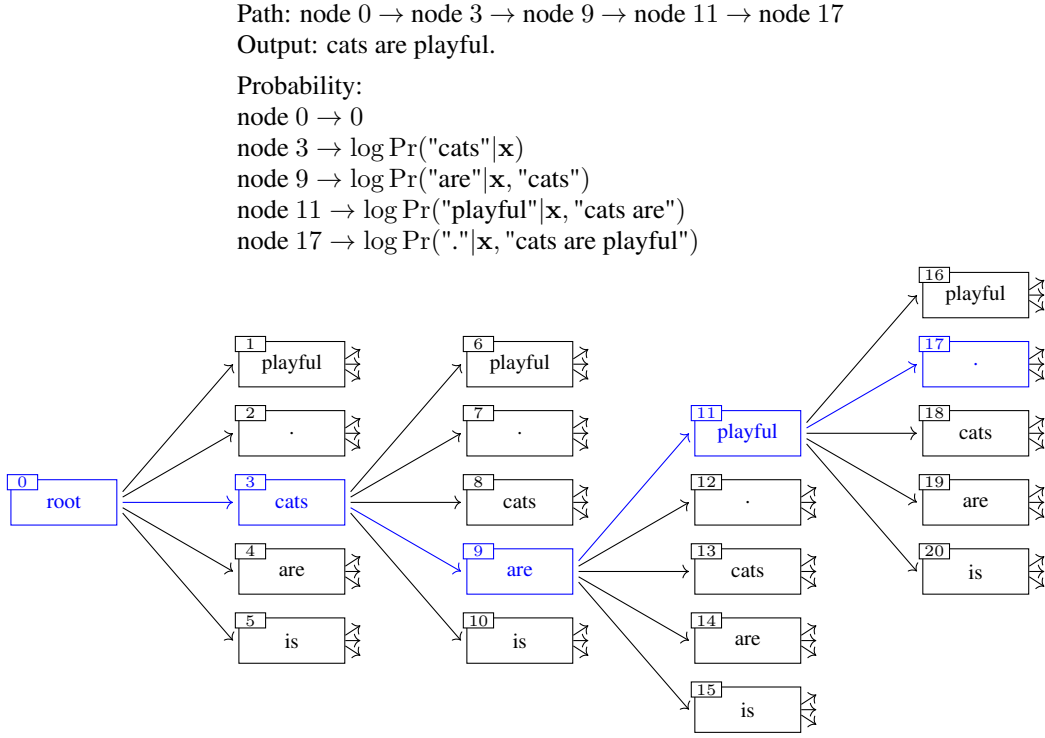


Fig. 5.4: A search tree for decoding. At each node, we expand the tree by considering all possible tokens, each leading to a new node representing a potential continuation of the text. Here we highlight a path through nodes 0, 3, 9, 11, and 17. The path represents the output sequence “cats are playful.”, whose log-probability can be computed by accumulating the log-probabilities of these nodes.

represents a prefix subsequence that can be shared by many sequences. The search starts with the root of the tree, which can be regarded as the beginning of all sequences that can be generated¹. Each child node extends the prefix of its parent node by adding one token from the vocabulary to the sequence, along with the probability of predicting the token given the prefix. This process continues as each node further branches out into additional child nodes, each representing a new possible extension of the sequence with another token. The search tree thus grows deeper and wider, representing an ever-increasing number of potential sequences as more tokens are appended. This structure allows us to efficiently traverse through possible sequences, evaluating each in terms of the log-probability accumulated over the path from the root to that node. For example, in Figure 5.4, the path from the root to the node 17 corresponds to the output sequence “Cats are playful.”. The prediction log-probability $\log \Pr(\mathbf{y}|\mathbf{x})$ is the sum of the log-probabilities of all the nodes on this path.

In general, the search tree is organized as levels, where each level consists of all nodes that are the same distance from the root node. Thus, a breadth-first search over the tree essentially performs left-to-right generation of tokens. Nodes in the same level correspond to sequences of the same length. As the search progresses, new tokens are appended to these sequences, expanding them incrementally.

Let Y_i be the set of the sequences that the LLM generates at step i . Y_i can be obtained by expanding each sequence in Y_{i-1} with all possible next tokens in the vocabulary V , given in the

¹Here, since the predictions in LLMs are based on \mathbf{x} , we can think of the root as a representation of \mathbf{x} .

following recursive form

$$Y_i = Y_{i-1} \times V \quad (5.15)$$

where $Y_{i-1} \times V$ denotes the Cartesian product of Y_{i-1} and V (i.e., each sequence in Y_{i-1} is concatenated with each token in V). Note that if a sequence in Y_{i-1} is complete (e.g., ending with the $\langle \text{EOS} \rangle$ token), it will not be expanded any further. Let $\Psi(Y_i)$ be the set of all complete sequences in Y_i . Then, the search space can be expressed as

$$\mathcal{Y} = \Psi(Y_1) \cup \Psi(Y_2) \cup \dots \cup \Psi(Y_{n_{\max}}) \quad (5.16)$$

where n_{\max} is the maximum length of a sequence.

Most decoding algorithms follow this level-by-level search process. However, \mathcal{Y} consists of an exponentially large number of sequences, and a direct search in such a vast space is computationally infeasible. Therefore, practical decoding algorithms often rely on strategies to prune the search space and avoid exploring low-quality sequences. For example, at each decoding step, Y_i can be obtained in the following way

$$Y_i = \text{Prune}(Y_{i-1} \times V) \quad (5.17)$$

where $\text{Prune}(\cdot)$ is a function that selectively removes sequences less likely to result in high-quality outcomes. In general, we expect that $|Y_i| \ll |Y_{i-1}| \cdot |V|$. Thus we can drastically reduce the number of sequences under consideration at each step, ensuring that the computational load does not grow exponentially with the sequence length.

Next, we will introduce these decoding algorithms. Some of them have already been discussed in sequence-to-sequence models, while others are more commonly used in LLMs.

5.1.3.1 Greedy Decoding

Greedy search (or greedy decoding) is one of the most widely used decoding methods in NLP, particularly in text generation tasks like machine translation. The idea behind greedy search is straightforward: at each step in generation, it selects the next token that has the highest prediction probability. For each sequence $\mathbf{y} = y_1 \dots y_i \in Y_{i-1} \times V$, we can evaluate it using $\log \Pr(\mathbf{y}|\mathbf{x})$. This log-probability can be easily computed by noting that

$$\begin{aligned} \log \Pr(\mathbf{y}|\mathbf{x}) &= \log \Pr(y_1 \dots y_i|\mathbf{x}) \\ &= \underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{accumulated up to the parent node}} + \underbrace{\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}_{\text{newly computed for the current node}} \end{aligned} \quad (5.18)$$

Here the first term is the sum of the log-probabilities of the path from the root to the parent node, which has been computed in the previous decoding steps. At step i , we only need to compute the second term which is the standard token prediction log-probability produced by the LLM.

The “best” token at step i is then chosen as

$$\begin{aligned}
 y_i^{\text{top1}} &= \arg \max_{y_i \in V} \log \Pr(y_1 \dots y_i | \mathbf{x}) \\
 &= \arg \max_{y_i \in V} \left[\underbrace{\log \Pr(\mathbf{y}_{<i} | \mathbf{x})}_{\text{fixed wrt. } y_i} + \log \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \right] \\
 &= \arg \max_{y_i \in V} \log \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i})
 \end{aligned} \tag{5.19}$$

Thus, the “best” sequence generated up to step i is given by

$$\mathbf{y}^{\text{top1}} = y_1 \dots y_{i-1} y_i^{\text{top1}} \tag{5.20}$$

Finally, Y_i contains only this sequence

$$Y_i = \{\mathbf{y}^{\text{top1}}\} \tag{5.21}$$

The greedy choice in one decoding step is illustrated in Figure 5.5 (a). Greedy search offers computational efficiency and simplicity in implementation for LLM inference. Its primary disadvantage, however, lies in its suboptimal nature — high-quality sequences are likely pruned at early stages of decoding. Therefore, greedy search is appealing for tasks that demand speed and simplicity. For tasks that require better search results, alternative strategies such as beam search, which explores multiple potential paths simultaneously, are preferable.

5.1.3.2 Beam Decoding

Beam search (or beam decoding) is a natural extension of greedy search. Instead of selecting the single most probable token at each step, beam search maintains a fixed number of the best candidates at each step, known as the “beam width”. See Figure 5.5 (b) for an illustration of beam search.

Let K be the beam width. Given a parent node, which corresponds to the prefix $y_1 \dots y_{i-1}$, we can select the top- K next tokens by

$$\{y_i^{\text{top1}}, \dots, y_i^{\text{topK}}\} = \arg \text{TopK}_{y_i \in V} \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \tag{5.22}$$

where $\arg \text{TopK}$ is a function that ranks the prediction probabilities of all possible next tokens and selects the top K candidates. Given these tokens, the top- K sequences for step i are given by

$$\mathbf{y}^{\text{top1}} = y_1 \dots y_{i-1} y_i^{\text{top1}} \tag{5.23}$$

$$\vdots$$

$$\mathbf{y}^{\text{topK}} = y_1 \dots y_{i-1} y_i^{\text{topK}} \tag{5.24}$$

Then, we can define Y_i as

$$Y_i = \{\mathbf{y}^{\text{top1}}, \dots, \mathbf{y}^{\text{topK}}\} \tag{5.25}$$

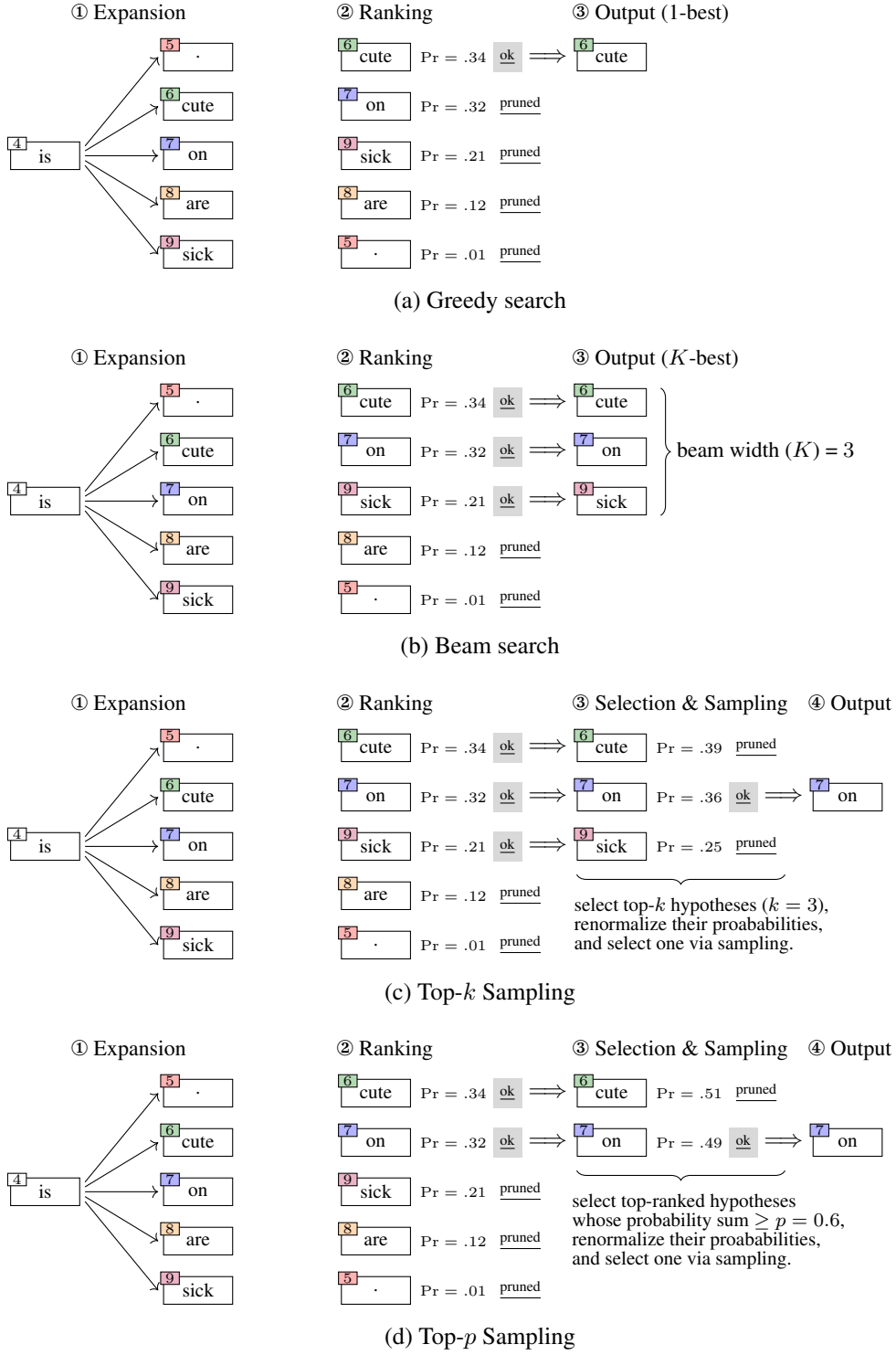


Fig. 5.5: Illustrations of greedy decoding, beam decoding, top- k decoding and top- p decoding methods (in one decoding step).

We can adjust the beam width K to balance search efficiency and accuracy. But a very large beam width might not be helpful. In many practical applications, selecting a relatively small number for K , such as $K = 2$ or $K = 4$, is often sufficient to achieve satisfactory performance in LLM inference.

5.1.3.3 Sampling-based Decoding

Both greedy and beam search generate deterministic outputs, that is, given an LLM, the output of the model will always be the same every time it processes the same input. The deterministic nature of greedy and beam search ensures predictability and reliability in applications where consistent outcomes are critical, such as in formal document generation, where varying outputs could cause confusion or errors. On the other hand, one disadvantage of these methods is the lack of diversity and flexibility. For example, in creative tasks like story generation or conversational agents, generic or repetitive outputs generated by deterministic systems are often less engaging.

To add variation into LLM outputs, we can use sampling-based decoding methods. There are two commonly used methods.

- **Top- k Sampling.** This method selects the next token from the top- k most likely candidates at each step of the generation process [Fan et al., 2018]. Let \bar{V}_i be the selection pool for top- k sampling. We can define it as

$$\bar{V}_i = \{y_i^{\text{top}1}, \dots, y_i^{\text{top}k}\} \quad (5.26)$$

where $\{y_i^{\text{top}1}, \dots, y_i^{\text{top}k}\}$ are the top- k tokens selected based on their prediction probabilities (see Eq. (5.22)). Once the selection pool is determined, we recompute the prediction probability distribution over \bar{V}_i . One of the simplest ways to do this is to renormalize their probabilities:

$$\bar{\text{Pr}}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\text{Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i})}{\sum_{y_j \in \bar{V}_i} \text{Pr}(y_j|\mathbf{x}, \mathbf{y}_{<i})} \quad (5.27)$$

Alternatively, we can calculate the distribution by using the Softmax function:

$$\bar{\text{Pr}}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i})}{\sum_{y_j \in \bar{V}_i} \exp(u_{y_j})} \quad (5.28)$$

where u_{y_i} is the logit for token y_i . Then, we sample a token \bar{y}_i from this distribution:

$$\bar{y}_i \sim \bar{\text{Pr}}(y_i|\mathbf{x}, \mathbf{y}_{<i}) \quad (5.29)$$

The corresponding sequence is $\bar{\mathbf{y}} = y_1 \dots y_{i-1} \bar{y}_i$, and Y_i is given by

$$Y_i = \{\bar{\mathbf{y}}\} \quad (5.30)$$

- **Top- p Sampling.** This sampling method, also known as **nucleus sampling**, follows a procedure similar to that of top- k sampling. Instead of drawing from a fixed size candidate pool, it selects the next token from the smallest set of tokens that together have a cumulative probability higher than a predefined threshold p [Holtzman et al., 2020]. In this way we prevent the prediction from choosing from low-probability tokens in the long tail that could lead to incoherent or nonsensical outputs. To obtain the candidate pool in the top- p sampling method, we can sort all tokens by their predicted probabilities. Then, starting with the token with the highest probability, we continue to add tokens to the candidate pool until the

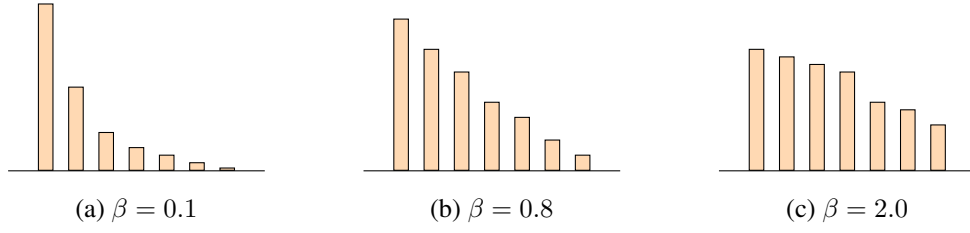


Fig. 5.6: Histogram estimates of the distributions generated by the Softmax function with different values of the temperature parameter β .

cumulative probability of the tokens in the pool reaches or exceeds p (we denote the size of the candidate pool at this point as k_p). The candidate pool can then be expressed as

$$\bar{V}_i = \{y_i^{\text{top}1}, \dots, y_i^{\text{top}k_p}\} \quad (5.31)$$

The subsequent steps, such as the renormalization of the distribution and sampling, are the same as in the top- k sampling method (see Eqs.(5.27-5.30)).

