Foundations of Large Language Models

Tong Xiao and Jingbo Zhu

January 17, 2025

NLP Lab, Northeastern University & NiuTrans Research

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 four 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. This chapter focuses on instruction fine-tuning and alignment based on human feedback.

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.

In writing this book, we have gradually realized that it is more like a compilation of "notes" we have taken while learning about large language models. Through this note-taking writing style, we hope to offer readers a flexible learning path. Whether they wish to dive deep into a specific area or gain a comprehensive understanding of large language models, they will find the knowledge and insights they need within these "notes".

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, Xinyu Liu, Huiwen Bao, Tong Zheng, Junhao Ruan, and Qing Yang.

Notation

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

Contents

CHAPTER 1

Pre-training

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

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

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

1.1 Pre-training NLP Models

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation. While these problems have different forms, for simplicity, we describe them using a single model defined as follows:

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

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

There are two fundamental issues here.

- Optimizing θ on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.

- Applying the pre-trained model $g_{\hat{\theta}}(\cdot)$ to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters $\hat{\theta}$ slightly using labeled data or prompt the model with task descriptions.

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

1.1.1 Unsupervised, Supervised and Self-supervised Pre-training

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

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

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

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

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

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

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

1.1.2 Adapting Pre-trained Models

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

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

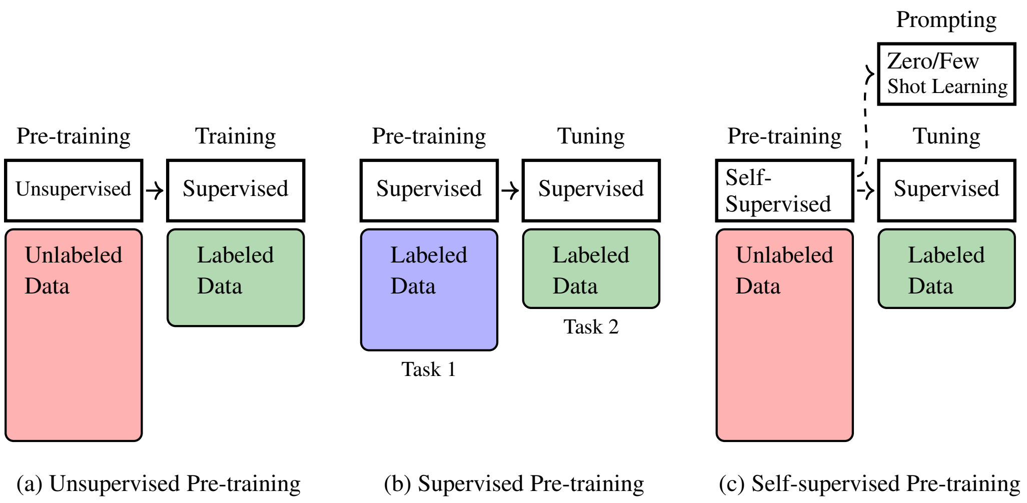


Fig. 1.1: Illustration of unsupervised, supervised, and self-supervised pre-training. In unsupervised pre-training, the pre-training is performed on large-scale unlabeled data. It can be viewed as a preliminary step to have a good starting point for the subsequent optimization process, though considerable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

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

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

1.1.2.1 Fine-tuning of Pre-trained Models

For sequence encoding pre-training, a common method of adapting pre-trained models is fine-tuning. Let $\mathrm{Encode}_\theta(\cdot)$ denote an encoder with parameters θ, for example, $\mathrm{Encode}_\theta(\cdot)$ 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

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

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

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

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

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

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

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

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

I love the food here. It's amazing!

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

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

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 adaption of pre-trained models very efficient in practice: given a pre-trained model and a downstream task, we just need to collect some labeled data, and slightly adjust the model parameters on this data. 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

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

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

Given a sequence of m tokens $\{x_0, ..., x_m\}$, the loss on this sequence is the sum of the loss over the positions $\{0, ..., m-1\}$, given by

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

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

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

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

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 $\mathbf{x} = x_0...x_m$ and produces a sequence of vectors $\mathbf{H} = \mathbf{h}_0...\mathbf{h}_m$[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

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

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

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

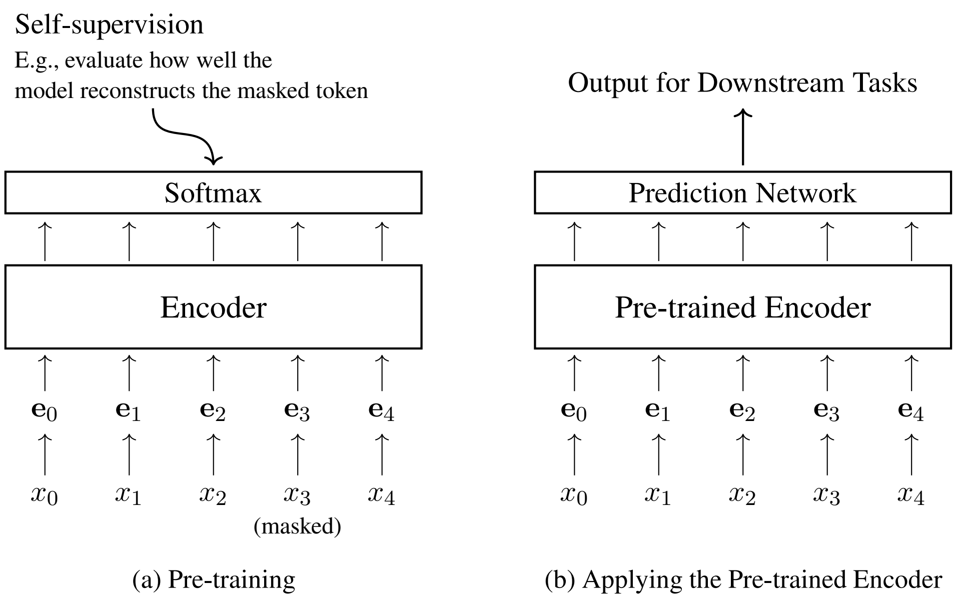

Self-supervision
E.g., evaluate how well the model reconstructs the masked token
Softmax
Encoder
$\mathbf{e}_0$ $\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{e}_3$ $\mathbf{e}_4$
x_0 x_1 x_2 x_3 x_4
(masked)
(a) Pre-training
Output for Downstream Tasks
Prediction Network
Pre-trained Encoder
$\mathbf{e}_0$ $\mathbf{e}_1$ $\mathbf{e}_2$ $\mathbf{e}_3$ $\mathbf{e}_4$
x_0 x_1 x_2 x_3 x_4
(b) Applying the Pre-trained Encoder