See Figure 5.5 (c-d) for illustrations of the top- k and top- p sampling methods. By limiting the choices to a smaller set of high-probability tokens, these methods strike a balance between randomness and coherence. They allow for more diverse outputs while still maintaining a reasonable level of relevance and fluency. However, the value of k or p must be carefully chosen: if k or p is too small, the output may still be overly deterministic (more like greedy decoding), and if k or p is too large, the LLM might produce degenerate outputs.

In order to further control the randomness of the token selection process, the renormalized distribution $\bar{\text{Pr}}(\cdot)$ is typically obtained by using the Softmax function with the temperature parameter, given by

$$\bar{\text{Pr}}(y_i | \mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i} / \beta)}{\sum_{y_j \in \bar{V}_i} \exp(u_{y_j} / \beta)} \quad (5.32)$$

Here β is a temperature parameter β that controls the sharpness of the probability distribution derived from logits. In Figure 5.6, we show simple examples involving distributions generated by the above function with different temperatures. When the temperature is set to a higher value, the resulting probability distribution becomes more uniform, as the differences between the logits are diminished. This means that each token in the candidate pool has a more equal chance of being selected, leading to greater diversity in the generated output. By contrast, when the temperature is set to a lower value, the distribution becomes sharper, making the high-probability tokens even more likely to be chosen, which often results in more deterministic outputs. For example, if we set p to 1 and β to a very small number (approaching zero), the top- p sampling method will become equivalent to the greedy search method.

5.1.3.4 Decoding with Penalty Terms

One common improvement to decoding methods in text generation is to modify the search objective. For example, one can replace maximum a posteriori (MAP) decoding with minimum Bayes risk (MBR) decoding [Kumar and Byrne, 2004], where the focus shifts from selecting the single

most probable output to choosing an output that minimizes the expected risk over a distribution of possible outputs. Here we explore methods that incorporate penalty terms into decoding. These methods offer a simple but effective way to make decoding more controllable.

Recall from Eq. (5.14) that the goal of decoding is to maximize the likelihood of the output sequence. With penalty terms, the objective is extended to include additional factors that penalize or reward certain behaviors in the generated text. A general form of the new objective is given by

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \mathcal{Y}} [\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})] \quad (5.33)$$

where $\text{Penalty}(\mathbf{x}, \mathbf{y})$ is a function that quantifies the degree to which the generated sequence \mathbf{y} violates certain constraints or exhibits undesirable behaviors given the input \mathbf{x} . The design of $\text{Penalty}(\cdot)$ is very flexible, thus allowing us to incorporate a wide range of constraints or prior knowledge into it. Below, we present some common types of penalty functions.

- **Repetition Penalty.** A repetition penalty discourages the model from generating repetitive or redundant text. The penalty function might measure the frequency of repeated tokens or phrases in the generated sequence and impose a penalty proportional to their occurrence.
- **Length Penalty.** A length penalty ensures that the generated sequence adheres to a desired length. For example, in text summarization tasks, the penalty function could penalize outputs that are too short or too long.
- **Diversity Penalty.** A diversity penalty promotes variation in the generated text. For example, in beam search, we can measure the similarity between generated hypotheses, and encourage the model to explore different hypotheses.
- **Constraint-based Penalty.** A constraint-based penalty enforces specific constraints related to the content or style of the generated text. For example, in machine translation, the penalty function could penalize outputs that deviate from a desired tone or terminology.

In general, we can consider $\text{Penalty}(\mathbf{x}, \mathbf{y})$ as a function that defines the cost of generating the surface form of the output sequence \mathbf{y} given the input sequence \mathbf{x} . Alternatively, this function can be defined to assess the hidden states of an LLM when generating \mathbf{y} . For example, [Su et al. \[2022\]](#) develop a penalty term that calculates the maximum distance between the representation of the predicted token and the representations of the previously generated tokens. Therefore, the search objective will penalize degenerated outputs, such as texts with many repetitions.

The method described in Eq. (5.33) is general and can be easily adapted to different search algorithms. For example, in greedy search, we can keep the single sequence that maximizes $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ at each decoding step; in sampling-based search, we can rank and select the top-ranked sequences based on $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ to form the candidate pool.

5.1.3.5 Speculative Decoding

Speculative decoding stems from the concept of **speculative execution**, where a system makes educated guesses about future actions and performs them in advance. If the guess is correct, the

results are immediately available, which speeds up processing. In the case of LLM inference, suppose we have two models. One is a smaller, faster model (called draft model), and the other is the full, more accurate model (called verification model). These two models represent two baselines in LLM inference: the draft model is efficient but not very accurate; the verification model is usually the one we want to run, but it is very slow. Given a prefix, we first use the draft model to speculatively predict a sequence of likely future tokens. This is a standard autoregressive decoding process, but it is still fast in practice due to the high efficiency of the draft model. Then, the verification model evaluates the speculated tokens in parallel. It checks whether the predicted tokens are correct or need to be adjusted. Note that, since we can deal with these tokens all at once, the verification can be done in a single step for all the tokens simultaneously, rather than in a token-by-token manner. If the speculated tokens are correct, they are accepted, and the process continues with the next set of tokens. If they are incorrect, the incorrect speculations are discarded, and the verification model is used to generate the correct tokens.

To be more specific, let us see the speculative decoding method presented in [Leviathan et al. \[2023\]](#)'s work. In this method, the draft model is a small language model, denoted by $\Pr_q(y_i|\mathbf{x}, \mathbf{y}_{<i})$, while the verification model is a normal LLM, denoted by $\Pr_p(y_i|\mathbf{x}, \mathbf{y}_{<i})$. The goal is that, given a prefix, we use the draft model to autoregressively predict up to τ tokens. The verification model is then employed to generate the last token at the point where errors begin to occur in the speculative predictions. Figure 5.7 illustrates one step in this decoding process.

The speculative decoding algorithm can be summarized as follows.

- Given the prefix $[\mathbf{x}, \mathbf{y}_{\leq i}]$, we use the draft model to predict the next τ consecutive tokens, denoted by $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+\tau}\}$. This is a token-by-token generation process, given by

$$\hat{y}_{i+t} = \arg \max_{y_{i+t}} \Pr_q(y_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1}) \quad (5.34)$$

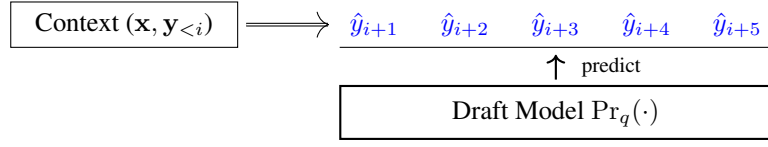
- We evaluate $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+\tau}\}$ using the verification model, that is, we compute $\{\Pr_p(\hat{y}_{i+1}|\mathbf{x}, \mathbf{y}_{\leq i}), \dots, \Pr_p(\hat{y}_{i+\tau}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+\tau-1})\}$. Note that we can compute these probabilities in parallel, and so this verification step is efficient.
- We determine the maximum number of accepted speculated tokens. In order to keep the notation uncluttered, we denote $\Pr_q(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1})$ and $\Pr_p(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+t-1})$ simply by $q(\hat{y}_{i+t})$ and $p(\hat{y}_{i+t})$, respectively. We then define that, if $q(\hat{y}_{i+t}) \leq p(\hat{y}_{i+t})$, then we accept this speculation. By contrast, if $q(\hat{y}_{i+t}) > p(\hat{y}_{i+t})$, we reject this speculation with probability $1 - \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}$. Starting from \hat{y}_{i+1} , the maximum number of accepted consecutive speculated tokens is defined as

$$n_a = \min \left\{ t - 1 \mid 1 \leq t \leq \tau, r_t > \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})} \right\} \quad (5.35)$$

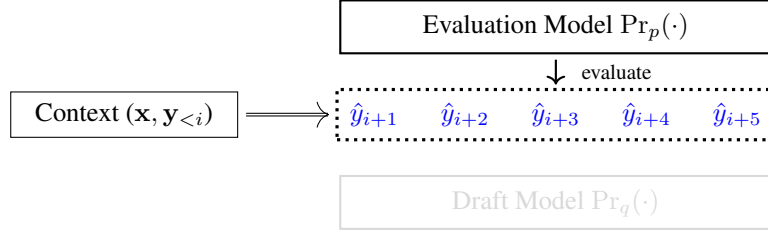
where r_t is a variable drawn from the uniform distribution $U(0, 1)$.

- Given n_a , we keep the speculated tokens $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}\}$. We then use the verification model to make a new prediction at $i + n_a + 1$

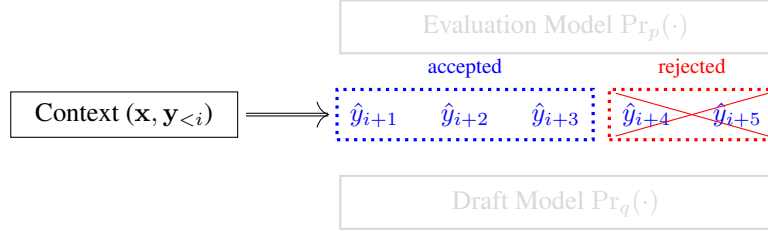
$$\bar{y}_{i+n_a+1} = \arg \max_{y_{i+n_a+1}} \Pr_p(y_{i+n_a+1}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} \dots \hat{y}_{i+n_a}) \quad (5.36)$$



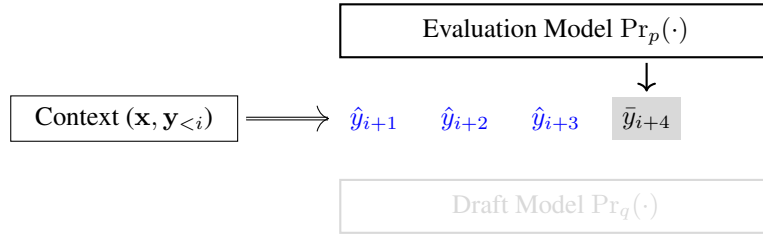
(a) Predict the next τ tokens given the context using the draft model ($\tau = 5$)



(b) Evaluate the predicted tokens using the evaluation model



(c) Determine the number of accepted tokens



(d) Predict a new token following the accepted tokens using the evaluation model

Fig. 5.7: Illustration of one step of speculative decoding. The goal is to predict as many next tokens as possible using the draft model. There are four sub-steps. Given the context, we first use the draft model to predict the next τ tokens. Then, we evaluate these predictions in parallel using the evaluation model. Next, we determine the maximum number of predicted tokens that can be accepted. Finally, we use the evaluation model to predict a new token following these accepted tokens.

- Above, we have described one step of speculative decoding. The result sequence (including both the context and predicted tokens) is illustrated as follows

$$\begin{array}{ccc}
\underbrace{[\mathbf{x}, \mathbf{y}_{<i}]}_{\text{Context}} & \underbrace{\hat{y}_{i+1} \dots \hat{y}_{i+n_a}}_{\substack{n_a \text{ tokens} \\ \text{predicted using} \\ \text{the draft model}}} & \underbrace{\bar{y}_{i+n_a+1}}_{\substack{\text{One token} \\ \text{predicted using} \\ \text{the verification model}}}
\end{array}$$

Once we have finished this step, we add the predicted tokens $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}, \bar{y}_{i+n_a+1}\}$ to the context, and repeat the above process.

In practice, we usually wish to use a smaller draft model so that predicting $\{\hat{y}_{i+1}, \dots, \hat{y}_{i+n_a}\}$ would be computationally cheaper. But a very small draft model is less accurate and can result in smaller n_a . We therefore need to carefully select the draft model to make the trade-off between the computational efficiency and accuracy.

5.1.3.6 Stopping Criteria

Stopping criteria are a critical component of LLM inference. They typically involve rules or conditions that specify when the model should stop generating text during decoding. Most LLMs are trained to generate an end-of-sequence token (e.g., $\langle \text{EOS} \rangle$ or $\langle /s \rangle$) to signal the end of the generated text. So one of the simplest strategies is that the generation process stops when this token is produced. For beam search, which explores multiple hypotheses simultaneously, the process can continue until a given number of complete sequences have been generated.

In practical applications, it will generally be undesirable to generate very long sequences, and so we need to reduce the decoding cost and unnecessary verbosity. One commonly-used stopping criterion is the maximum length of the output. The model stops generating text once it has produced a predetermined number of tokens. Alternatively, we can stop the decoding based on the real cost, such as the computational resources or time constraints. For example, in real-time applications like chatbots, decoding may need to stop after a certain time limit to ensure responsiveness.

Another approach is to design stopping criteria based on the behavior of LLMs. For example, decoding can be stopped if the probability of predicting the next token falls below a certain threshold. In addition to probability-based stopping, a repetition detection module can be implemented to trigger the model to stop if it begins repeating tokens or phrases beyond a predefined limit. This helps prevent redundant or incoherent outputs.

5.1.4 Evaluation Metrics for LLM Inference

Evaluating the performance of LLMs during inference involves a variety of metrics to assess how well these models meet desired standards, such as accuracy, robustness, usability, and efficiency. As with most NLP systems, we can evaluate LLMs using accuracy-based metrics, such as perplexity and F1 score. We can also examine their robustness by testing how well they handle ambiguous or challenging inputs, including adversarial, perturbed, or out-of-distribution data. Additionally, usability can be assessed by measuring how well the generated outputs align with user expectations in terms of fluency, coherence, relevance, and diversity. Human evaluators can rate the naturalness of the text or assess whether the responses are contextually appropriate and logically consistent. Ethical and fairness metrics can also be included to ensure LLMs avoid perpetuating biases or generating harmful content.

All of the evaluation metrics mentioned above essentially focus on assessing the quality of the outputs. Given the high cost of deploying and applying LLMs, efficiency metrics are also very important for practitioners. Below are some commonly used efficiency metrics [Nvidia, 2025]:

- **Request Latency.** This metric measures the total time taken from when a request is sent to the LLM until the complete response is received. This includes the time taken for data transmission, processing by the model, and the return of the output to the user.
- **Throughput.** It refers to the number of tokens or requests the model can process per second.
- **Time to First Token (TTFT).** This metric measures the time it takes from the beginning of a request being sent to the generation of the first token of the response. If data transmission does not consume too much time, then TTFT is mainly the time for prefilling and predicting the first token.
- **Inter-token Latency (ITL).** This metric refers to the time taken to generate each subsequent token after the first one. It reflects the efficiency of the decoding process.
- **Tokens Per Second (TPS).** This metric quantifies the number of tokens that the model can generate per second.
- **Resource Utilization.** This involves measuring the computational resource usage (e.g., CPU and GPU utilization) and memory consumption of the model during inference.

In addition to these metrics, energy efficiency and cost efficiency are practical considerations for deploying LLMs at scale. Energy efficiency measures the amount of electrical power consumed by the model during inference. Cost efficiency, on the other hand, evaluates the total expenses related to deploying and maintaining the model.

In general, choosing the right evaluation metrics depends on the specific task and application. While quality-focused metrics are essential for assessing LLMs, efficiency metrics are equally crucial for their effective deployment in real-world applications. A comprehensive evaluation framework should include both sets of metrics to accurately estimate an LLM's performance and practicality.

5.2 Efficient Inference Techniques

In practical applications, we often wish a system to be as efficient as possible. For LLM inference, this typically involves two types of improvements: reducing memory requirements and accelerating the system. For example, we can modify the Transformer architecture to avoid memory explosion when processing very long input sequences. Another example is that we can compress input sequences to reduce computational overhead while preserving their semantic information. In addition, techniques like quantization and pruning can be employed to further optimize memory usage and inference speed.

Efficient inference is a wide-ranging topic that overlaps with several sub-fields of LLMs, such as architecture design and model compression. Most of these topics have been covered in previous

chapters. For example, in Chapter 2, we discussed efficient Transformer architectures and long-context LLMs; and in Chapter 3, we discussed prompt compression methods for reducing prompt length. In this section, we focus on techniques that are commonly used in LLM deployment and serving.

5.2.1 More Caching

In real-world applications, it is common practice to store frequent requests and their corresponding responses in a cache. When a new request hits the cache, the system can retrieve the response directly from the cache instead of recomputing the result. One straightforward implementation is a key-value datastore (e.g., a hash table) that maps input sequences to their LLM-generated output sequences. In the simplest case, we can collect frequent queries, generate their responses using the LLM, and store these query-response pairs in the datastore. This creates a basic sequence-level caching mechanism that allows the system to bypass LLM computation when the input sequence exactly matches a cached query.