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 $\mathrm{Softmax}_{\mathbf{W}}(\cdot)$ to denote that the Softmax layer is parameterized by $\mathbf{W}$, that is, $\mathrm{Softmax}_{\mathbf{W}}(\mathbf{H}) = \mathrm{Softmax}(\mathbf{H} \cdot \mathbf{W})$. 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 $\mathbf{x} = x_0...x_m$, suppose that we mask the tokens at positions $\mathcal{A}(\mathbf{x}) = \{i_1, ..., i_u\}$. 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

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

or alternatively express it using the cross-entropy loss

$$(\widehat{\mathbf{W}}, \hat{\theta}) = \underset{\mathbf{W},\theta}{\arg\min} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i\in\mathcal{A}(\mathbf{x})} \mathrm{LogCrossEntropy}(\mathbf{p}_i^{\mathbf{W},\theta}, \mathbf{p}_i^{\mathrm{gold}}) \tag{1.11}$$

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

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

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 $\mathrm{Encoder}_{\hat{\theta}}(\cdot)$ 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 $x_0 \to x_1 \to x_2 \to x_3 \to x_4$. The probability of the sequence can be modeled via a generation process.

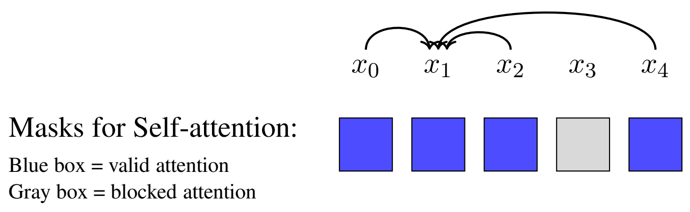
$$\begin{aligned}\Pr(\mathbf{x}) &= \Pr(x_0)\cdot\Pr(x_1|x_0)\cdot\Pr(x_2|x_0,x_1)\cdot\Pr(x_3|x_0,x_1,x_2)\cdot \\ &\quad \Pr(x_4|x_0,x_1,x_2,x_3) \\ &= \Pr(x_0)\cdot\Pr(x_1|\mathbf{e}_0)\cdot\Pr(x_2|\mathbf{e}_0,\mathbf{e}_1)\cdot\Pr(x_3|\mathbf{e}_0,\mathbf{e}_1,\mathbf{e}_2)\cdot \\ &\quad \Pr(x_4|\mathbf{e}_0,\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3)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

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

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

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


x_0
x_1
x_2
x_3
x_4
Masks for Self-attention:
Blue box = valid attention
Gray box = blocked attention

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

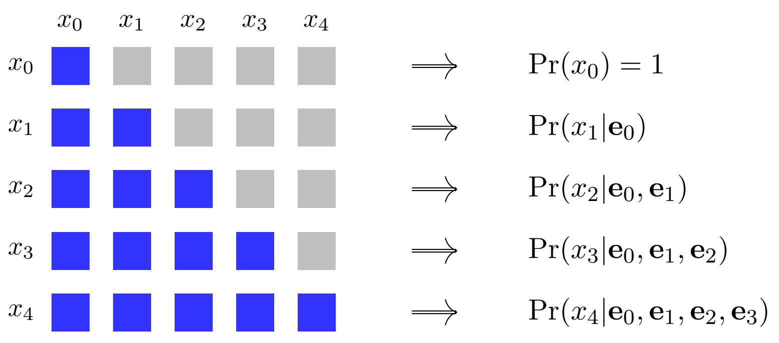
Another commonly-used idea to train an encoder is to consider classification tasks. In self-supervised learning, this is typically done by creating new classification challenges from the unlabeled text. There are many different ways to design the classification tasks. Here we present two popular tasks.

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 $\text{Sent}_A =$ *'It is raining .'* and $\text{Sent}_B =$ *'I need an umbrella .'*. The input sequence of the encoder could be

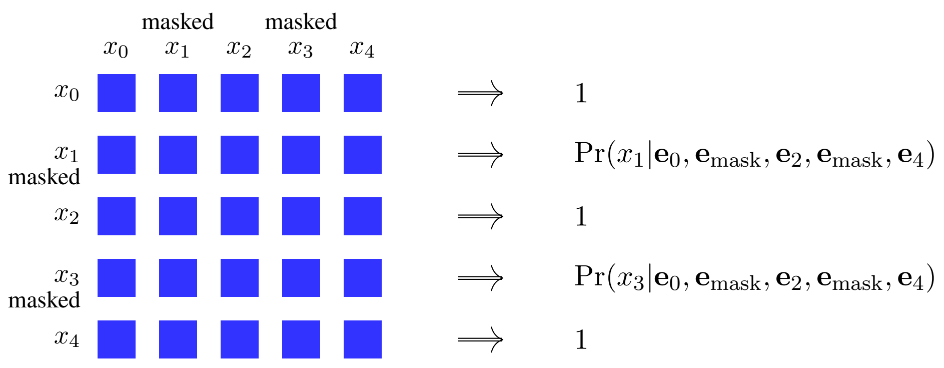
$$\text{[CLS] It is raining . [SEP] I need an umbrella . [SEP]}$$

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 $\mathbf{e}_i$, and then feed the embedding sequence $\{\mathbf{e}_0, ..., \mathbf{e}_m\}$ into the encoder to obtain the output sequence $\{\mathbf{h}_0, ..., \mathbf{h}_m\}$. Since $\mathbf{h}_0$ 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