A straightforward extension of the caching mechanism is to cache prefixes and their corresponding hidden states. Given an input sequence \mathbf{x} in a dataset \mathcal{D} , we can process it as in the standard prefilling phase. Thus, we obtain a sequence of prefixes and their corresponding KV cache states:

$$\begin{aligned} x_0 (\mathbf{x}_{<1}) &\Rightarrow \text{cache}_{<1} \\ x_0 x_1 (\mathbf{x}_{<2}) &\Rightarrow \text{cache}_{<2} \\ &\dots \\ x_0 x_1 \dots x_{m-1} (\mathbf{x}_{<m}) &\Rightarrow \text{cache}_{<m} \end{aligned}$$

where $\text{cache}_{<i}$ denotes the KV cache for the prefix $\mathbf{x}_{<i}$ (see also Eq. (5.10)). All these mappings can be stored in the prefix cache for efficient reuse.

When processing a new sequence that shares a common prefix with a previously seen sequence in \mathcal{D} , we can load the corresponding cached hidden states instead of recomputing them. Specifically, if a new input \mathbf{x}' has $\mathbf{x}_{<k}$ (i.e., $\mathbf{x}'_{<k} = \mathbf{x}_{<k}$ for some $k \leq m$), we can initialize the KV cache with $\text{cache}_{<k}$ and only compute the hidden states for the remaining tokens $\mathbf{x}'_{\geq k}$.

As usual, we can maintain a key-value datastore that maps frequently encountered prefixes to their precomputed KV caches. The lookup can be performed using a hash of the prefix tokens, allowing constant-time access to the cached states. Care must be taken to manage memory usage, as storing all possible prefixes may be infeasible for large datasets. Practical systems often employ least recently used (LRU) caching methods or other strategies to balance between computational savings and memory constraints.

5.2.2 Batching

Batching in LLM inference refers to the process of processing multiple input sequences simultaneously as a group (called a batch) rather than one at a time. Because modern GPUs excel at parallel processing, batching allows them to compute multiple sequences in a single forward pass, keeping the hardware fully occupied. Therefore, when serving LLMs at scale, batching is important for

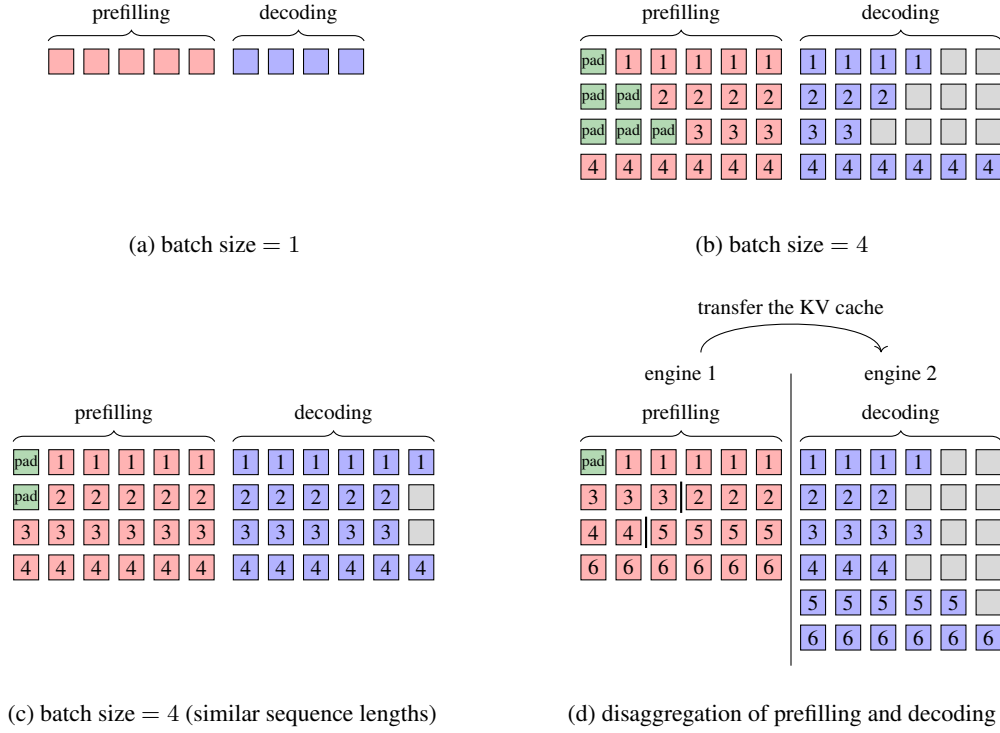


Fig. 5.8: Illustrations of basic batching methods. We use a 2D layout to illustrate the batch, where each square represents a token. Red squares indicate tokens in the prefilling stage, blue squares represent tokens in the decoding stage, green squares denote padding tokens, and gray squares correspond to meaningless tokens. Subfigures (a) and (b) compare the cases where the batch size is 1 and 4, respectively. Subfigure (c) shows the strategy of grouping sequences with similar lengths into the same batch. Subfigure (d) illustrates the disaggregation of prefilling and decoding. In this approach, we can make better use of the parallelism of GPUs by concatenating multiple short sequences into a single long sequence for joint processing. This allows us to maximize the number of tokens processed in a batch while minimizing the number of padding tokens. However, as a trade-off, we need to copy the KV cache to the decoding engine and reorganize it after the prefilling phase, which introduces additional data transfer overhead.

improving computational efficiency and maximizing hardware utilization².

To illustrate the idea of batching, Figure 5.8 (a-b) show simple examples with batch sizes of 1 and 4, respectively. When using a batch size of 1 (i.e., without batching), the GPU processes one input sequence at a time. Thus, the processing is sequential: the next sequence must wait for the current computation to finish. By contrast, when using a batch size of 4, the GPU can process four sequences simultaneously in a single forward pass. As the input sequences vary in length, we need to standardize their length using padding techniques. Here we use left padding, which adds dummy tokens to the beginnings of short sequences, so all the sequences in the batch would have the same length for prefilling. For decoding, tokens are generated simultaneously for all these sequences, and the generation process continues until the longest sequence reaches completion.

The above examples imply a trade-off between throughput and latency, which is a very important consideration in designing and implementing LLM inference systems. If we choose a smaller batch size, the latency would be lower, as fewer tokens need to be processed in a single run of inference. Imagine that we have only one sequence. The result becomes available immediately after generation completes, with no additional computational overhead. However, this low-latency advantage comes at the cost of underutilizing parallel computing resources, as the parallelism of

²See <https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf> for a simple evaluation.

GPUs remains largely idle during sequential processing. On the other hand, if we use a larger batch, we can make better use of the parallelism, as GPUs can be occupied by large-scale matrix computations. As a result, we can process more tokens in the same period of time and the throughput is improved. However, since the result is obtained only when the last token in the batch is predicted, the latency would be higher.

In practice, we usually prefer to use a slightly larger batch, but try to fill the batch with sequences of similar lengths to reduce the number of padding tokens and improve device utilization. For example, we can group the incoming user requests in a short period of time into buckets, each of which contains sequences with similar lengths. Then, we can fill the batch with sequences in the same bucket, so that we can minimize wasted computational resources, as illustrated in Figure 5.8 (c).

Another approach to implementing batching in LLMs is to disaggregate the prefilling and decoding processes [Wu et al., 2023a; Patel et al., 2024; Zhong et al., 2024]. For example, we can perform prefilling on one GPU, and perform decoding on another GPU. One advantage of disaggregation is that we can rearrange the input sequences in the batch to better fill it, because there is no interference between prefilling and decoding. For example, we can concatenate multiple short sequences into a longer one, thus ensuring that the lengths of sequences in the batch are as consistent as possible, as illustrated in Figure 5.8 (d). In this way, we can maximize the throughput of the prefilling phase. However, as a trade-off, we need to transfer the KV cache to the devices performing decoding, which also incurs extra communication overhead. Typically, this method requires a high-bandwidth, low-latency network to achieve optimal performance.

In this section, we will discuss several improvements to the above basic batching strategies. Most of them are based on an aggregated architecture, that is, decoding and prefilling can be considered as different stages of a model executed on the same device.

5.2.2.1 Scheduling

A practical LLM inference system typically consists of two components:

- **Scheduler.** Its primary role is to efficiently queue and dispatch tasks (i.e., input sequences) to the inference engine based on the current system load and task priorities. This often involves a variety of batching strategies that group certain requests together to maximize processing efficiency in some way.
- **Inference Engine.** It is responsible for the actual execution of the LLMs, processing the queued requests as they come in. As discussed previously, this engine involves both prefilling and decoding processes.

This architecture is illustrated in Figure 5.9. Incorporating scheduling into batch processing provides a flexible way to optimize both the system's throughput and latency, thereby achieving a better balance between them. For example, the batching methods shown in Figure 5.8 (a) and (b) can be considered one of the simplest scheduling strategies, called **request-level scheduling**. In this strategy, once a batch is filled and sent to the engine, the processing of the entire batch cannot

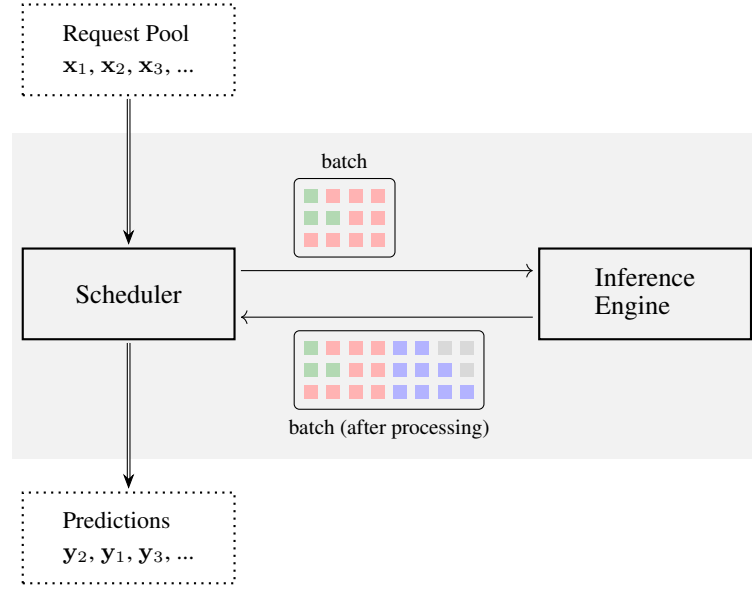


Fig. 5.9: Illustration of the LLM inference architecture involving a scheduler and an inference engine. Each time, the scheduler selects a number of user requests to form a batch and sends it to the inference engine. The scheduler can interact with the inference engine and adjust the batch at certain points during inference, such as at the beginning of batch processing and at the start of each token prediction.

be interrupted. The scheduler waits for this batch to be processed before handling the next batch [Timonin et al., 2022].

A more sophisticated scheduling strategy, called **iteration-based scheduling**, interacts with the inference engine at each token prediction step rather than at the sequence level. This approach allows dynamic batch adjustment during inference, as illustrated in Figure 5.10. Such fine-grained control lets the system prioritize critical tokens or sequences in real-time. For instance, if an urgent request arrives at some decoding step, the scheduler can add this request into the batch so that it can be processed as early as possible. In the following subsections, we will discuss batching methods based on iteration-based scheduling.

5.2.2.2 Continuous Batching

Continuous batching is an iteration-based scheduling method used in the Orca system [Yu et al., 2022]. In this method, an iteration refers to either the entire prefilling procedure or a single decoding step. For example, given an input sequence $\mathbf{x} = x_0 \dots x_m$ and an output sequence $\mathbf{y} = y_1 \dots y_n$, there are $n + 1$ iterations in total: one for prefilling, and n for generating the output tokens (one per token). During scheduling, the batch can be adjusted between iterations. For example, we can either add a new input sequence to the batch, or remove a complete sequence from the batch at some iteration, even if the batch processing is not yet finished.

The general process of continuous batching includes the following steps:

- Initially, a batch is created with one or more input sequences, based on both the inference engine's processing capacity and the current user requests. The batch is then fed into the inference engine.

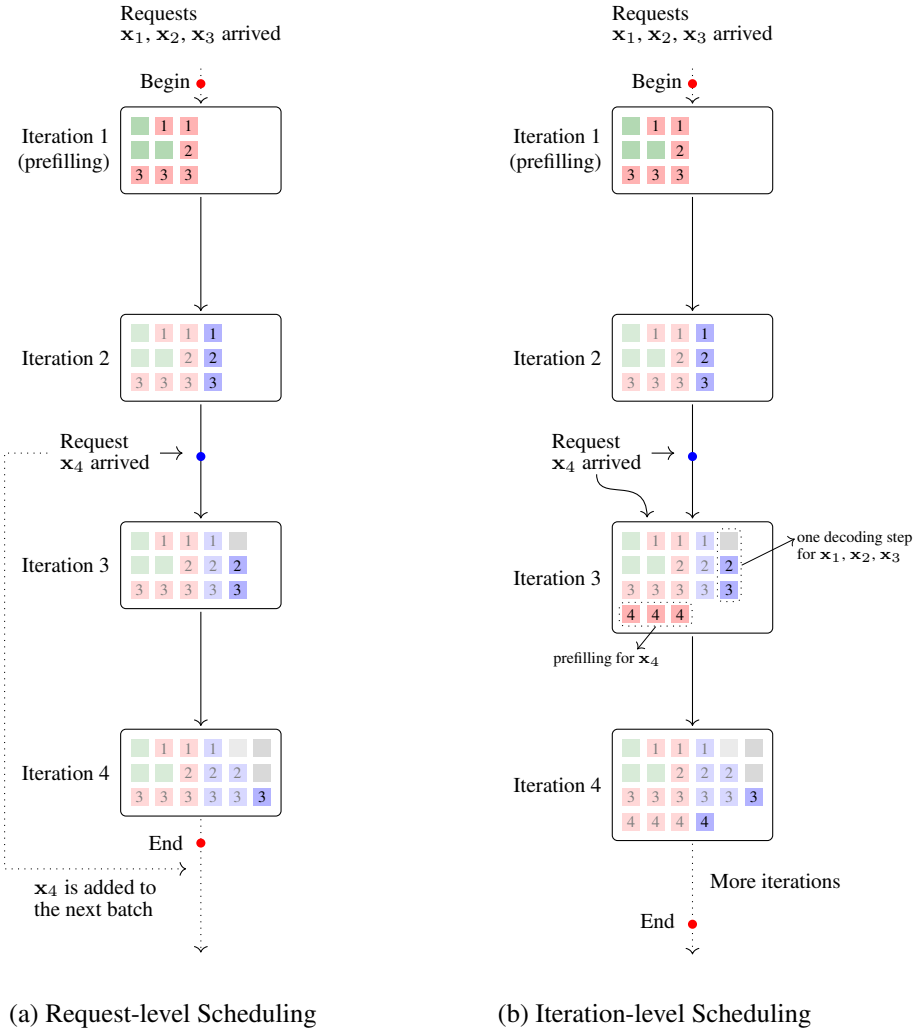


Fig. 5.10: Illustrations of request-level scheduling and iteration-based scheduling. In request-level scheduling, once a batch is created and sent to the inference engine, we cannot adjust the batch. In other words, scheduling only occurs after the processing of a batch finishes. In iteration-level scheduling, we can perform scheduling during batch processing. For example, if a new request arrives at some point during inference, we can add it to the batch and continue processing.

- The inference engine processes the batch iteration by iteration. After each iteration, the scheduler may adjust the batch in one of the following ways:
 - If a sequence in the batch completes generation (i.e., generates the end-of-sequence symbol), that sequence is removed from the batch.
 - If a new user request arrives and the inference engine has additional processing capacity, it is added to the batch.
 - If no sequences are added to or removed from the batch, the batch remains unchanged.
- The processing terminates only when all sequences have been completed and no new user requests arrive.