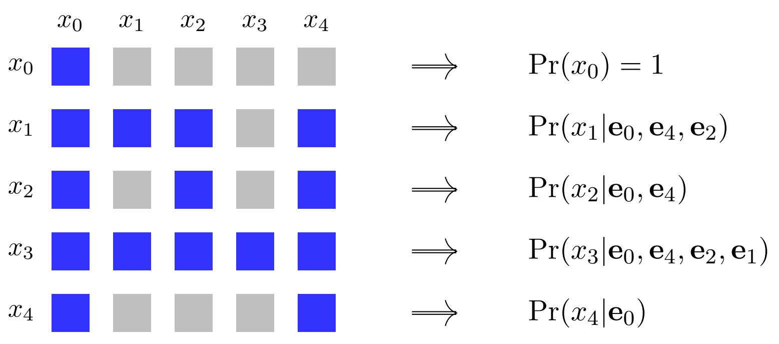
$$\begin{array}{rcccccccccccc}
\text{token:} & \color{red}{\text{[CLS]}} & \text{It} & \text{is} & \text{raining} & . & \text{[SEP]} & \text{I} & \text{need} & \text{an} & \text{umbrella} & . & \text{[SEP]} \\
\text{embedding:} & \color{red}{\mathbf{e}_0} & \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_4 & \mathbf{e}_5 & \mathbf{e}_6 & \mathbf{e}_7 & \mathbf{e}_8 & \mathbf{e}_9 & \mathbf{e}_{10} & \mathbf{e}_{11} \\
 & \color{red}{\downarrow} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\cline{2-13}
 & \multicolumn{12}{|c|}{\text{Encoder}} \\
\cline{2-13}
 & \color{red}{\downarrow} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\text{encoding:} & \color{red}{\mathbf{h}_0} & \mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3 & \mathbf{h}_4 & \mathbf{h}_5 & \mathbf{h}_6 & \mathbf{h}_7 & \mathbf{h}_8 & \mathbf{h}_9 & \mathbf{h}_{10} & \mathbf{h}_{11} \\
 & \downarrow & & & & & & & & & & & \\
 & \text{Softmax} & & & & & & & & & & & \\
 & \downarrow & & & & & & & & & & & \\
 & \text{Is Next or Not?} & & & & & & & & & & &
\end{array}$$


x_0 x_1 x_2 x_3 x_4
x_0 $\Longrightarrow$ $\Pr(x_0) = 1$
x_1 $\Longrightarrow$ $\Pr(x_1|\mathbf{e}_0)$
x_2 $\Longrightarrow$ $\Pr(x_2|\mathbf{e}_0, \mathbf{e}_1)$
x_3 $\Longrightarrow$ $\Pr(x_3|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2)$
x_4 $\Longrightarrow$ $\Pr(x_4|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$

(a) Causal Language Modeling (order: $x_0 \to x_1 \to x_2 \to x_3 \to x_4$)


masked
masked
x_0 x_1 x_2 x_3 x_4
x_0 $\Longrightarrow$ 1
x_1 masked $\Longrightarrow$ $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_{\text{mask}}, \mathbf{e}_2, \mathbf{e}_{\text{mask}}, \mathbf{e}_4)$
x_2 $\Longrightarrow$ 1
x_3 masked $\Longrightarrow$ $\Pr(x_3|\mathbf{e}_0, \mathbf{e}_{\text{mask}}, \mathbf{e}_2, \mathbf{e}_{\text{mask}}, \mathbf{e}_4)$
x_4 $\Longrightarrow$ 1

(b) Masked Language Modeling (order: $x_0, \text{[MASK]}, x_2, \text{[MASK]}, x_4 \to x_1, x_3$)

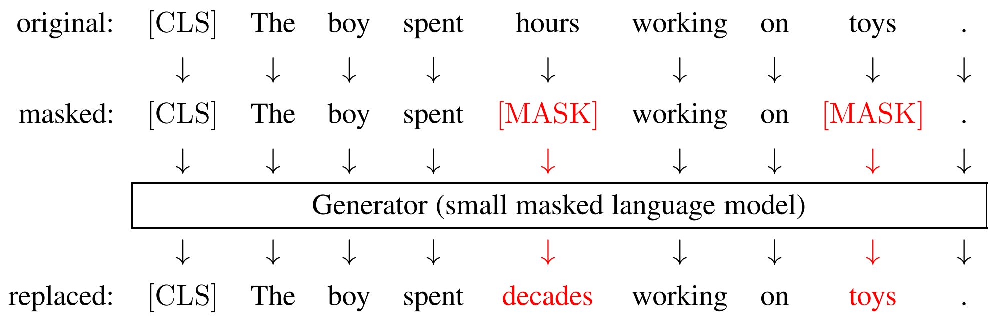

x_0 x_1 x_2 x_3 x_4
x_0 $\Longrightarrow$ $\Pr(x_0) = 1$
x_1 $\Longrightarrow$ $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2)$
x_2 $\Longrightarrow$ $\Pr(x_2|\mathbf{e}_0, \mathbf{e}_4)$
x_3 $\Longrightarrow$ $\Pr(x_3|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2, \mathbf{e}_1)$
x_4 $\Longrightarrow$ $\Pr(x_4|\mathbf{e}_0)$

(c) Permuted Language Modeling (order: $x_0 \to x_4 \to x_2 \to x_1 \to x_3$)

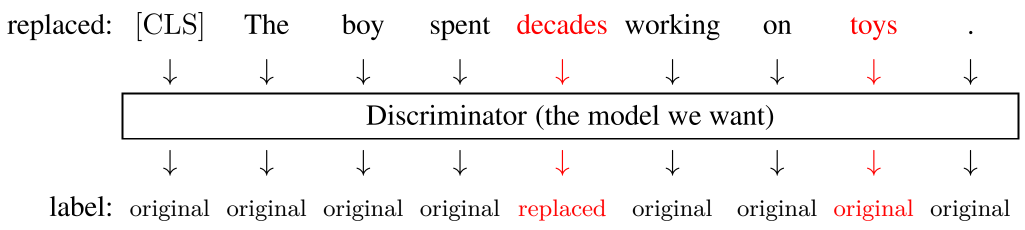
Fig. 1.3: Comparison of self-attention masking results of causal language modeling, masked language modeling and permuted language modeling. The gray cell denotes the token at position j does not attend to the token at position i. The blue cell (i, j) denotes that the token at position j attends to the token at position i. $\mathbf{e}_{\text{mask}}$ 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.


original: [CLS] The boy spent hours working on toys .
masked: [CLS] The boy spent [MASK] working on [MASK] .
Generator (small masked language model)
replaced: [CLS] The boy spent decades working on toys .

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


replaced: [CLS] The boy spent decades working on toys .
Discriminator (the model we want)
label: original original original original replaced original original original original

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

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

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

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

where [CLS] and $\langle s \rangle$ are the start symbols on the source and target sides, respectively[5].

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

[CLS] Answer: when was Albert Einstein born?
$\rightarrow$ $\langle s \rangle$ He was born on March 14, 1879.

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

[CLS] Score the translation from English to Chinese. English: when in Rome, do as the Romans do. Chinese: 人 在 罗马 就 像 罗马 人 一样 做事 。
$\rightarrow$ $\langle s \rangle$ 0.81

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

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

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

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

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

$$\begin{array}{rl} & \text{[CLS] The puppies are [MASK] outside [MASK] house .} \\ \rightarrow & \langle s \rangle \ \_\_ \ \_\_ \ \_\_ \ \text{frolicking} \ \_\_ \ \text{the} \ \_\_ \ \_\_ \end{array}$$

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

$$\begin{array}{rl} & \text{[CLS] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK]} \\ \rightarrow & \langle s \rangle \ \text{The puppies are frolicking outside the house .} \end{array}$$

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.

$$\begin{array}{rl} & \text{[CLS] The puppies are [MASK] outside [MASK] house .} \\ \rightarrow & \langle s \rangle \ \text{The puppies are frolicking outside the house .} \end{array}$$

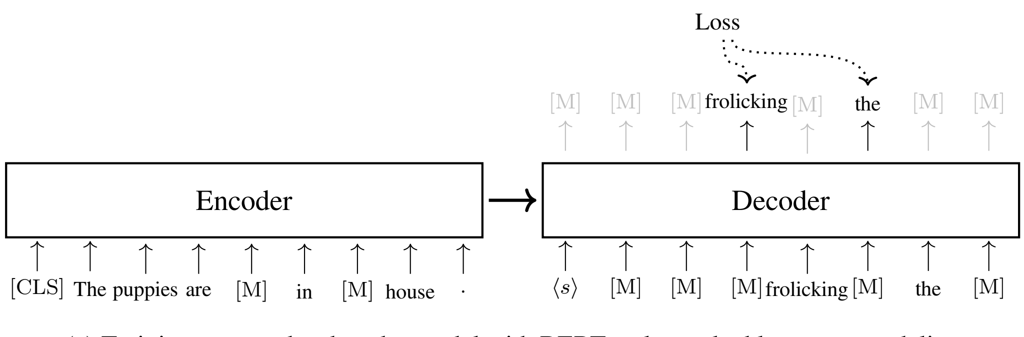
By learning to map from this corrupted sequence to its uncorrupted counterpart, the model gains the ability to understand on the encoder side and to generate on the decoder side. See Figure 1.4 for an illustration of how an encoder-decoder model is trained with BERT-style and denoising autoencoding objectives.

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

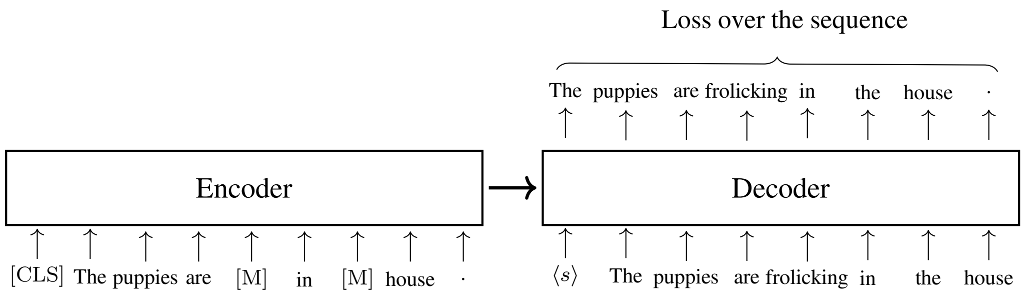
$$\begin{array}{rl} & \text{[CLS] The puppies are [MASK] outside [MASK] [MASK] .} \\ \rightarrow & \langle s \rangle \ \text{The puppies are frolicking outside the house .} \end{array}$$

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

$$\begin{array}{rl} & \text{[CLS] The puppies are [X] outside [Y] .} \\ \rightarrow & \langle s \rangle \ \text{[X] frolicking [Y] the house [Z]} \end{array}$$


Loss
[M]
frolicking
the
Encoder
Decoder
[CLS] The puppies are [M] in [M] house .
⟨s⟩ [M] [M] [M] frolicking [M] the [M]

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


Loss over the sequence
The puppies are frolicking in the house .
Encoder
Decoder
[CLS] The puppies are [M] in [M] house .
⟨s⟩ The puppies are frolicking in the house

(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 $\mathbf{y}$

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

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 $\mathbf{x}_{\mathrm{noise}}$. By feeding $\mathbf{x}_{\mathrm{noise}}$ into the encoder, we wish the decoder to output the original input. The training objective can be defined as

$$(\hat{\theta}, \hat{\omega}) = \underset{\theta,\omega}{\arg\max}\, \mathrm{Loss}(\mathrm{Model}_{\theta,\omega}(\mathbf{x}_{\mathrm{noise}}), \mathbf{x}) \qquad (1.16)$$

Here the loss function $\mathrm{Loss}(\mathrm{Model}_{\theta,\omega}(\mathbf{x}_{\mathrm{noise}}), \mathbf{x})$ evaluates how well the model $\mathrm{Model}_{\theta,\omega}(\mathbf{x}_{\mathrm{noise}})$ 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 $\underline{\text{frolicking}}$ outside $\underline{\text{the}}$ house .
Token Masking ($\mathbf{x}_{\mathrm{noise}}$):	The puppies are [MASK] outside [MASK] house .
Token Deletion ($\mathbf{x}_{\mathrm{noise}}$):	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 $\underline{0}$ puppies are $\underline{\text{frolicking outside the}}$ house .
Span Masking ($\mathbf{x}_{\mathrm{noise}}$):	The [MASK] puppies are [MASK] house .

  Here the span *frolicking outside the* is replaced with a single [MASK]. $\underline{0}$ 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.

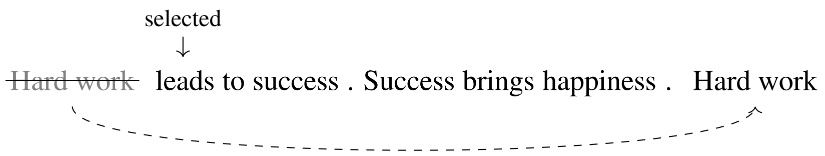
- **Sentence Reordering**. This method randomly permutes the sentences so that the model can learn to reorder sentences in a document. Consider, for example, two consecutive sentences

  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


selected
~~Hard work~~ leads to success . Success brings happiness . Hard work

  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.

$$\text{Loss}_{\text{BERT}} = \text{Loss}_{\text{MLM}} + \text{Loss}_{\text{NSP}} \tag{1.17}$$

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 $\mathrm{Loss}_{\mathrm{BERT}}$ 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

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

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

Given this sequence, we can obtain LossMLM and LossNSP 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 $\underline{\text{raining}}$. [SEP] $\underline{\text{I}}$ need $\underline{\text{an}}$ $\underline{\text{umbrella}}$. [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 $\underline{\text{raining}}$. [SEP] $\underline{\text{I}}$ need $\underline{\text{an}}$ $\underline{\text{umbrella}}$. [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 $\underline{\text{raining}}$. [SEP] $\underline{\text{I}}$ need $\underline{\text{an}}$ $\underline{\text{umbrella}}$. [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