See Figure 5.11 for an example of continuous batching. In this example, we start with two user

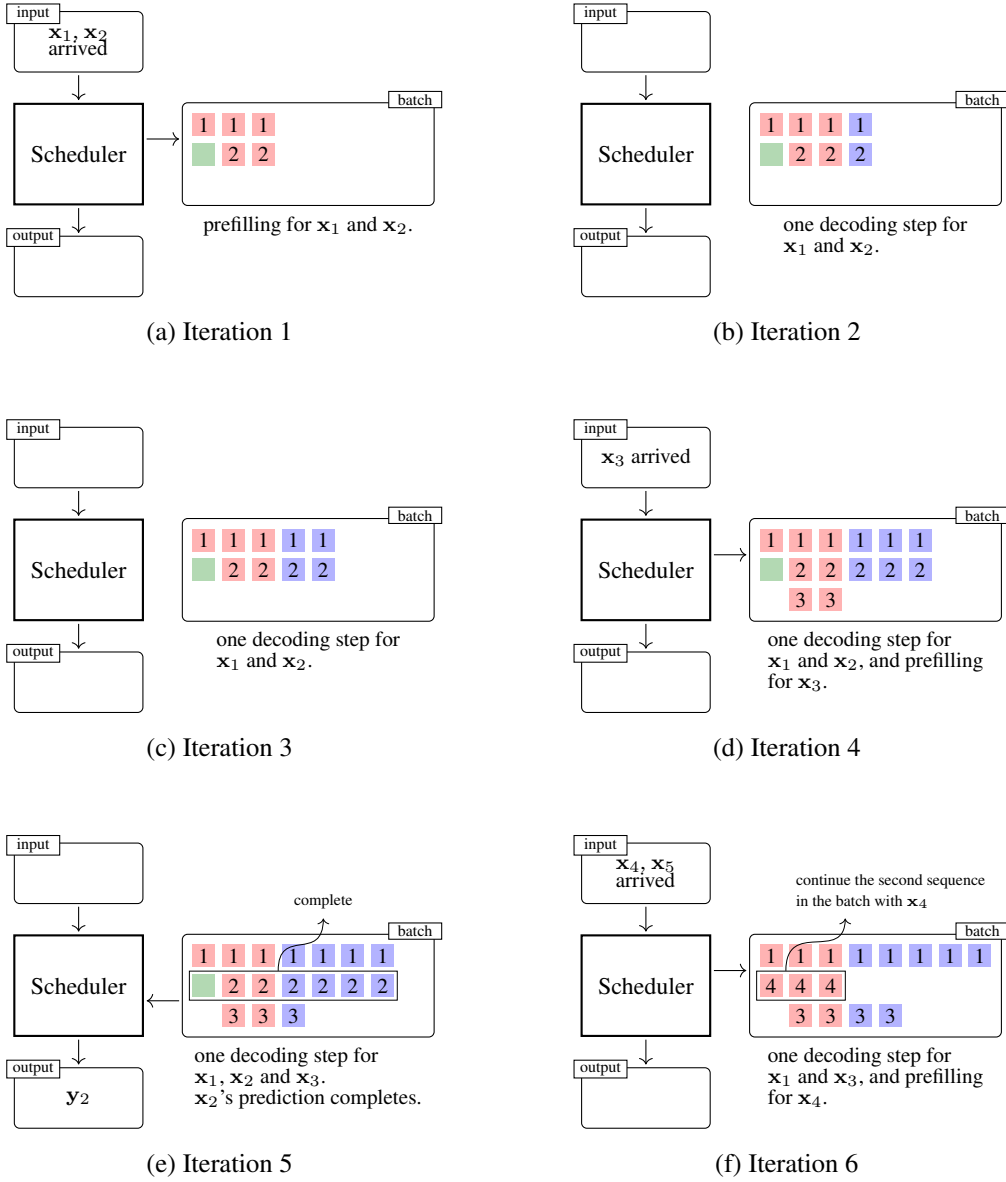


Fig. 5.11: Illustration of batch adjustment in continuous batching. Instead of fixing a batch of input sequences and processing them to completion (as in request-level batching), continuous batching dynamically updates the batch during inference. The system continuously accepts and adds new requests (e.g., x_3 and x_4) into the current batch as long as there is available compute capacity.

requests, x_1 and x_2 . These two sequences are packed into a batch and sent to the inference engine for processing. After the engine completes two iterations, a new user request, x_3 , arrives. At this point, the scheduler adjusts the batch by adding x_3 to it. The inference engine then continues processing the updated batch. Note that the inference engine now processes different sequences in different ways: x_1 and x_2 proceed with the decoding process (i.e., predicting the next tokens), while x_3 undergoes the prefilling process. After some time, the generation for x_2 completes. As it happens, two more user requests, x_4 and x_5 , arrive. The scheduler removes the completed sequence x_2 from the batch and, considering the current load of the inference engine, adds x_4 to the batch. However, x_5 must wait until another sequence in the batch finishes before it can be added.

The idea behind continuous batching is to keep the inference engine fully utilized by processing as many sequences as possible, thereby maximizing computational resource usage. A key difference between continuous batching and standard batching (see Figure 5.8) lies in the fact that, in continuous batching, prefilling and decoding can occur simultaneously across different sequences, whereas in standard batching, these two phases are performed sequentially for the entire batch. As discussed in Section 5.1.2, prefilling is considered a compute-bound process, while decoding is considered a memory-bound process. The intuition behind overlapping prefilling and decoding is to reduce idle times for both computation and data transfer. Consider two mini-batches: one for prefilling and one for decoding. While the prefilling mini-batch keeps the GPUs occupied, the decoding mini-batch can perform memory transfers concurrently.

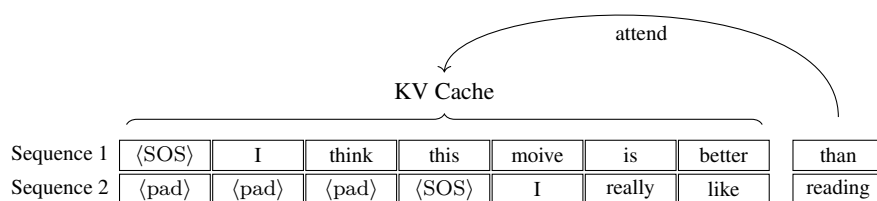
Another difference between continuous batching and standard batching is that continuous batching is prefilling-prioritized, while standard batching is decoding-prioritized [Agrawal et al., 2024]. In continuous batching, once the inference engine has spare computational resources, the scheduler will add new requests to the batch. In other words, these newly added requests will be processed for prefilling as early as possible. This approach improves system throughput, but at the cost of increased latency, as the newly added requests extend the processing time of earlier ones. In contrast, in standard batching, once the batch is created, we must wait for the last sequence in the batch to complete before processing new requests. This ensures relatively low latency, but results in lower device utilization and system throughput.

It is important to note that the cost of continuous batching is that we need to continuously reorganize the batches, which involves rearranging the data in memory. Each time a new request is added, the scheduler needs to reassess and optimize the current batch structure. This dynamic adjustment can incur additional memory and computational overhead, especially when the batches are frequently adjusted. Therefore, while this method can improve throughput, it may also lead to increased memory fragmentation and, in some cases, introduce additional latency.

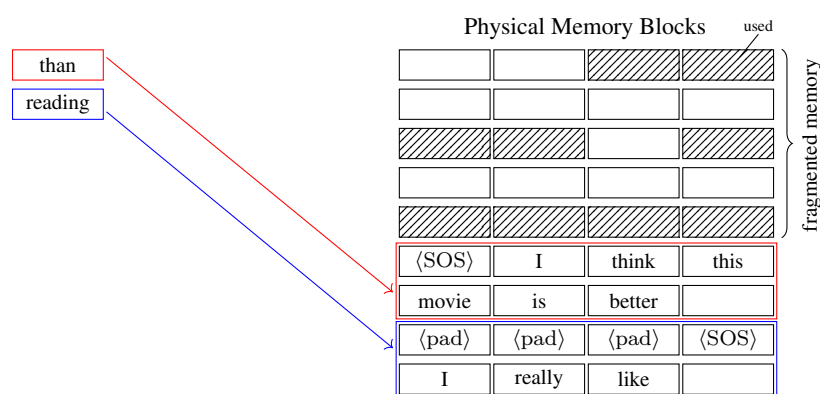
5.2.2.3 PagedAttention

PagedAttention (or paged KV caching) is a technique used in the vLLM system [Kwon et al., 2023]. Inspired by operating system paging, it optimizes memory usage during LLM inference — particularly for the KV cache — by addressing fragmented memory allocation in dynamic batching scenarios with variable-length sequences. The idea behind PagedAttention is to break down large memory requirements for KV caching into more manageable "pages" or chunks of memory. In this way, we do not need to store the KV cache of the full sequence in a continuous memory. Instead, the KV cache is divided into fixed-size blocks (analogous to memory pages in an operating system), which can be non-contiguously allocated in physical memory. One advantage of PagedAttention is that it enables flexible memory management, supporting dynamic sequence growth without requiring expensive reallocation or copying of large contiguous memory regions. Note that PagedAttention is not specifically designed for batching. But it indeed helps improve memory efficiency in batched inference scenarios, where memory management is more demanding and complicated.

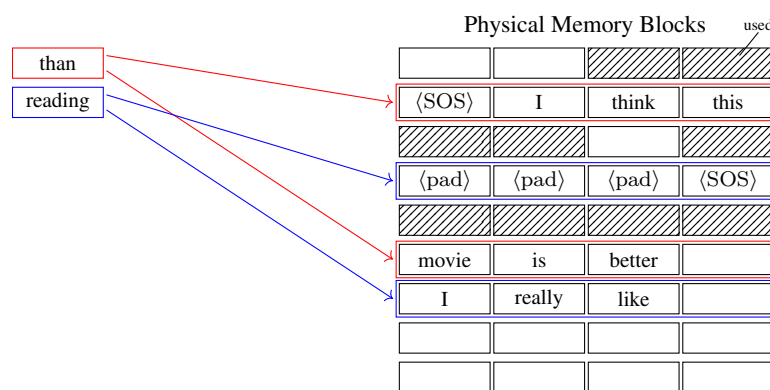
Consider a simple example of memory allocation in Figure 5.12 in which self-attention is performed for a batch consisting of two sequences. For each sequence, we need to attend the current token to the key-value pairs in the KV cache of this sequence, as required by self-attention. In the



(a) Two sequences in a batch



(b) Memory allocation for KV caching in standard self-attention



(c) Memory allocation for KV caching in PagedAttention

Fig. 5.12: Illustration of memory allocation in PagedAttention. There are two sequences in the batch, as illustrated in sub-figure (a). Since the memory is fragmented, the KV cache is stored in a large unused block of memory in standard self-attention (see sub-figure (b)), but the fragmented memory is not used. By contrast, in PagedAttention (see sub-figure (c)), the KV cache is divided into smaller blocks and thus fits into fragmented memory.

standard implementation of self-attention, the KV cache is stored in a contiguous block of memory, allowing us to efficiently access this continuous memory. However, in a paged KV caching system, the KV cache is divided into smaller, fixed-size memory blocks which are not necessarily contiguous. These smaller KV cache blocks can be more effectively allocated to fragmented memory regions, thereby improving memory utilization. Another benefit of distributing chunks of

the KV cache across different memory blocks is that it enables parallelization of the caching process. For example, if the input sequence is long and the memory bandwidth is sufficient, it would be beneficial to write and read the key and value vectors of different segments of the sequence in parallel across multiple memory blocks.

In general, storing contiguous data in non-contiguous regions can cause issues, for example, accessing fragmented data requires additional seek time, which reduces I/O efficiency. However, when handling large-scale data (e.g., performing multiplication on extremely large matrices), we typically do not process all the data at once but instead divide it into smaller blocks for block-level computation. From this perspective, it is also reasonable to partition the attention computation. If the paging strategy is well designed, the additional overhead in memory access can be minimal, while the improvement in memory utilization can be significant.

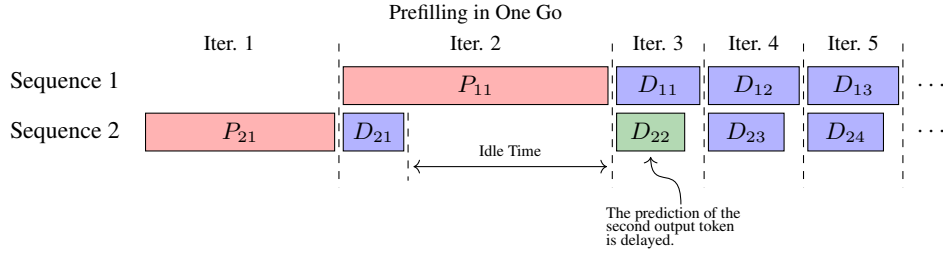
5.2.2.4 Chunked Prefilling

We have seen that, in iteration-level scheduling, prefilling and decoding for different sequences can occur simultaneously. This can be seen as a prefilling-prioritized strategy which can maximize the throughput. However, one such iteration can take a long time if the input sequence is very long and the prefilling process dominates the computation. In this case, decoding for other sequences has to wait until the prefilling completes, leading to increased latency for generating output tokens. Therefore, while prefilling-prioritized strategies are effective for maximizing hardware utilization, they may introduce significant variability in token generation latency, particularly when the system is handling a mix of long and short input sequences.

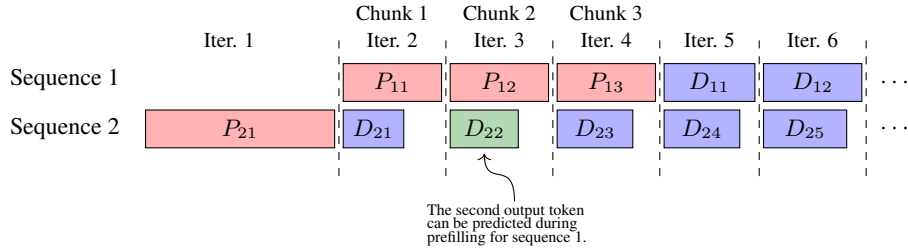
A simple way to reduce decoding latency is to make computations for different sequences in the batch comparable. One such method is to divide sequences into chunks and perform prefilling chunk by chunk. This approach, often referred to as chunked prefilling, processes smaller portions of each sequence at a time, allowing the system to better balance the computational load across sequences [Agrawal et al., 2023]. By choosing an appropriate chunk size, we can ensure that when prefilling and decoding overlap for two sequences, their processing within the same iteration tends to take a similar amount of time. As a result, decoding idle time is reduced and overall throughput is improved.

Figure 5.13 shows an illustration of chunked prefilling in a few iterations. In this example, the batch contains two sequences. The whole prefilling process of the first sequence is divided into three prefilling steps, giving rise to the chunks denoted P_{11} , P_{12} and P_{13} . Each chunk corresponds to one iteration and can thus overlap with one decoding step. In this way, during the prefilling of the first sequence, we can perform three decoding steps, rather than only a single decoding step, as is the case in standard iteration-level scheduling. As a result, the idle time of the decoding process is reduced, and the output tokens can be generated earlier.

Chunked Prefilling improves decoding efficiency by overlapping prefilling and decoding, but at the cost of additional memory overhead and scheduling complexity. In standard prefilling, we process the whole input sequence once, building the KV cache in one go. By contrast, in chunked prefilling, each chunk needs a separate forward pass to compute its attention outputs and update the KV cache. As a result, we need to maintain the KV cache of early chunks while processing later chunks. This also compromises the parallelism of completing the prefilling for the entire sequence in a single pass. In practice, it is usually possible to balance throughput and latency by choosing an appropriate chunk size.



(a) Simple Iteration-level Scheduling



(b) Chunked Prefilling

Fig. 5.13: Comparison of simple iteration-based scheduling and chunked prefilling. P_{xy} denotes the y -th prefilling step for sequence x , and D_{xy} denotes the y -th decoding step for sequence x . In simple iteration-based scheduling (or prefilling-prioritized scheduling), since prefilling is treated as a single iteration, D_{22} has to wait for the completion of the prefilling of sequence 1. In chunked prefilling, the prefilling process can be divided into multiple steps. Thus, D_{22} can execute during prefilling for sequence 1 (i.e., during P_{12}).

It is worth noting that the methods discussed in this subsection can broadly be categorized as priority-based scheduling methods. In these methods, we can give priority to certain requests, or to certain prefilling or decoding steps, so that system resources are allocated in a way that better aligns with specific performance goals. As presented above, for example, we may prioritize decoding over prefilling to minimize token generation latency, or prioritize prefilling over decoding to maximize overall throughput in batch-processing scenarios. Practitioners can design custom priority policies for specific needs and operational constraints in real-world applications, such as request deadlines and importance levels defined by users.

5.2.3 Parallelization

Parallelization is a widely used approach to scale up LLM inference, especially for large-scale deployments. In Chapter 1, we have discussed several common parallelization strategies to parallelize LLM pre-training, such as model parallelism, tensor parallelism, and pipeline parallelism. We have also discussed efficient architectures that are easy to deploy in distributed computing systems. For example, in MoE models, we assigns different experts to different devices³. Only the active experts for a given input are executed, which significantly improves computational efficiency while maintaining model quality. Many of these methods can be directly applied to LLM inference with minimal modifications.

³In LLMs, the experts are typically modular FFNs. So each expert is a part of the FFN component in the Transformer architecture.

However, applying these parallelization techniques to inference poses new challenges compared to pre-training. These issues become especially pronounced in real-time or low-latency inference scenarios, where load imbalance across devices and communication overhead can significantly impact performance. For example, unlike pre-training, where batches can be prepared in advance, inference must handle variable-length sequences in real time. This makes it harder to maintain optimal device utilization and complicates scheduling across heterogeneous computational resources. A related issue is load balancing. When a large number of requests arrive in a short period of time, the system must efficiently distribute workloads across available devices. For example, real-world requests typically exhibit highly variable computational demands due to differences in task types and prompt lengths. Such variability renders simple static load balancing approaches ineffective, and so we need to use finer-grained strategies that can adapt to runtime conditions. The problem becomes even more complicated when we deploy the system on heterogeneous hardware and there are strict latency constraints.

In the development of LLMs, parallelization is closely related to LLM serving. Generally, building a high-quality LLM serving system is not a simple task — it typically requires the combination of multiple techniques, such as architectural design, workload distribution, and LLM-specific hardware/software optimizations. As such, LLM serving constitutes an exceptionally broad subject that often demands substantial engineering expertise. Here, we will not go into the details of LLM serving. For related concepts and techniques, readers may refer to relevant open-source systems (such as vLLM⁴, TensorRT-LLM⁵ and TGI⁶) and papers [Pope et al., 2023; Li et al., 2024a].