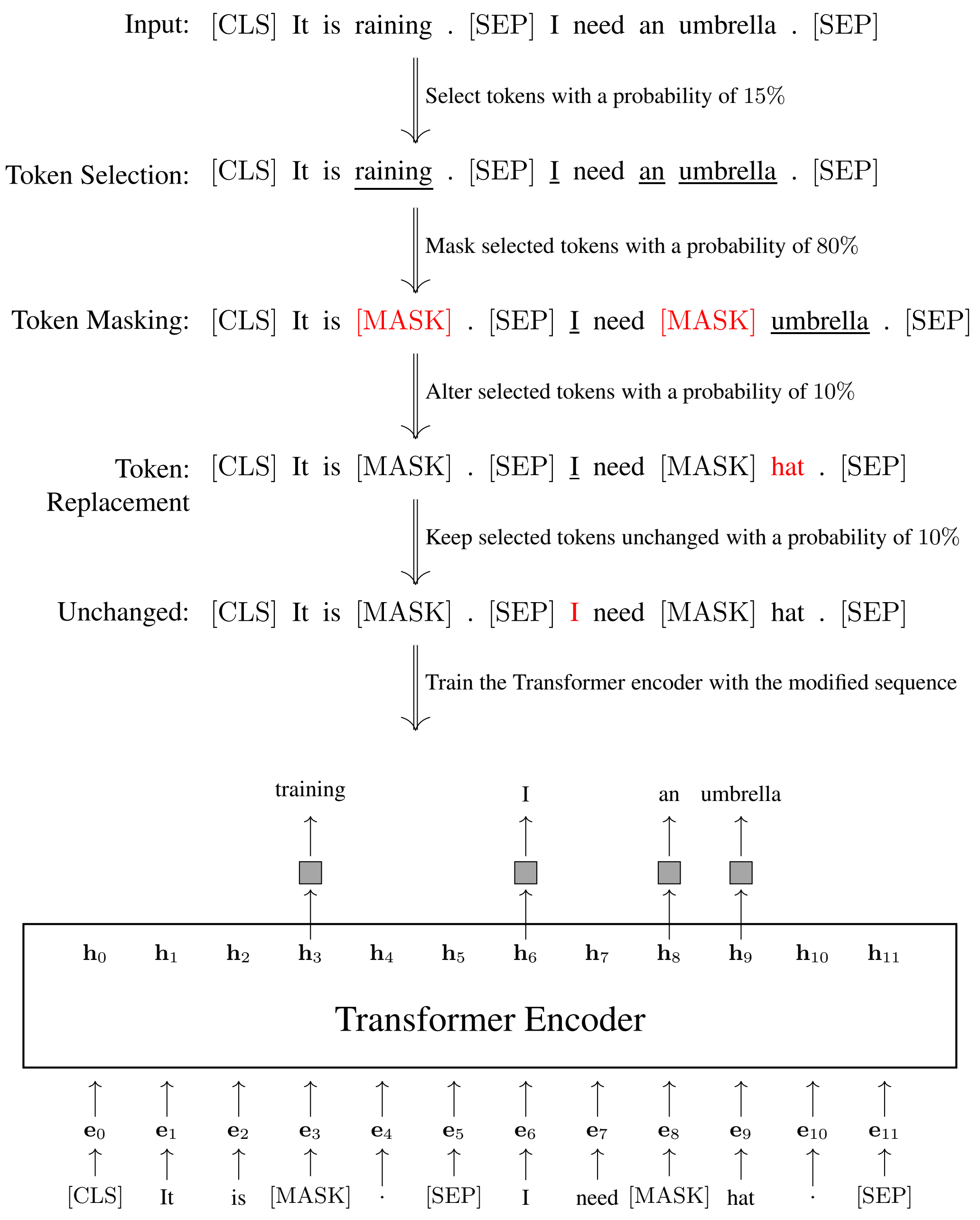
$$\text{Loss}_{\text{MLM}} = -\sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i(x_i|\bar{\mathbf{x}}) \tag{1.18}$$

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



Input: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
Select tokens with a probability of 15%
Token Selection: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
Mask selected tokens with a probability of 80%
Token Masking: [CLS] It is [MASK] . [SEP] I need [MASK] umbrella . [SEP]
Alter selected tokens with a probability of 10%
Token: Replacement [CLS] It is [MASK] . [SEP] I need [MASK] hat . [SEP]
Keep selected tokens unchanged with a probability of 10%
Unchanged: [CLS] It is [MASK] . [SEP] I need [MASK] hat . [SEP]
Train the Transformer encoder with the modified sequence
training
I
an
umbrella
Transformer Encoder

Fig. 1.5: A running example of BERT-style masked language modeling. First, 15% 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. $\mathbf{e}_i$ 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 $\mathbf{h}_{\text{cls}}$ (or $\mathbf{h}_0$). A classifier is built on top of $\mathbf{h}_{\text{cls}}$. Then, we can compute the probability of a label c given $\mathbf{h}_{\text{cls}}$, i.e., $\Pr(c|\mathbf{h}_{\text{cls}})$. There are many loss functions one can choose for classification problems. For example, in maximum likelihood training, we can define Loss_{NSP} as

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

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.

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

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

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

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

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

- **Vocabulary Size** ($|V|$). In Transformers, each input token is represented as an entry in a vocabulary V. Large vocabularies can cover more surface form variants of words, but may lead to increased storage requirements.
- **Embedding Size** (d_e). Every token is represented as a d_e-dimensional real-valued vector. As presented above, this vector is the sum of the token embedding, positional embedding, and segment embedding, all of which are also d_e-dimensional real-valued vectors.
- **Hidden Size** (d). The input and output of a sub-layer are of d dimensions. Besides, most of the hidden states of a sub-layer are d-dimensional vectors. In general, d can be roughly viewed as the width of the network.
- **Number of Heads** (n_{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 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.