5.2.4 Remarks

We have considered many methods for improving the efficiency of LLMs in this and previous chapters. Although these approaches address different issues, most of them essentially explore trade-offs between various performance factors. One important trade-off is between inference speed and accuracy. For example, techniques like quantization, pruning, and knowledge distillation can significantly reduce computational overhead and latency but may introduce minor degradations in model performance. Conversely, preserving full precision or using larger models enhances accuracy but at the cost of slower inference and higher resource demands.

Another important consideration in LLM inference is the memory-compute trade-off. As in computer system design, we need to consider the balance between memory usage and computation required to generate the output. In particular, storing intermediate results such as KV caches during inference can significantly reduce redundant computation, but at the cost of increased memory usage. In KV caching, storing past attention states avoids recomputation of self-attention over previous tokens, thereby reducing compute time per token. However, as the number of tokens grows, so does the memory footprint of the KV cache, especially when processing very long sequences or multiple sequences in parallel. In response, various techniques have been developed to reduce memory consumption by partially recomputing intermediate states. For instance, chunked or windowed attention limits the attention span to a recent subset of tokens, reducing KV cache size at the cost of reduced context or additional compute if past information must be reprocessed.

⁴<https://github.com/vllm-project/vllm>

⁵<https://github.com/NVIDIA/TensorRT-LLM>

⁶<https://github.com/huggingface/text-generation-inference>

Note that considering the memory-compute trade-off is a very general principle. It can be extended beyond attention mechanisms and Transformers to other components in system design. An example is the choice of data precision. Using lower-precision formats such as FP16 or INT8 can reduce both memory usage and memory bandwidth requirements, effectively alleviating pressure on the memory subsystem. However, lower precision may lead to numerical instability or slight accuracy degradation, requiring careful calibration or retraining. Thus, this trade-off can also be seen as a memory-compute-accuracy triangle, where improvements in one dimension may come at the expense of another.

Beyond speed, accuracy, and memory, several other dimensions also influence LLM inference efficiency. Some of these dimensions have been discussed in this chapter, while others have not. Here we outline them as follows.

- **Throughput vs. Latency:** In large-scale multi-user LLM serving scenarios, we often aim to maximize system throughput. For example, as discussed in this section, we can batch multiple requests together to increase the number of tokens processed at the same time. However, batching increases waiting time and may lead to higher per-request latency, especially for short or interactive requests. By contrast, optimizing for low latency often requires serving requests individually or in smaller batches, which underutilizes hardware resources and reduces throughput. Achieving a good balance depends on the quality-of-service requirements and user interaction patterns.
- **Generalization vs. Specialization:** General-purpose LLMs are trained to perform a wide range of tasks with a single set of parameters. While flexible, they may be less efficient or accurate for specific tasks. Specialized models can yield better performance and lower inference costs for targeted applications. However, maintaining multiple specialized models increases system complexity and storage requirements. The trade-off between maintaining a single general model versus multiple specialized models is an important system-level design choice.
- **Energy Efficiency vs. Performance:** High-performance inference often requires running large models at high throughput on powerful accelerators, which consumes considerable energy. This may be problematic for edge deployments or energy-sensitive environments. Techniques like model compression can improve energy efficiency, but usually with some degradation in output quality or increase in latency. Energy constraints thus introduce another important dimension in optimizing LLM inference.

5.3 Inference-time Scaling

Scaling laws can be considered one of the fundamental principles guiding the development of LLMs. In previous chapters, we discussed several times that scaling up training data, model size, and compute can effectively improve the performance of pretraining. In fact, scaling laws also apply to downstream stages such as fine-tuning and inference (see Figure 5.14). Here we consider **inference-time scaling**, which has been widely employed by recent LLMs to solve complex

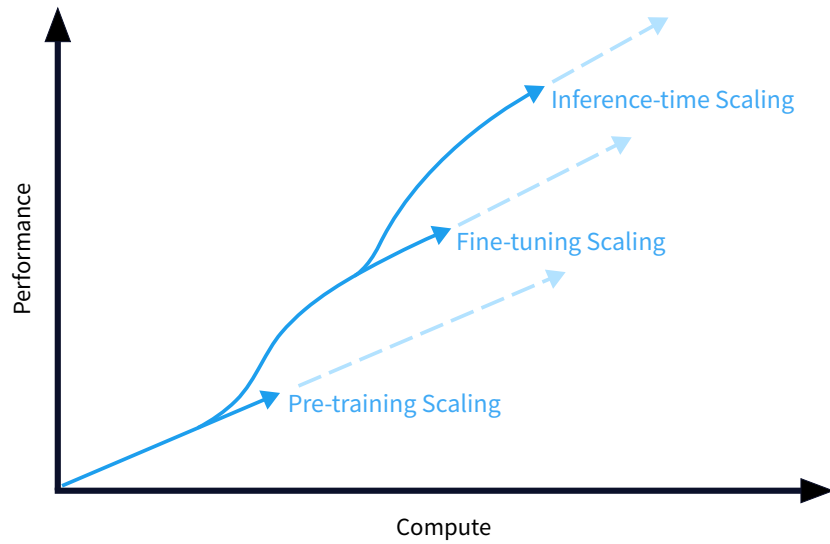


Fig. 5.14: Scaling for pre-training, fine-tuning and inference stages [Briski, 2025].

problems, such as complex math problems [Snell et al., 2025]. Unlike pre-training and fine-tuning scaling, which focuses on improving LLMs via parameter updates, inference-time scaling improves these models during inference without further training. This includes a large variety of methods which scale LLMs in different dimensions, such as ensembling multiple model outputs, increasing context length, adopting more aggressive decoding algorithms, and using external tools to extend model capabilities.

While inference-time scaling is wide-ranging, in this section we consider those methods that incorporate more compute into inference (called inference-time compute scaling). Here is a list of inference-time (test-time) compute scaling methods, organized by category:

- **Context Scaling.** It involves scaling the input or context to improve generation (or potentially scale the output).
- **Search Scaling.** It involves increasing computational effort during decoding.
- **Output Ensembling.** It involves combining multiple model outputs.
- **Generating and Verifying Thinking Paths.** It involves guiding LLMs to generate and verify thinking paths for solving complex reasoning problems.

We will describe these methods in the following subsections.

5.3.1 Context Scaling

Context scaling improves LLM performance by extending the input to the model. A straightforward approach is to incorporate more helpful context during inference, allowing the model to

condition its predictions on more prior information. One example is few-shot prompting. It augments the context with multiple input-output examples, and so the model can learn task behavior implicitly from these examples without parameter updates. On top of few-shot prompting, we can use chain-of-thought prompting to encourage the model to produce intermediate reasoning steps before final answers. Note that chain-of-thought prompting is one of the most important methods in addressing reasoning problems. By explicitly providing intermediate steps in problem-solving, we can prompt the model to break down complex tasks into simpler sub-tasks, which is found to be very beneficial for generating accurate and interpretable outputs.

Beyond extending the prompt with examples or reasoning steps, another approach to context scaling involves dynamically incorporating external knowledge. This is often achieved through RAG. RAG systems first retrieve relevant document snippets from a large collection of documents or a database based on the current input. These retrieved pieces of information are then added to the context provided to the LLM. This essentially expands the context to include timely or specialized external knowledge. By doing so, the model grounds its responses in specific knowledge found in the external source. The LLM thus can generate responses that are not only relevant to the input but also factually accurate and up-to-date.

However, as the context grows, these methods often suffer from the constraints of finite context window length. While model architectures and techniques (like efficient attention models) are continually evolving to support longer contexts, processing extremely long inputs still poses challenges. Increased computational cost is one factor. More critically, when the context window becomes very large, the model might struggle to attend effectively to the most relevant information (e.g., the “lost in the middle” phenomenon). Therefore, effective context scaling is not just about adding more information, but also about strategically selecting, structuring, and presenting the most pertinent information within the model’s processing capabilities.

Here we omit the detailed discussion of these methods, as they have already been covered in previous chapters. See Chapters 2 and 3 for more details, including prompting, RAG, and long-sequence modeling methods.

5.3.2 Search Scaling

In LLMs, decoding is a search process that aims to efficiently find the best output sequence given the input sequence. Search scaling (or decoding scaling) typically involves two aspects: scaling the output length and scaling the search space.

Scaling the output length refers to increasing the number of tokens generated during inference. This is especially important in tasks that require long-form generation, such as story writing. More recently, generating outputs with long thinking paths has shown strong performance in math problem solving and code generation. For example, encouraging the model to generate long thinking paths before producing the final answers has been found to be very beneficial in performing complex reasoning. This idea has been widely used in developing recent LLMs for reasoning, such as [OpenAI \[2024\]](#)’s o1 and [Deepseek \[2025\]](#)’s R1. We will discuss more about output length scaling in Section 5.3.4.

Scaling the search space, on the other hand, refers to expanding the set of candidate output sequences considered during search, so that higher-quality outputs can be found. As discussed in Section 5.1.3, a simple example is that in beam search we increase the beam width to allow more candidate sequences to be explored in parallel at each decoding step. This increases the chance

of discovering better outputs, especially in tasks where the optimal solution is not immediately apparent from local decisions.

In addition to decoding algorithm adjustments, it is also possible to explore compact structures to encode a large number of outputs. For example, we can construct and navigate a tree or graph of reasoning steps [Yao et al., 2024]. In this paradigm, each node represents a partial solution or intermediate step, and edges represent transitions between reasoning states. Such structured search enables the model to consider multiple paths simultaneously. Another related direction is Monte Carlo tree search-inspired decoding, where the model stochastically explores and scores different paths based on learned heuristics or external reward models.

Search scaling is a very general idea, and it is often implicitly involved in the design of search procedures that exploit search structure, heuristics, and model uncertainty. Many of the above methods have been discussed previously, though they were not originally developed with scaling as their primary goal. However, search scaling inherently comes with computational costs. Increasing beam width, for instance, directly translates to higher memory usage and longer inference times. In practice, there is often a point of diminishing returns, where further expansion of the search space yields marginal improvements in output quality at a significant computational expense. Therefore, an effective strategy often involves finding an optimal balance between scaling and computational feasibility.

5.3.3 Output Ensembling

If we have multiple model outputs, it is often beneficial to combine them to mitigate the impact of individual model errors and synthesize a superior final output. Each model might capture different aspects of the underlying data distribution or possess unique strengths and weaknesses. By ensembling, we can average out the noise or random errors present in individual predictions, leading to a more stable and reliable outcome. In LLM ensembling, one of the simplest approaches is to average the probability distributions over the next token from each model, and select the best token using this averaged distribution. Or, if we regard the problem as a discrete decision-making task, majority voting can be employed. More sophisticated methods might involve re-ranking candidate outputs generated by different models based on a separate scoring function or even using a meta-learner to intelligently combine the predictions.

The “scaling” from output ensembling comes at the cost of running multiple models or sampling multiple outputs. This not only increases the latency of inference but also leads to the additional complexity of managing multiple models. But the quality of outputs does not continue to improve indefinitely as more models are added. In some cases, the benefits of output ensembling may diminish as the number of component models in the ensemble exceeds a certain threshold. Instead, the benefits of ensembling are generally greater when the individual models are diverse (i.e., they make different errors), even if there are a relatively small number of component models. Therefore, it is common practice to use a set of diverse LLMs which differ in their training data, model architectures, or fine-tuning objectives.

In LLMs, “scaling” often implies making things “bigger” for quality with more resources. However, in addition to scaling up the quality, scaling can mean more. It can also signify scaling up the robustness (making the system less prone to errors and more reliable) and exploration (covering a wider range of potential solutions). In output ensembling, these dimensions are naturally integrated. For instance, the very act of averaging or voting across different model outputs is a

direct strategy to scale up robustness against individual model failures. Furthermore, by intentionally including varied models, ensembling increases the chances of discovering novel or superior solutions. In this sense, scaling is not limited to making models larger or running them longer — it also means strategies for making inference more robust, exploratory, and adaptive.

5.3.4 Generating and Verifying Thinking Paths

So far, we have viewed inference-time scaling as a general class of methods for scaling various aspects of inference, such as sequence length, model size, and/or search strategies. In fact, one successful application is the use of inference-time scaling to enhance the reasoning capabilities of LLMs. As we have seen, the reasoning performance of LLMs can be improved by using chain-of-thought methods. We can therefore make use of the chain-of-thought prompts to generate intermediate reasoning steps and reach a correct answer. However, reasoning problems are often so complicated that we cannot obtain high-quality solutions by providing simple chain-of-thought prompts. For example, when solving a math problem, we typically need to reason over a sequence of steps. At each step, we need to work out some intermediate result, verify it, and then determine what to do next. The reasoning path is not a fixed pattern but a dynamically generated thinking process that often involves trial-and-error, backtracking, and self-correction. This requires more sophisticated prompting strategies or search algorithms to navigate such complex reasoning. In this subsection, we focus on inference-scaling methods that go beyond simple chain-of-thought to address complex reasoning problems more effectively.

At a high level, methods for scaling the reasoning of LLMs can be categorized into two classes:

- **Training-free Methods.** These methods aim to improve reasoning capabilities without requiring any modification or retraining of the pre-trained parameters. Instead, they focus on techniques applied during inference, such as sophisticated prompting strategies (e.g., chain-of-thought) and algorithmic control over the reasoning process (e.g., search).
- **Training-based Methods.** These methods involve further training or fine-tuning the model parameters to explicitly improve reasoning abilities, such as supervised fine-tuning on datasets with reasoning examples (e.g., math problems with step-by-step solutions).

In the following, we first discuss training-free methods, and then training-based methods.

5.3.4.1 Solution-level Search with Verifiers

Given an input sequence (e.g., a math problem), there are many possible output sequences (e.g., solutions to the problem). If we have a model to evaluate or verify each solution, we can select the best one. This is the fundamental principle behind methods like best-of- N sampling, where multiple outputs are generated, and the optimal result is picked based on some selection mechanism. Such a selection process can be viewed as a search problem, which involves two components:

- **Search Algorithm.** This defines the strategy used to explore the space of possible output sequences (solutions) and generate a set of candidates. It can range from simple independent sampling to more sophisticated search techniques as discussed in Section 5.1.3.

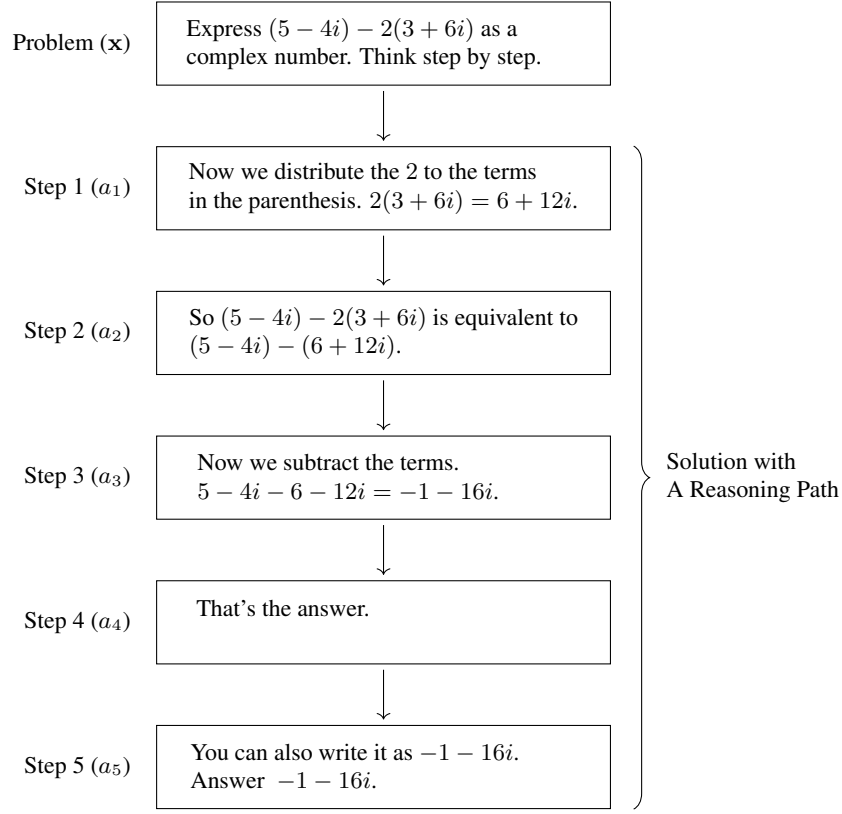


Fig. 5.15: Illustration of multi-step reasoning. This example is from the PRM800K dataset [Lightman et al., 2024]. Given a math problem, the LLM is prompted to generate a thinking path (or reasoning path) consisting of several reasoning steps. Each step addresses a sub-problem based on the results of the previous steps. The answer to the original problem is contained in the last step.

- **Verifier.** This is a model or function responsible for evaluating the quality, correctness, or utility of each candidate solution generated by the search algorithm. It provides a score, a probability, or a judgment that allows the system to select the best among the candidates. The verifier can be another LLM, or even a set of predefined rules or heuristics.

Given an input problem \mathbf{x} , we define that an output solution \mathbf{y} can be represented as a sequence of reasoning steps:

$$\mathbf{y} = (a_1, a_2, \dots, a_{n_r}) \quad (5.37)$$

where a_i is the i -th reasoning step, and a_{n_r} is the last step which should contain the answer to the problem. See Figure 5.15 for an example of a multi-step reasoning path.