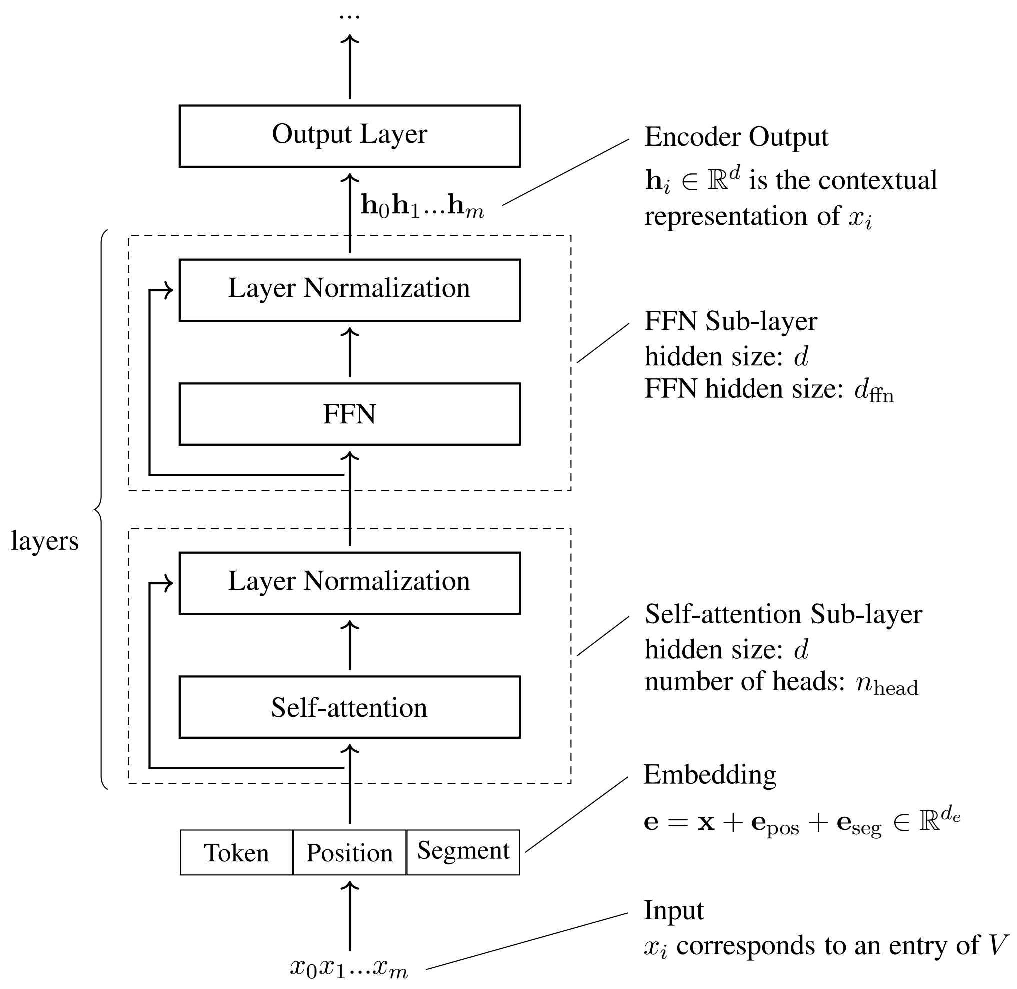

...
Output Layer
$\mathbf{h}_0\mathbf{h}_1...\mathbf{h}_m$
Encoder Output
$\mathbf{h}_i \in \mathbb{R}^d$ is the contextual representation of x_i
Layer Normalization
FFN
FFN Sub-layer
hidden size: d
FFN hidden size: d_{ffn}
layers
Layer Normalization
Self-attention
Self-attention Sub-layer
hidden size: d
number of heads: n_{head}
Embedding
$\mathbf{e} = \mathbf{x} + \mathbf{e}_{\text{pos}} + \mathbf{e}_{\text{seg}} \in \mathbb{R}^{d_e}$
Token
Position
Segment
Input
x_i corresponds to an entry of V
$x_0x_1...x_m$

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$_{\text{base}}$: $d = 768$, $L = 12$, $n_{\text{head}} = 12$, total number of parameters $=$ 110M.
- BERT$_{\text{large}}$: $d = 1,024$, $L = 14$, $n_{\text{head}} = 16$, total number of parameters $=$ 340M.

Training BERT models follows the standard training process of Transformers. Training larger models such as BERT$_{\text{large}}$ 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, Shoeybi et al. [2019] successfully trained a 3.9 billion-parameter BERT-like model, where hundreds of GPUs were used to manage the increased computational demands.

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 languages [6]. 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

$$\text{鲸鱼 是 哺乳 动物 。} \leftrightarrow \text{Whales are mammals .}$$

We can pack them to obtain a sequence, like this

$$\text{[CLS] 鲸鱼 是 哺乳 动物 。 [SEP] Whales are mammals . [SEP]}$$

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

$$\text{[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]}$$

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.

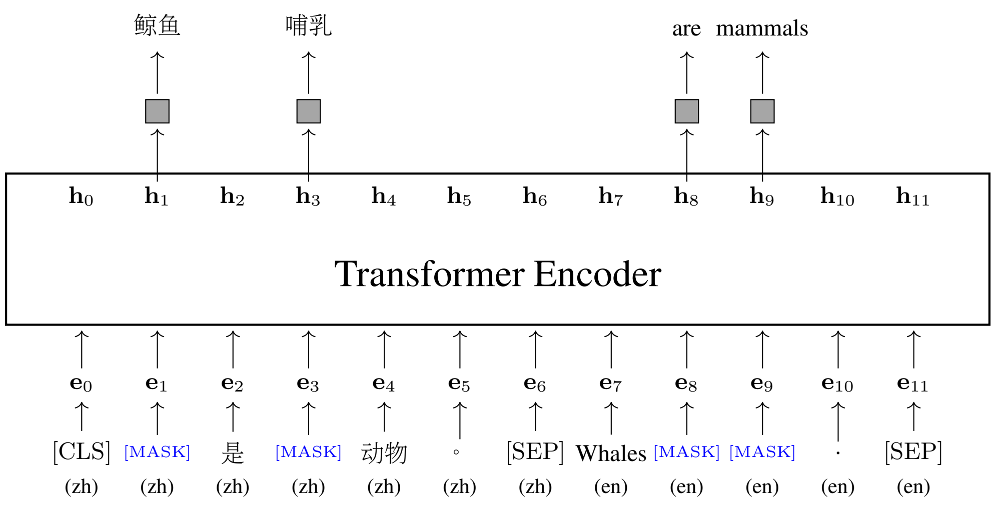
A benefit of multi-lingual pre-trained models is their inherent capability of handling code-switching. In NLP and linguistics, code-switching refers to switching among languages in a text. For example, the following is a mixed language text containing both Chinese and English:

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,



鲸鱼
哺乳
are
mammals
h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11
Transformer Encoder
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11
[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]
(zh) (zh) (zh) (zh) (zh) (zh) (zh) (en) (en) (en) (en) (en)

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., $\mathbf{e}_i$) 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 $\mathrm{BERT}_{\hat{\theta}}(\cdot)$ be a BERT model with pre-trained parameters $\hat{\theta}$, and $\mathrm{Predict}_{\omega}(\cdot)$ 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

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

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.