The search algorithm can efficiently generate a set of candidate solutions

$$\mathcal{D}_c = \{\mathbf{y}_1, \dots, \mathbf{y}_K\} \quad (5.38)$$

Then, we can use a verifier, which evaluates each solution by the function $V(\mathbf{y})$, to score the candidates in \mathcal{D}_c . The final output is the best candidate selected by the verifier

$$\hat{\mathbf{y}} = \arg \max_{\mathbf{y} \in \mathcal{D}_c} V(\mathbf{y}) \quad (5.39)$$

Although verifying the entire reasoning path is possible, a simpler alternative is to verify only the final reasoning step. In this way the verifier function $V(\mathbf{y})$ is simplified to depend solely on the final answer contained within a_{n_r} . This can be achieved in various ways, depending on the nature of the problem and the expected answer format.

- For some math and coding problems, we can use off-the-shelf tools as verifiers. Examples include proof checkers for mathematical theorems, interpreters or compilers for code execution, and unit test systems for verifying program correctness against predefined test cases.
- If there is labeled data for evaluating the answer, such as human preference data, we can train a reward model on such data. The learned reward model is then used as the verifier which assigns a scalar score to each candidate answer.
- If there are no existing systems or suitable reward models, we can use another LLM to act as the verifier. This LLM is prompted to assess the quality of the candidate answer. It could potentially be a more capable model, or the same LLM used with a specific “evaluator” prompt.
- Alternatively, simpler heuristic-based verifiers can be designed. A commonly used approach is to employ majority voting, where the most frequently occurring answer among a set of candidates is selected.

Based on these verifiers, we can search to obtain a set of candidate solutions for selection. One simple strategy, which is often referred to as **parallel scaling** [Brown et al., 2024; Snell et al., 2024], involves generating K candidate solutions by running the base LLM K times independently. In this process, we can adjust the temperature in sampling to control the diversity in the outputs. The verifier then assesses each of these K complete solutions, and the one with the highest score is selected as the final output. This is conceptually very similar to best-of- N sampling, which in previous chapters we primarily described as a method of selecting the best one from a set of sampled outputs using a reward model.

Another approach is **sequential scaling**, which builds a sequence of solutions incrementally [Gou et al., 2024; Zhang et al., 2024]. It starts with an initial solution generated by the LLM with prompting. Then, we use a verifier (often the same LLM) to evaluate the solution. This can be seen as a critique stage. The output of this stage is some form of feedback, such as textual critiques pinpointing errors or suggesting improvements, numerical scores reflecting solution quality, or even a revised plan or intermediate step to guide the next generation. This feedback, along with the original problem and the current solution, is then used to prompt the LLM to generate a potentially improved solution. This can be seen as a refine stage. This critique-refine cycle can be repeated, forming an iterative loop:

$$\mathbf{y}_{k+1} = \text{Refine}(\mathbf{x}, \mathbf{y}_k, \text{Feedback}(\mathbf{y}_k)) \quad (5.40)$$

where $\text{Feedback}(\mathbf{y}_k)$ represents the feedback from the verifier. The $\text{Refine}(\cdot)$ function generates the improved solution \mathbf{y}_{k+1} by prompting the LLM with the original problem \mathbf{x} , the previous solution \mathbf{y}_k , and this feedback. The process can be iterated for K times, or until the solution

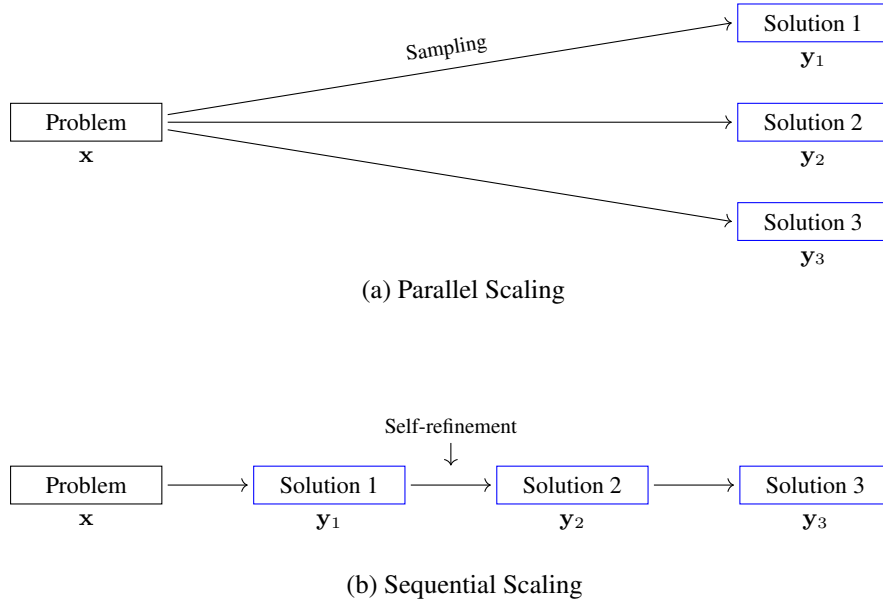


Fig. 5.16: Illustrations of parallel scaling and sequential scaling. In parallel scaling, we obtain multiple solutions by running the LLM several times independently. In sequential scaling, the LLM generates an initial solution. Then, we use the LLM to refine it iteratively, with each refinement yielding a new, possibly better solution.

quality, as assessed by the verifier, converges to a satisfactory level. This iterative framework, where a solution is progressively improved through cycles of generation, evaluation (critique), and revision, is precisely what constitutes self-refinement [Shinn et al., 2023; Madaan et al., 2024]. In such scenarios, the role of the verifier is not just to pick the best complete solution from a static set, but to actively guide the generation process itself.

See Figure 5.16 for illustrations of parallel scaling and sequential scaling. Note that there are other ways to perform search and obtain different sets of candidate solutions. One alternative method is to organize search as a tree structure. This approach, often referred to as tree search, provides a more structured way to explore the space of possible reasoning paths. In solution-level search, each node of the tree represents a complete solution. During search, we need to expand a node to a set of child nodes, representing new solutions that can be considered in verification. The expansion process typically involves taking an existing solution (the parent node) and using the LLM to generate variations or alternative solutions.

5.3.4.2 Step-level Search with Verifiers

While the methods discussed above primarily focus on generating complete solutions before final selection, the search process can also be integrated more deeply into the step-by-step generation of the reasoning path itself. This leads to approaches that perform step-level search with verifiers, where guidance or pruning occurs at intermediate reasoning steps $\{a_1, \dots, a_{n_k}\}$ rather than only after a full solution y is formed.

Such fine-grained control is particularly beneficial for complex reasoning problems where a single incorrect intermediate step can render the entire subsequent reasoning chain invalid. By evaluating or guiding the generation at each intermediate step, the LLM can explore the reasoning space more effectively, potentially pruning unpromising paths early or allocating more resources to explore more plausible ones.

Step-level search with verifiers can also be modeled as a tree search problem. In this paradigm, each node (or state) corresponds to a partial reasoning path, $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$, representing the sequence of i reasoning steps taken so far (i.e., a path from the root node to the current node). The objective of the search process is to explore the underlying state space, starting from an initial empty path, to find a complete path that constitutes a correct solution. Note that we use $\mathbf{a}_{\leq i}$ here to represent a partial reasoning path instead of $\mathbf{y}_{\leq i}$. While this makes notation a bit inconsistent with that used for representing complete solutions (\mathbf{y}) or full paths in solution-level search, it serves to highlight the focus on individual actions or steps.

The core components of step-level search with verifiers are:

- **Node Representation.** A node is a partial reasoning path $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$. The root node is an empty path, and terminal nodes are complete reasoning paths.
- **Node Expansion.** Given a current partial path $\mathbf{a}_{\leq i}$, the LLM is used to generate one or more candidate next reasoning steps $\{a_{i+1}^{(1)}, \dots, a_{i+1}^{(M)}\}$. Each candidate step, when appended to $\mathbf{a}_{\leq i}$, forms a new potential partial path $\mathbf{a}_{\leq i+1} = (a_1, \dots, a_i, a_{i+1}^{(j)})$.
- **Verification.** The verifier $V(\cdot)$ evaluates the quality of a newly generated step in the context of the current partial path $\mathbf{a}_{\leq i} = (a_1, \dots, a_i)$ and the original problem \mathbf{x} . As with solution-level verification, step-level verifiers might output a numerical score, a categorical label, and textual feedback.
- **Search.** This governs how the search space is explored. Based on the evaluations from the verifier, the search strategy decides which partial paths to extend further, which to prune, and the order of exploration.

This step-by-step verification allows for dynamic adjustments to the reasoning process. If a step a_{i+1} is deemed incorrect or unpromising by $V(\cdot)$, the search algorithm can backtrack and explore alternative steps from $\mathbf{a}_{\leq i}$, or even from an earlier node $\mathbf{a}_{\leq i'}$ (where $i' < i$). Conversely, if a step is highly rated, resources can be focused on extending that path. See Figure 5.17 for an illustration of step-level search with verifiers.

Clearly, this search framework is very similar to that used in decoding methods for LLMs, as discussed in Section 5.1.3. For example, beam search maintains a set of K most promising partial sequences at each generation step. This is a form of step-level search where the “verifier” is implicitly the LLM’s own probability model, and the “search” is the pruning mechanism to maintain the beam size.

However, step-level search with explicit verifiers, as described here, presents differences from standard decoding. One of them is that the verifier can be a much more sophisticated component than just the raw output probabilities of the generative LLM. The design of step-level verifiers basically follows that of solution-level verification. A step-level verifier might be a language model that assesses the quality of an individual reasoning step within the context of the preceding path. This LLM can even be fine-tuned to enhance its verification capability. Alternatively, for domains with well-defined rules, it could be a symbolic engine or a set of programmatic checks. Furthermore, verifiers can be designed to predict the future utility or likelihood of success given the current partial path, drawing inspiration from value functions in reinforcement learning. Human

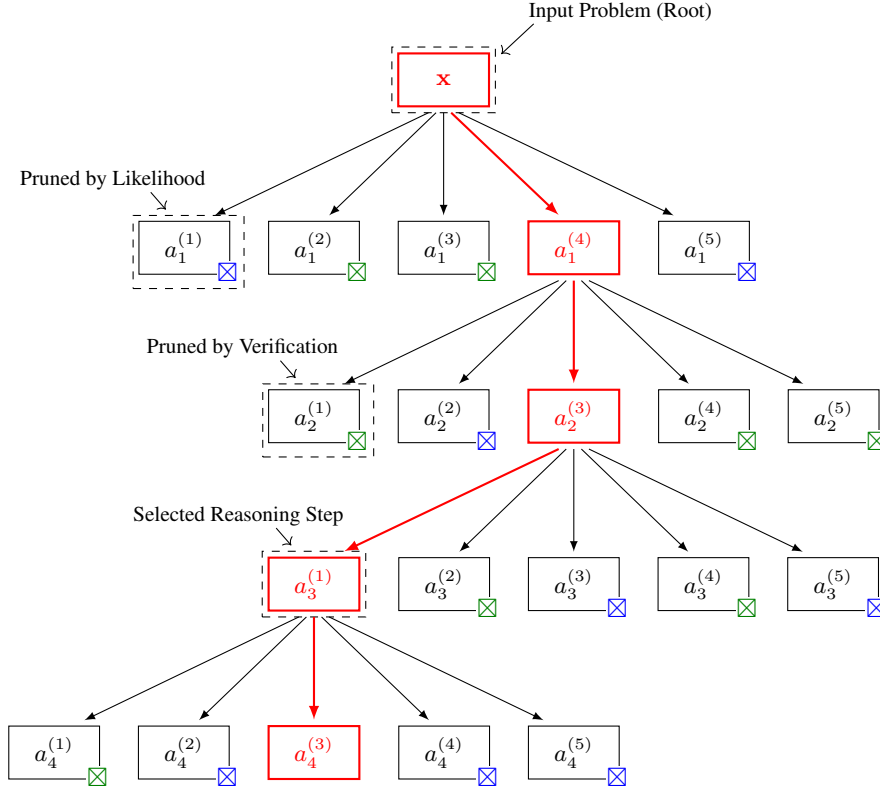


Fig. 5.17: Illustration of step-level search with verifiers. $a_i^{(j)}$ = the j -th candidate for the i -th reasoning step, \boxtimes = candidate pruned by the LLM’s output probability, and \boxtimes = candidate pruned by the verifier. Given the input problem as the root node, we expand the tree by generating multiple reasoning steps at each expansion. Each candidate can be pruned by either likelihood (as in standard decoding) or step-level verification. The unpruned candidates are then expanded to generate further reasoning steps. The process is iterated until a complete reasoning chain leading to a final answer is generated, or until a predefined search limit is reached.

expertise can also be incorporated to provide judgments on critical steps, especially in high-stakes scenarios.

One example of such a step-level verifier, particularly when using human feedback to assess intermediate progress, is the **process reward model (PRM)**. A PRM is typically a separate language model trained to output a scalar reward for each reasoning step $a_{i'}$ within a partial path $\mathbf{a}_{\leq i}$. It provides a more direct and fine-grained supervisory signal compared to **outcome reward models (ORMs)** which only evaluate the final solution. However, the development of PRMs relies on step-level human annotations, such as preferences on different next steps. Collecting supervision for each intermediate step is considerably more labor-intensive and requires greater cognitive effort from human annotators than simply labeling final outcomes.

One alternative approach to developing training data for step-level verification is to use LLMs to generate such annotations automatically. For example, we can take a strong LLM, referred to as a teacher model, and prompt it to first generate a complete reasoning path for a given problem. Then, at each intermediate step within this path, we can prompt the same teacher LLM (or another capable LLM) to generate several alternative candidate next steps in addition to the one it originally chose. The teacher LLM can then be prompted again to evaluate these alternatives. These evaluation results (e.g., correct vs. incorrect) can then serve as data annotations. Alternatively, the generalization capabilities of PRMs can be leveraged. We can train a PRM on tasks where step-level verification is easier and then generalize this PRM to other tasks with little or no additional

training.

Note that step-level verification also comes with its own problems. Frequent verification, especially if using an LLM as the verifier, can substantially increase computational costs and latency. The design of effective step-level verifiers is non-trivial itself. An inaccurate verifier might prematurely discard good reasoning paths or fail to identify flawed ones, thereby misleading the search. This makes the development of such systems more complex and difficult.

5.3.4.3 Encouraging Long Thinking

So far in this subsection, most of the methods are implicitly based on a simple idea: generating longer reasoning paths can help. In addition to CoT and search with verifications, we can consider alternative methods to achieve this. For example, we can prompt the LLM by explicitly asking for extended deliberation. Beyond direct prompting, we can also make modifications to the decoding process itself, such as adjusting token limits or applying penalties for short outputs. Another approach is to employ multi-stage generation schemes where the model incrementally builds upon its reasoning.

5.3.4.4 Training-based Scaling

As well as considering inference-time scaling methods without training, we also wish to consider methods that can improve intrinsic reasoning capabilities of LLMs by modifying their parameters through further training. While such training-based scaling methods typically require additional training cost and computational resources, they instill stronger reasoning skills directly into the model parameters, which in turn can lead to more effective and efficient reasoning performance. We can even combine them with training-free methods for better inference-time scaling results.

Although our discussion here is restricted to reasoning problems, methods for training-based scaling are common. Most of them have been discussed in Chapter 4. Here, we will briefly describe how these methods can be applied to improving inference-time scaling for reasoning problems.

- **Fine-tuning on Reasoning Data.** One of the most direct ways to enhance reasoning is by fine-tuning pre-trained LLMs on datasets specifically curated for reasoning tasks. These datasets can range from simple input-output pairs to more structured formats that include step-by-step reasoning processes. Typical examples include datasets of math word problems, logical deduction exercises, or code generation with explanations. By training on such data, the model learns from common reasoning patterns, and thus can generate detailed and coherent reasoning paths at test time.
- **Reinforcement Learning for Reasoning.** If we regard a verifier as a reward model, we can see that the methods discussed in the previous subsection are a direct application of the reward model to reasoning problems, though they are training-free. Of course, we can apply this reward model to LLM fine-tuning. This follows a standard paradigm of reinforcement learning. Given a reward model, the LLM, acting as a policy, is fine-tuned using reinforcement learning algorithms. The LLM generates reasoning steps or full solutions, receives feedback (rewards) from the reward model, and updates its parameters to produce outputs

that maximize these rewards. This process aligns the LLM output with notions of high-quality reasoning, thereby encouraging the LLM to generate more reliable reasoning paths. Another key issue is the training of the reward model. Generally, this reward model could be an outcome reward model that evaluates the correctness or quality of the final answer, or a process reward model that assesses the quality of each intermediate reasoning step, as discussed in the context of step-level verifiers. In some cases, we can even develop a reward model based on simple rules, such as giving bonuses to longer outputs.