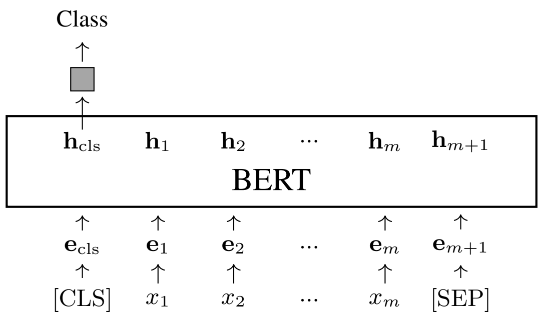
Then, we collect a set of labeled samples $\mathcal{D}$, and fine-tune the model by

$$(\tilde{\omega}, \tilde{\theta}) = \underset{\omega, \hat{\theta}+}{\arg\min} \sum_{(\mathbf{x}, \mathbf{y}_{\mathrm{gold}}) \in \mathcal{D}} \mathrm{Loss}(\mathbf{y}_{\omega, \hat{\theta}+}, \mathbf{y}_{\mathrm{gold}}) \tag{1.22}$$

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 $\hat{\theta}^{+}$.

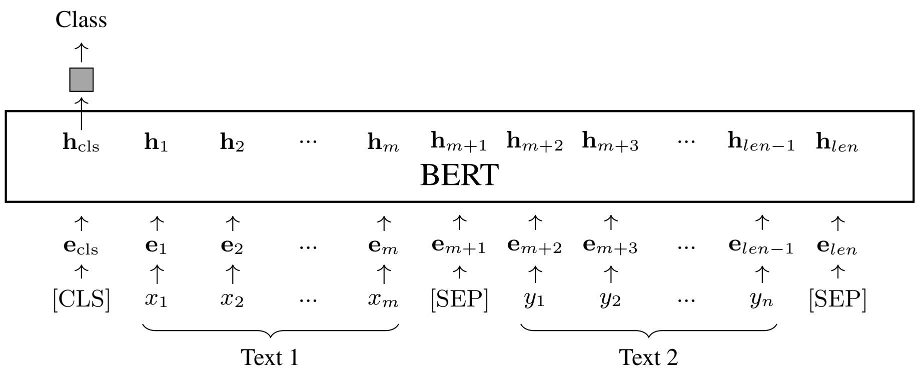
With the fine-tuned parameters $\tilde{\omega}$ and $\tilde{\theta}$, we can apply the model $\text{Predict}_{\tilde{\omega}}(\text{BERT}_{\tilde{\theta}}(\cdot))$ to new data of the same tasks for which the model was fine-tuned. The form of the downstream tasks determines the input and output formats 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 $\mathbf{h}_{\text{cls}}$ (or $\mathbf{h}_0$) is typically used as the representation of the entire text. The prediction network takes $\mathbf{h}_{\text{cls}}$ as input to produce a distribution of labels. Let $[\text{CLS}]x_1x_2...x_m$ be an input text. See below for an illustration of BERT-based text classification.


Class
$\mathbf{h}_{\text{cls}}$ $\mathbf{h}_1$ $\mathbf{h}_2$ ... $\mathbf{h}_m$ $\mathbf{h}_{m+1}$
BERT
$\mathbf{e}_{\text{cls}}$ $\mathbf{e}_1$ $\mathbf{e}_2$ ... $\mathbf{e}_m$ $\mathbf{e}_{m+1}$
[CLS] x_1 x_2 ... x_m [SEP]

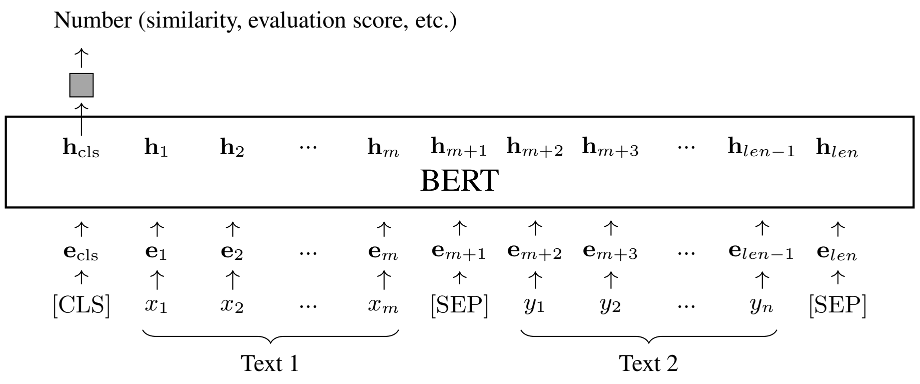
  Here the gray box denotes the prediction network. Many NLP problems can be categorized as text classification tasks, and there have been several text classification benchmarks for evaluating pre-trained models. For example, we can classify texts by their grammatical correctness (grammaticality) or emotional 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, $x_1...x_m$ and $y_1...y_n$. 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 $\mathbf{h}_{\text{cls}}$ vector, as follows


Class
$\mathbf{h}_{\text{cls}}$ $\mathbf{h}_1$ $\mathbf{h}_2$ $\cdots$ $\mathbf{h}_m$ $\mathbf{h}_{m+1}$ $\mathbf{h}_{m+2}$ $\mathbf{h}_{m+3}$ $\cdots$ $\mathbf{h}_{len-1}$ $\mathbf{h}_{len}$
BERT
$\mathbf{e}_{\text{cls}}$ $\mathbf{e}_1$ $\mathbf{e}_2$ $\cdots$ $\mathbf{e}_m$ $\mathbf{e}_{m+1}$ $\mathbf{e}_{m+2}$ $\mathbf{e}_{m+3}$ $\cdots$ $\mathbf{e}_{len-1}$ $\mathbf{e}_{len}$
[CLS] x_1 x_2 $\cdots$ x_m [SEP] y_1 y_2 $\cdots$ y_n [SEP]
Text 1
Text 2

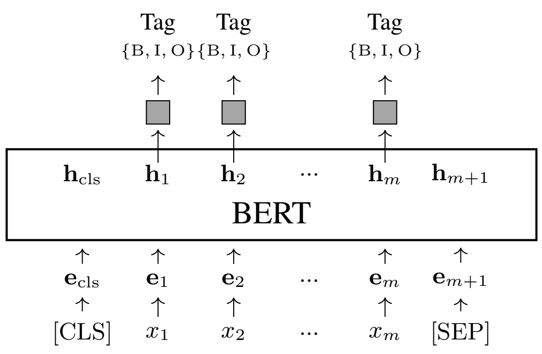
where $len = n + m + 2$. Text pair classification covers several problems, including semantic equivalence judgement (determine whether two texts are semantically equivalent) [Dolan and Brockett, 2005], text entailment judgement (determine whether a hypothesis can be logically inferred or entailed from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