- **Knowledge Distillation for Reasoning.** In this approach, a smaller, more efficient student LLM is trained to mimic the reasoning outputs or internal representations of a larger, more capable teacher LLM. The teacher model might generate detailed reasoning steps for a variety of problems. The student model then learns to reproduce these high-quality reasoning demonstrations. This strategy makes stronger reasoning capabilities more accessible by deploying them in smaller models that are less computationally expensive at inference time.
- **Iterative Refinement.** Training-based scaling can also involve iterative refinement. For example, an LLM can generate solutions to a set of problems. These solutions and their reasoning paths are then verified, either by humans or automatic verifiers. The correct reasoning paths are subsequently added to the training data, and the LLM is further fine-tuned on this augmented dataset. This creates a cycle where the LLM progressively improves its reasoning capabilities through repeated generation, critique, and learning.

The primary advantage of these training-based scaling methods is that they endow the LLM with stronger inherent reasoning skills. This directly contributes to improved inference-time scaling in several ways: it can lead to more efficient inference, as the LLM might require less extensive search or fewer generation samples to arrive at a correct solution. Moreover, the base quality of generated steps or solutions is higher. Therefore, a well-trained LLM might generalize its learned reasoning abilities to novel problems more effectively than an LLM relying solely on in-context learning or training-free inference schemes.

On the other hand, training-based approaches also present challenges, compared to the training-free counterparts. The creation of high-quality, large-scale training datasets for reasoning can be expensive and labor-intensive. The fine-tuning process itself, particularly for the largest LLMs or when using RL, can be computationally intensive and require substantial engineering effort. There is also the risk of the model overfitting to the specific types of problems or reasoning styles present in the training data, potentially limiting its performance on out-of-distribution tasks.

5.4 Summary

In this chapter, we have discussed the inference issue for LLMs. We have presented the prefilling-decoding framework and related decoding algorithms for LLM inference. Then, we have described several techniques for efficient inference. We have also discussed inference-time scaling, which has been considered one of the most important methods for improving LLM reasoning.

Inference over sequential data has long been a concern in AI [Wozengraft and Reiffen, 1961; Viterbi, 1967; Forney, 1972]. In the context of NLP, this line of work dates back to the very early days of speech recognition and statistical machine translation [Koehn, 2010], where researchers

faced the challenge of efficiently searching vast hypothesis spaces to find the most probable output sequence. Techniques like beam search and various pruning strategies were developed then to make this computationally tractable. At that time, models were relatively weak, and much of the research focused on developing powerful search algorithms to reduce search errors. These foundational ideas continue to influence modern approaches.

As we enter the era dominated by deep learning methods, models based on deep neural networks have become extremely powerful. Even with very simple search algorithms, these models can achieve excellent results. In this context, inference no longer seems as “important” as it once was, and research attention has gradually shifted toward model architectures, training methods, and scaling up models.

However, history tends to repeat itself. With the rise of LLMs, inference has once again attracted significant attention. This renewed focus is primarily manifested in two aspects:

- The inference cost for LLMs is very high. For example, efficiently deploying LLMs in high-concurrency, low-latency scenarios remains a challenging problem, making inference efficiency critically important. In this context, efficient architecture designs, optimized search algorithms, and various inference optimization strategies hold substantial practical significance.
- Input and output sequence lengths have significantly increased in complex tasks. Especially in tasks like mathematical reasoning, the growth of sequence lengths further highlights the importance of inference efficiency. Moreover, scaling the inference process has recently proven to be an effective way to improve the reasoning capabilities of models. Therefore, achieving efficient inference scaling is emerging as a particularly promising research direction.

Inference is now a wide-ranging topic that encompasses many techniques. It involves not only the development of model architectures and decoding algorithms, but is increasingly shaped by the intricate engineering and sophisticated systems-level optimizations required to deploy LLMs effectively and efficiently. Many of these techniques are beyond the scope of NLP or a specific AI area. Instead, the frontier of LLM inference optimization now extends deeply into domains traditionally considered core computer science and engineering. This systemic perspective has brought many new ideas to the study of inference problems. Unfortunately, this chapter cannot cover all relevant techniques — indeed, that would be an almost impossible task in itself. Ultimately, the best way to better understand and master these techniques may still lie in hands-on practice.

Bibliography

- [Agrawal et al., 2023] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.
- [Agrawal et al., 2024] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} trade-off in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.
- [Ainslie et al., 2020] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. Etc: Encoding long and structured inputs in transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 268–284, 2020.
- [Ainslie et al., 2023] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.
- [Akyürek et al., 2023] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Alabdulmohsin et al., 2022] Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting neural scaling laws in language and vision. *Advances in Neural Information Processing Systems*, 35: 22300–22312, 2022.
- [Allal et al., 2024] Loubna Ben Allal, Anton Lozhkov, and Daniel van Strien. cosmopedia: how to create large-scale synthetic data for pre-training. <https://huggingface.co/blog/cosmopedia>, 2024.
- [Almazrouei et al., 2023] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.
- [Andreas et al., 2016] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48, 2016.
- [Arjovsky et al., 2016] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International conference on machine learning*, pages 1120–1128, 2016.
- [Aschenbrenner, 2024] Leopold Aschenbrenner. Situational awareness: The decade ahead, 2024. URL <https://situational-awareness.ai/>.
- [Askell et al., 2021] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.
- [Bach et al., 2022] Stephen H. Bach, Victor Sanh, Zheng Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Févry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged Saeed AlShaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir R. Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. Promptsources: An integrated development environment and repository for natural language prompts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 93–104, 2022.

- [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.
- [Bengio et al., 2024] Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Trevor Darrell, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, Gillian K. Hadfield, Jeff Clune, Tegan Maharaj, Frank Hutter, Atilim Gunes Baydin, Sheila A. McIlraith, Qiqi Gao, Ashwin Acharya, David Krueger, Anca Dragan, Philip Torr, Stuart Russell, Daniel Kahneman, Jan Markus Brauner, and Sören Mindermann. Managing extreme ai risks amid rapid progress. *Science*, 384(6698): 842–845, 2024.
- [Bentivogli and Giampiccolo, 2011] Luisa Bentivogli and Danilo Giampiccolo. Pascal recognizing textual entailment challenge (rte-7) at tac 2011. <https://tac.nist.gov/2011/RTE/>, 2011.
- [Besta et al., 2024] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoeffler. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.
- [Biderman et al., 2021] Stella Biderman, Sid Black, Charles Foster, Leo Gao, Eric Hallahan, Horace He, Ben Wang, and Phil Wang. Rotary embeddings: A relative revolution. <https://blog.eleuther.ai/rotary-embeddings/>, 2021.
- [Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 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.
- [Bradley and Terry, 1952] Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- [Brandon et al., 2024] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*, 2024.
- [Brill, 1992] Eric Brill. A simple rule-based part of speech tagger. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.
- [Briski, 2025] Kari Briski. How scaling laws drive smarter, more powerful ai, 2025.
- [Brown et al., 2024] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.
- [Brown et al., 1993] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.
- [Brown et al., 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments

- with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.
- [Bulatov et al., 2022] Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. *Advances in Neural Information Processing Systems*, 35:11079–11091, 2022.
- [Burges et al., 2005] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.
- [Burns et al., 2023] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023a.
- [Burns et al., 2023] Collin Burns, Jan Leike, Leopold Aschenbrenner, Jeffrey Wu, Pavel Izmailov, Leo Gao, Bowen Baker, and Jan Hendrik Kirchner. Weak-to-strong generalization, 2023b. URL <https://openai.com/index/weak-to-strong-generalization>.
- [Caballero et al., 2023] Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken neural scaling laws. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2023.
- [Cao et al., 2007] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.
- [Chang et al., 2024] Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv:2404.01077*, 2024.
- [Charniak, 1997] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997.
- [Chen et al., 2023] Banghao Chen, Zhaofeng Zhang, Nicolas Langren , and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023a.
- [Chen et al., 2023] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpapasus: Training a better alpaca with fewer data. *arXiv preprint arXiv:2307.08701*, 2023b.
- [Chen et al., 2024] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpapasus: Training a better alpaca with fewer data. In *The Twelfth International Conference on Learning Representations*, 2024a.
- [Chen et al., 2023] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023c.
- [Chen et al., 2020] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in neural information processing systems*, 33:15834–15846, 2020.
- [Chen et al., 2024] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024b.
- [Chevalier et al., 2023] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3829–3846, 2023.
- [Chi et al., 2022] Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. Kerple:

- Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems*, 35:8386–8399, 2022.
- [Chi et al., 2023] Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. Dissecting transformer length extrapolation via the lens of receptive field analysis. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13522–13537, 2023.
- [Chiang et al., 2023] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL <https://lmsys.org/blog/2023-03-30-vicuna/>.
- [Chowdhery et al., 2022] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [Christiano et al., 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.
- [Chu et al., 2023] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*, 2023.
- [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.
- [Clark et al., 2019] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *Proceedings of International Conference on Learning Representations*, 2019.
- [Cobbe et al., 2021] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukas 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.
- [Conneau et al., 2020] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, 2020.
- [Coste et al., 2024] Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Cui et al., 2024] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. ULTRAFEEDBACK:

- Boosting language models with scaled AI feedback. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 9722–9744, 2024.
- [Dai et al., 2023] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta-optimizers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4005–4019, 2023.
- [Dai et al., 2019] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.
- [Dao et al., 2022] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.
- [Deepseek, 2025] Deepseek. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.
- [Deletang et al., 2024] Gregoire Deletang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Deng et al., 2022] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3369–3391, 2022.
- [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.
- [Ding et al., 2024] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.
- [Dolan and Brockett, 2005] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of Third International Workshop on Paraphrasing (IWP2005)*, 2005.
- [Dong et al., 2019] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems*, 32, 2019.
- [Dong et al., 2022] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.
- [Dong et al., 2021] Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In *International Conference on Machine Learning*, pages 2793–2803. PMLR, 2021.
- [Drozdo et al., 2022] Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. Compositional semantic parsing with large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2022.
- [Dua et al., 2022] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1251–1265, 2022.

- [Dubey et al., 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [Dubois et al., 2024] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. AlpacaFarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Eisenstein et al., 2023] Jacob Eisenstein, Chirag Nagpal, Alekh Agarwal, Ahmad Beirami, Alex D’Amour, DJ Dvijotham, Adam Fisch, Katherine Heller, Stephen Pfohl, Deepak Ramachandran, and Peter Shaw. Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking. *arXiv preprint arXiv:2312.09244*, 2023.
- [Elsken et al., 2019] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.
- [Erhan et al., 2010] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208, 2010.
- [Fan et al., 2018] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, 2018.
- [Fan et al., 2019] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *Proceedings of International Conference on Learning Representations*, 2019.
- [Fedus et al., 2022] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.
- [Fernandes et al., 2023] Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. Bridging the gap: A survey on integrating (human) feedback for natural language generation. *Transactions of the Association for Computational Linguistics*, 11:1643–1668, 2023.
- [Forney, 1972] GDJR Forney. Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. *IEEE Transactions on Information theory*, 18(3):363–378, 1972.
- [Franklin and Graesser, 1996] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International workshop on agent theories, architectures, and languages*, pages 21–35. Springer, 1996.
- [Frensch and Funke, 2014] Peter A Frensch and Joachim Funke. *Complex problem solving: The European perspective*. Psychology Press, 2014.
- [Gale et al., 2019] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.
- [Ganguli et al., 2023] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamile Lukosiute, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, Dawn Drain, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jackson Kernion, Jamie Kerr, Jared Mueller, Joshua Landau, Kamal Ndousse, Karina Nguyen, Liane Lovitt, Michael Sellitto, Nelson Elhage, Noemí Mercado, Nova DasSarma, Oliver Rausch, Robert Lasenby, Robin Larson, Sam Ringer, Sandipan Kundu, Saurav Kadavath, Scott Johnston, Shauna Kravec, Sheer El Showk, Tamera Lanham, Timothy Telleen-Lawton, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, Christopher Olah, Jack Clark, Samuel R. Bowman, and Jared Kaplan. The capacity for moral self-correction in large language models. *arXiv preprint arXiv:2302.07459*, 2023.
- [Gao et al., 2023] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR, 2023a.

- [Gao et al., 2023] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023b.
- [Gao et al., 2023] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023c.
- [Garg et al., 2022] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.
- [Ge et al., 2024] Yuan Ge, Yilun Liu, Chi Hu, Weibin Meng, Shimin Tao, Xiaofeng Zhao, Hongxia Ma, Li Zhang, Boxing Chen, Hao Yang, Bei Li, Tong Xiao, and Jingbo Zhu. Clustering and ranking: Diversity-preserved instruction selection through expert-aligned quality estimation. *arXiv preprint arXiv:2402.18191*, 2024.
- [Gemma Team, 2024] Google DeepMind Gemma Team. Gemma: Open Models Based on Gemini Research and Technology, 2024.
- [Goodhart, 1984] Charles AE Goodhart. *Problems of monetary management: the UK experience*. Springer, 1984.
- [Gordon et al., 2021] Mitchell A Gordon, Kevin Duh, and Jared Kaplan. Data and parameter scaling laws for neural machine translation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5915–5922, 2021.
- [Gou et al., 2024] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujia Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Gu and Dao, 2023] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.
- [Gunasekar et al., 2023] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.
- [Guo et al., 2024] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujia Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Gupta and Berant, 2020] Ankit Gupta and Jonathan Berant. Gmat: Global memory augmentation for transformers. *arXiv preprint arXiv:2006.03274*, 2020.
- [Gupta et al., 2021] Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, pages 39–52, 2021.
- [Han et al., 2021] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. Pre-trained models: Past, present and future. *AI Open*, 2:225–250, 2021.
- [Han et al., 2024] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.
- [Harlap et al., 2018] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.
- [He et al., 2019] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In

- Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.
- [He et al., 2021] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced bert with disentangled attention. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Hendrycks and Gimpel, 2016] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [Hendrycks et al., 2020] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedziec, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2744–2751, 2020.
- [Hendrycks et al., 2021] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Hestness et al., 2017] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.
- [Hewitt, 2024] John Hewitt. Instruction following without instruction tuning, 2024. URL <https://nlp.stanford.edu/~johnhew/instruction-following.html>.
- [Hewitt et al., 2024] John Hewitt, Nelson F Liu, Percy Liang, and Christopher D Manning. Instruction following without instruction tuning. *arXiv preprint arXiv:2409.14254*, 2024.
- [Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [Hoffmann et al., 2022] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- [Holtzman et al., 2020] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.
- [Honovich et al., 2023] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14409–14428, 2023.
- [Houlsby et al., 2019] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.
- [Hu et al., 2022] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.
- [Huang, 2009] Liang Huang. Dynamic programming-based search algorithms in NLP. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, 2009.
- [Huang et al., 2019] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyoungho Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- [Hutchins et al., 2022] DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. *Advances in neural information processing systems*, 35:

33248–33261, 2022.

- [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, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023a.
- [Jiang et al., 2023] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LlmLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, 2023b.
- [Jiang et al., 2020] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438, 2020.
- [Jiao et al., 2020] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, 2020.
- [Joshi et al., 2017] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.
- [Joshi et al., 2020] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics*, 8:64–77, 2020.
- [Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. *Speech and Language Processing (2nd ed.)*. Prentice Hall, 2008.
- [Kahneman, 2011] Daniel Kahneman. *Thinking, fast and slow*. macmillan, 2011.
- [Kaplan et al., 2020] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [Katharopoulos et al., 2020] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and Fran  ois Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.
- [Khandelwal et al., 2020] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.
- [Khot et al., 2023] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Kim et al., 2023] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*, 2023.
- [Kirkpatrick et al., 2017] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

- [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.
- [Korthikanti et al., 2023] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.
- [Krakovna et al., 2020] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. <https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity>, 2020.
- [Kumar and Byrne, 2004] Shankar Kumar and William Byrne. Minimum bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 169–176, 2004.
- [Kung and Peng, 2023] Po-Nien Kung and Nanyun Peng. Do models really learn to follow instructions? an empirical study of instruction tuning. *arXiv preprint arXiv:2305.11383*, 2023.
- [Kwon et al., 2023] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.
- [Lake and Baroni, 2018] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.
- [Lambert et al., 2024] Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Han-naneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.
- [Lample and Conneau, 2019] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.
- [Lan et al., 2020] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Lee et al., 2023] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.
- [Lester et al., 2021] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, 2021.
- [Leviathan et al., 2023] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.
- [Lewis et al., 2020] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020.
- [Li et al., 2024] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391*, 2024a.

- [Li et al., 2023] Bei Li, Rui Wang, Junliang Guo, Kaitao Song, Xu Tan, Hany Hassan, Arul Menezes, Tong Xiao, Jiang Bian, and JingBo Zhu. Deliberate then generate: Enhanced prompting framework for text generation. *arXiv preprint arXiv:2305.19835*, 2023a.
- [Li, 2011] Hang Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Online access: Morgan & Claypool Synthesis Collection Five. Morgan & Claypool Publishers, 2011. ISBN 9781608457076.
- [Li et al., 2022] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110*, 2022.
- [Li et al., 2024] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. In *The Twelfth International Conference on Learning Representations*, 2024b.
- [Li et al., 2023] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023b.
- [Li and Liang, 2021] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.
- [Li, 2023] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, pages 641–647, 2023.
- [Li et al., 2023] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6342–6353, 2023c.
- [Lialin et al., 2023] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023.
- [Lightman et al., 2024] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Liu et al., 2024] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.
- [Liu et al., 2022] Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, 2022.
- [Liu et al., 2023] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023a.
- [Liu et al., 2024] Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*, 2024b.
- [Liu, 2009] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.
- [Liu et al., 2023] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023b.

- [Liu et al., 2023] Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. *arXiv preprint arXiv:2311.12785*, 2023c.
- [Liu et al., 2024] Xinyu Liu, Runsong Zhao, Pengcheng Huang, Chunyang Xiao, Bei Li, Jingang Wang, Tong Xiao, and Jingbo Zhu. Forgetting curve: A reliable method for evaluating memorization capability for long-context models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4667–4682, 2024c.
- [Liu et al., 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [Longpre et al., 2023] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pages 22631–22648. PMLR, 2023.
- [Ma et al., 2023] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. Mega: Moving average equipped gated attention. In *The Eleventh International Conference on Learning Representations*, 2023.
- [Ma et al., 2024] Xuezhe Ma, Xiaomeng Yang, Wenhan Xiong, Beidi Chen, Lili Yu, Hao Zhang, Jonathan May, Luke Zettlemoyer, Omer Levy, and Chunting Zhou. Megalodon: Efficient llm pretraining and inference with unlimited context length. *arXiv preprint arXiv:2404.08801*, 2024.
- [Madaan et al., 2024] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Manning, 2022] Christopher D Manning. Human language understanding & reasoning. *Daedalus*, 151(2):127–138, 2022.
- [Marcus, 1993] Gary F Marcus. Negative evidence in language acquisition. *Cognition*, 46(1):53–85, 1993.
- [Martins et al., 2022] Pedro Henrique Martins, Zita Marinho, and André FT Martins. ∞ -former: Infinite memory transformer-former: Infinite memory transformer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5468–5485, 2022.
- [Mavi et al., 2024] Vaibhav Mavi, Anubhav Jangra, and Adam Jatowt. Multi-hop question answering. *Foundations and Trends® in Information Retrieval*, 17(5):457–586, 2024.
- [Michel et al., 2019] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.
- [Micikevicius et al., 2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *Proceedings of International Conference on Learning Representations*, 2018.
- [Miettinen, 1999] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 1999.
- [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.

- [Min et al., 2019] Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. Multi-hop reading comprehension through question decomposition and rescoring. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6097–6109, 2019.
- [Minaee et al., 2024] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.
- [Mishra et al., 2022] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, 2022.
- [Mnih et al., 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pages 1928–1937, 2016.
- [Mohtashami and Jaggi, 2024] Amirkeivan Mohtashami and Martin Jaggi. Random-access infinite context length for transformers. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Mu et al., 2024] Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Munkhdalai et al., 2024] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.
- [Nakano et al., 2021] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- [Narayanan et al., 2021] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.
- [Ng et al., 1999] Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.
- [Nvidia, 2025] Nvidia. Nvidia nim llms benchmarking. <https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html>, 2025. Retrieved 2025-03-17.
- [OpenAI, 2024] OpenAI. Learning to reason with llms, September 2024. URL <https://openai.com/index/learning-to-reason-with-llms/>.
- [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.
- [Pal et al., 2023] Koyena Pal, Jiuding Sun, Andrew Yuan, Byron C Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, pages 548–560, 2023.

- [Pan et al., 2022] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022.
- [Pan et al., 2024] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024.
- [Parisi et al., 2022] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- [Parisi et al., 2019] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.
- [Parmar et al., 2018] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pages 4055–4064. PMLR, 2018.
- [Patel et al., 2024] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.
- [Penedo et al., 2023] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refined-web dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.
- [Peng et al., 2024] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.
- [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.
- [Plackett, 1975] Robin L Plackett. The analysis of permutations. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 24(2):193–202, 1975.
- [Pope et al., 2023] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023.
- [Prasad et al., 2023] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3845–3864, 2023.
- [Press et al., 2022] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *Proceedings of International Conference on Learning Representations*, 2022.
- [Press et al., 2023] Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, 2023.
- [Pryzant et al., 2023] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng.

- Automatic prompt optimization with "gradient descent" and beam search. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.
- [Qiu et al., 2020] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020.
- [Radford et al., 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.
- [Radford et al., 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8), 2019.
- [Radford et al., 2021] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.
- [Rae et al., 2019] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2019.
- [Rafailov et al., 2024] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Raffel et al., 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [Ramachandran et al., 2017] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [Rolnick et al., 2019] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.
- [Rosenfeld et al., 2020] Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. In *Proceedings of International Conference on Learning Representations*, 2020.
- [Ruan et al., 2024] Junhao Ruan, Long Meng, Weiqiao Shan, Tong Xiao, and Jingbo Zhu. A survey of llm surveys. <https://github.com/NiuTrans/ABigSurveyOfLLMs>, 2024.
- [Rubin et al., 2022] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, 2022.
- [Russell, 2019] Stuart Russell. *Human Compatible: Artificial Intelligence and the Problem of Controls*. Viking, 2019.
- [Sanh et al., 2020] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.
- [Sanh et al., 2022] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *Proceedings of International Conference on Learning Representations*, 2022.

- [Schick et al., 2023] Timo Schick, Jane A. Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Schick et al., 2024] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.
- [Schmidhuber, 2015] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [Schulman et al., 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pages 1889–1897, 2015.
- [Schulman et al., 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [Sennrich et al., 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, 2016.
- [Seo et al., 2017] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *Proceedings of International Conference on Learning Representations*, 2017.
- [Shannon, 1951] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.
- [Shaw et al., 2018] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.
- [Shazeer, 2019] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.
- [Shazeer, 2020] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.
- [Shen et al., 2020] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020.
- [Shinn et al., 2023] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
- [Shoeybi et al., 2019] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [Skalse et al., 2022] Joar Skalse, Nikolaus Howe, Dmitrii Krashenninikov, and David Krueger. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.
- [Snell et al., 2022] Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. *arXiv preprint arXiv:2209.15189*, 2022.
- [Snell et al., 2024] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

- [Snell et al., 2025] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [Socher et al., 2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.
- [Song et al., 2019] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936. PMLR, 2019.
- [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.
- [Su et al., 2024] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Ro-former: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
- [Su et al., 2022] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. *Advances in Neural Information Processing Systems*, 35:21548–21561, 2022.
- [Sun et al., 2020] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 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.
- [Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.
- [Szepesvári, 2010] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.
- [Talmor and Berant, 2018] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. *arXiv preprint arXiv:1803.06643*, 2018.
- [Taori et al., 2023] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.
- [Tay et al., 2020] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020.
- [Team et al., 2024] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.
- [Teknium, 2023] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL <https://huggingface.co/datasets/teknium/OpenHermes-2.5>.
- [Timonin et al., 2022] Denis Timonin, BoYang Hsueh, and Vinh Nguyen. Accelerated inference for large transformer models using nvidia triton inference server. <https://developer.nvidia.com/blog/accelerated-inference-for-large-transformer-models-using-nvidia-fastertransformer-and-nvidia-triton-inference-server/>, 2022.
- [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.
- [Uesato et al., 2022] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.
- [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.
- [Viterbi, 1967] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 1967.
- [Von Oswald et al., 2023] Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *Proceedings of International Conference on Machine Learning*, pages 35151–35174. PMLR, 2023.
- [Wang et al., 2024] Chenglong Wang, Hang Zhou, Yimin Hu, Yifu Huo, Bei Li, Tongran Liu, Tong Xiao, and Jingbo Zhu. Esrl: Efficient sampling-based reinforcement learning for sequence generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19107–19115, 2024.
- [Wang et al., 2023] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023a.
- [Wang et al., 2019] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, 2019.
- [Wang et al., 2022] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747*, 2022a.
- [Wang et al., 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023b.
- [Wang et al., 2022] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Es-haan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krma Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkrit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujana Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, 2022b.

- [Wang et al., 2023] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources. *Advances in Neural Information Processing Systems*, 36:74764–74786, 2023c.
- [Wang et al., 2023] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, 2023d.
- [Wang et al., 2023] Zhenyi Wang, Enneng Yang, Li Shen, and Heng Huang. A comprehensive survey of forgetting in deep learning beyond continual learning. *arXiv preprint arXiv:2307.09218*, 2023e.
- [Warstadt et al., 2019] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.
- [Wei et al., 2022] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *Proceedings of International Conference on Learning Representations*, 2022a.
- [Wei et al., 2022] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022b.
- [Wei et al., 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022c.
- [Welleck et al., 2023] Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023.
- [Weng, 2021] Lilian Weng. How to train really large models on many gpus? *lilianweng.github.io*, Sep 2021. URL <https://lilianweng.github.io/posts/2021-09-25-train-large/>.
- [Wiener, 1960] Norbert Wiener. Some moral and technical consequences of automation: As machines learn they may develop unforeseen strategies at rates that baffle their programmers. *Science*, 131(3410): 1355–1358, 1960.
- [Williams et al., 2018] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, 2018.
- [Williams, 1992] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.
- [Wingate et al., 2022] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. Prompt compression and contrastive conditioning for controllability and toxicity reduction in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 5621–5634, 2022.
- [Wozengraft and Reiffen, 1961] John M. Wozengraft and Barney Reiffen. *Sequential Decoding*. The MIT Press, 1961.
- [Wu et al., 2023] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023a.
- [Wu et al., 2024] Wilson Wu, John X Morris, and Lionel Levine. Do language models plan for future tokens? *arXiv preprint arXiv:2404.00859*, 2024.

- [Wu et al., 2021] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *Proceedings of International Conference on Learning Representations*, 2021.
- [Wu et al., 2023] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b.
- [Xia et al., 2024] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.
- [Xiao et al., 2024] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *Proceedings of The Twelfth International Conference on Learning Representations*, 2024.
- [Xiao and Zhu, 2023] Tong Xiao and Jingbo Zhu. Introduction to transformers: an nlp perspective. *arXiv preprint arXiv:2311.17633*, 2023.
- [Xiao et al., 2013] Tong Xiao, Jingbo Zhu, and Tongran Liu. Bagging and boosting statistical machine translation systems. *Artificial Intelligence*, 195:496–527, 2013.
- [Xiao et al., 2019] Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. Sharing attention weights for fast transformer. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*, pages 5292–5298, 2019.
- [Xie et al., 2022] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In *Proceedings of International Conference on Learning Representations*, 2022.
- [Xin et al., 2020] Ji Xin, Raphael Tang, Jaesun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, 2020.
- [Xu et al., 2024] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.
- [Yang et al., 2024] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [Yang et al., 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.
- [Yao et al., 2024] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
- [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.
- [Yu et al., 2022] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.
- [Yu et al., 2023] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. Towards better chain-of-thought prompting strategies: A survey. *arXiv preprint arXiv:2310.04959*, 2023.
- [Zaheer et al., 2020] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, C. Albeti, S. Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, L. Yang, and A. Ahmed. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

- [Zellers et al., 2018] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 93–104, 2018.
- [Zhang and Sennrich, 2019] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.
- [Zhang et al., 2024] Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moon-tae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. In *ACL (Findings)*, 2024.
- [Zhang et al., 2023] Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and Hai Zhao. Igniting language intelligence: The hitchhiker’s guide from chain-of-thought reasoning to language agents. *arXiv preprint arXiv:2311.11797*, 2023a.
- [Zhang et al., 2023] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*, 2023b.
- [Zhao et al., 2024] Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. *arXiv preprint arXiv:2402.04833*, 2024.
- [Zhao et al., 2023] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie, and Ji rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.
- [Zhong et al., 2024] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.
- [Zhou et al., 2023] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinu Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023a.
- [Zhou et al., 2023] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023b.
- [Zhou et al., 2020] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33:18330–18341, 2020.
- [Zhou et al., 2023] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023c.
- [Zoph and Le, 2016] Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *Proceedings of International Conference on Learning Representations*, 2016.
- [Zoph et al., 2020] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33:3833–3845, 2020.

Index

- k*-NN, [76](#)
- k*-NN LM, [76](#)
- k*-NN language modeling, [76](#)
- k*-nearest neighbors, [76](#)

- A2C, [178](#)
- action-value function, [174](#)
- advantage, [178](#)
- advantage actor-critic, [178](#)
- Agent, [47](#)
- ALiBi, [85](#)
- alignment, [46](#)
- attention with linear biases, [85](#)
- automated machine learning, [139](#)
- automatic prompt design, [139](#)
- AutoML, [139](#)
- autonomous agents, [137](#)

- BART, [19](#)
- BERT, [1](#)
- Best-of-*N* sampling, [200](#)
- BoN sampling, [200](#)
- Bradley-Terry model, [181](#)

- calculation annotation, [115](#)
- catastrophic forgetting, [35](#)
- causal language modeling, [9](#)
- chain of thought, [115](#)
- chain-of-thought prompting, [53](#)
- completion, [6](#)
- compositional generalization, [124](#)
- Continuous batching, [226](#)
- CoT, [115](#)
- COT prompting, [53](#)
- cross-lingual language models, [28](#)
- cumulative reward, [175](#)

- Decoding, [209](#)
- deliberate-then-generate, [128](#)
- demonstrations, [6](#)
- direct preference optimization, [193](#)
- Document Rotation, [20](#)
- DPO, [193](#)
- DTG, [128](#)

- emergent abilities, [64](#)
- external memories, [74](#)

- Extrapolation, [82](#)

- few-shot COT prompting, [54](#)

- gated linear unit, [59](#)
- gaussian error linear unit, [59](#)
- GeLU, [59](#)
- GLU, [59](#)
- GPT, [1](#)
- GQA, [80](#)
- Grouped query attention, [80](#)

- hard prompts, [142](#)
- human preference alignment, [155](#)

- ICL, [53](#)
- ICT, [6](#)
- importance sampling, [183](#)
- in-context learning, [6](#), [53](#), [96](#)
- Inference Engine, [225](#)
- inference-time scaling, [234](#)
- input inversion, [166](#)
- instruction alignment, [155](#)
- instruction fine-tuning, [43](#), [157](#)
- interference, [29](#)
- internal memories, [74](#)
- Interpolation, [82](#)
- irreducible error, [65](#)
- iteration-based scheduling, [226](#)

- key-value cache, [68](#), [207](#)
- KV cache, [68](#), [207](#)

- label mapping, [106](#)
- Learning from Human Feedback, [47](#)
- least-to-most prompting, [121](#)
- long-context LLMs, [66](#)

- masked language modeling, [1](#), [9](#)
- mBERT, [28](#)
- memory-based methods, [74](#)
- MQA, [80](#)
- multi-lingual BERT, [28](#)
- multi-query attention, [80](#)

- NAS, [139](#)
- neural architecture search, [139](#)
- next sentence prediction, [12](#)

- NSP, 12
- nucleus sampling, 216
- offline reinforcement learning, 196
- one-shot COT prompting, 54
- ORMs, 243
- outcome reward models, 243
- Outcome-based Approaches, 198
- overoptimization problem, 192
- parallel scaling, 240
- Performance Estimation, 139
- performance function, 176
- performance gap recovered, 170
- permuted language modeling, 11
- PGR, 170
- Plackett-Luce model, 187
- PPO, 51, 184
- Prefilling, 207
- prefix fine-tuning, 146
- prefix language modeling, 16
- PRM, 243
- problem decomposition, 117
- process reward model, 243
- Process-based Approaches, 198
- prompt embeddings, 150
- prompt engineering, 96
- prompt optimization, 139
- Prompt Search Space, 139
- prompting engineering, 51
- proximal policy optimization, 51, 184
- Q-value function, 175
- RAG, 77
- ratio function, 183
- rectified linear unit, 59
- reinforcement learning from human feedback, 47, 156
- rejection sampling, 201
- relation extraction, 109
- ReLU, 59
- request-level scheduling, 225
- retrieval-augmented generation, 77
- return, 175
- reward gaming, 192
- reward hacking, 192
- Reward Model, 48
- RLHF, 47, 156
- RoBERTa, 27
- sample efficient, 167
- scaling laws, 63
- Scheduler, 225
- self-consistency, 132
- self-instruct, 163
- self-supervised learning, 3
- self-training, 3
- Sentence Reordering, 20
- Sequence Encoding Models, 3
- Sequence Generation Models, 4
- sequential scaling, 240
- SFT, 47, 155
- single-round prediction, 158
- soft prompts, 142
- Span Masking, 19
- Speculative decoding, 218
- speculative execution, 218
- state-value function, 174
- Strong Ceiling Performance, 170
- Sub-problem Generation, 120
- Sub-problem Solving, 120
- superficial alignment hypothesis, 167
- Supervised Fine-tuning, 47
- supervised fine-tuning, 155
- supervised learning, 2
- surrogate objective, 183
- T5, 15
- TD, 179
- temporal difference, 179
- text completion, 110
- text transformation, 110
- Token Deletion, 19
- Token Masking, 19
- Transformers, 1
- translation language modeling, 29
- trust regions, 184
- unsupervised learning, 2
- Weak Performance, 170
- weak-to-strong generalization, 169
- Weak-to-strong Performance, 170
- XLMS, 28

zero-shot COT, [54](#)

zero-shot learning, [45](#)