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


Number (similarity, evaluation score, etc.)
$\mathbf{h}_{\text{cls}}$ $\mathbf{h}_1$ $\mathbf{h}_2$ $\cdots$ $\mathbf{h}_m$ $\mathbf{h}_{m+1}$ $\mathbf{h}_{m+2}$ $\mathbf{h}_{m+3}$ $\cdots$ $\mathbf{h}_{len-1}$ $\mathbf{h}_{len}$
BERT
$\mathbf{e}_{\text{cls}}$ $\mathbf{e}_1$ $\mathbf{e}_2$ $\cdots$ $\mathbf{e}_m$ $\mathbf{e}_{m+1}$ $\mathbf{e}_{m+2}$ $\mathbf{e}_{m+3}$ $\cdots$ $\mathbf{e}_{len-1}$ $\mathbf{e}_{len}$
[CLS] x_1 x_2 $\cdots$ x_m [SEP] y_1 y_2 $\cdots$ y_n [SEP]
Text 1
Text 2

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.

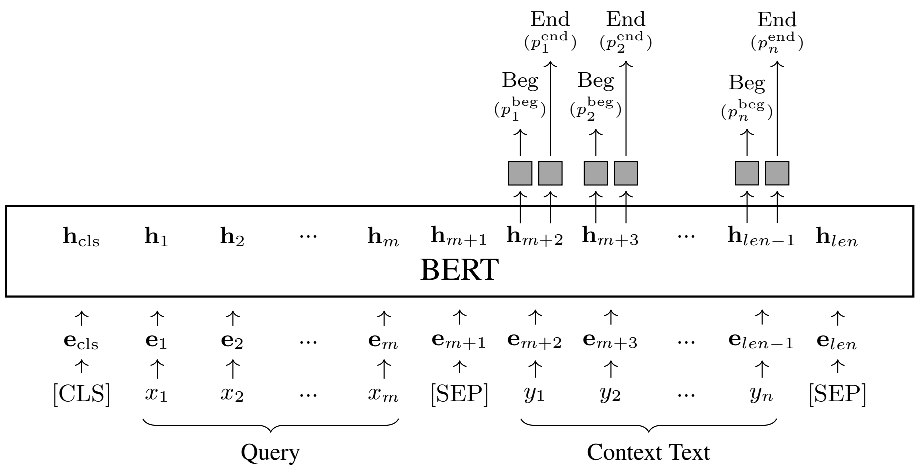


Here $\{\text{B}, \text{I}, \text{O}\}$ is the tag set of NER. For example, B-ORG means the beginning of an organization, I-ORG means the word is inside an organization, and O means the word does not belong to any named entity. This NER model can output a distribution over the tag set at each position,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

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

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 $x_1...x_m$ and a context text $y_1...y_n$. The goal is to identify a continuous span in $y_1...y_n$ 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



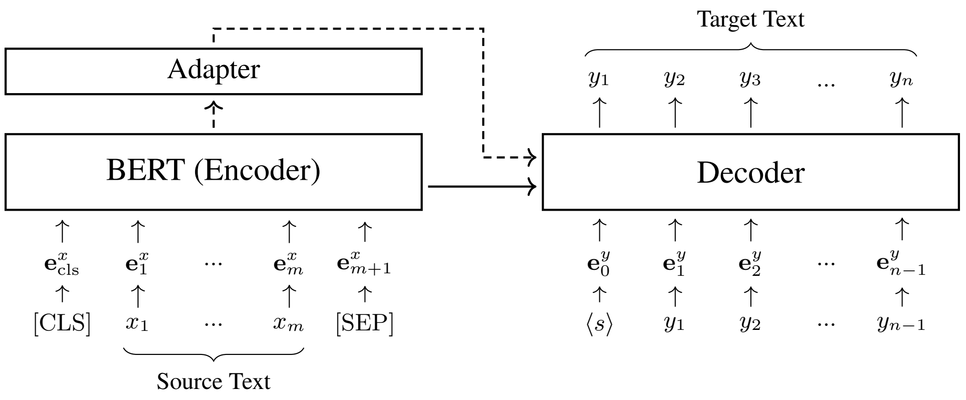
We pack the query and context text together to obtain the input sequence. The prediction networks are only applied to outputs for the context text, generating the probabilities p_j^{beg} and p_j^{end} at each position. The loss can be computed by summing the log likelihoods of the two models across the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

$$\mathrm{Loss} \;=\; -\frac{1}{n}\sum_{j=1}^{n}\big(\log p_j^{\mathrm{beg}} + \log p_j^{\mathrm{end}}\big) \tag{1.24}$$

At test time, we search for the best span by

$$(\hat{j}_1, \hat{j}_2) \;=\; \underset{1\le j_1\le j_2\le n}{\arg\max}\;\big(\log p_{j_1}^{\mathrm{beg}} + \log p_{j_2}^{\mathrm{end}}\big) \tag{1.25}$$

- **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.


Target Text
Adapter
y_1 y_2 y_3 ... y_n
BERT (Encoder)
Decoder
$\mathbf{e}_{\mathrm{cls}}^x$ $\mathbf{e}_1^x$ ... $\mathbf{e}_m^x$ $\mathbf{e}_{m+1}^x$
[CLS] x_1 ... x_m [SEP]
$\mathbf{e}_0^y$ $\mathbf{e}_1^y$ $\mathbf{e}_2^y$ ... $\mathbf{e}_{n-1}^y$
$\langle s\rangle$ y_1 y_2 ... y_{n-1}
Source Text

Here $x_1...x_m$ denotes the source sequence, $y_1...y_n$ denotes the target sequence, $\mathbf{e}_1^x...\mathbf{e}_m^x$ denotes the embedding sequence of $x_1...x_m$, and $\mathbf{e}_1^y...\mathbf{e}_n^y$ denotes the embedding sequence of $y_1...y_n$. 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 encode-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 $\{x_0, x_1, ..., x_m\}$ be a sequence of tokens, where x_0 is the start symbol $\langle s \rangle$ (or $\langle \text{SOS} \rangle$)[1]. The probability of this sequence can be defined using the chain rule

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

or alternatively in a logarithmic form

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

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

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

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

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

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

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

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

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

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

2.1.1 Decoder-only Transformers

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

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

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

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

or the pre-norm architecture

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

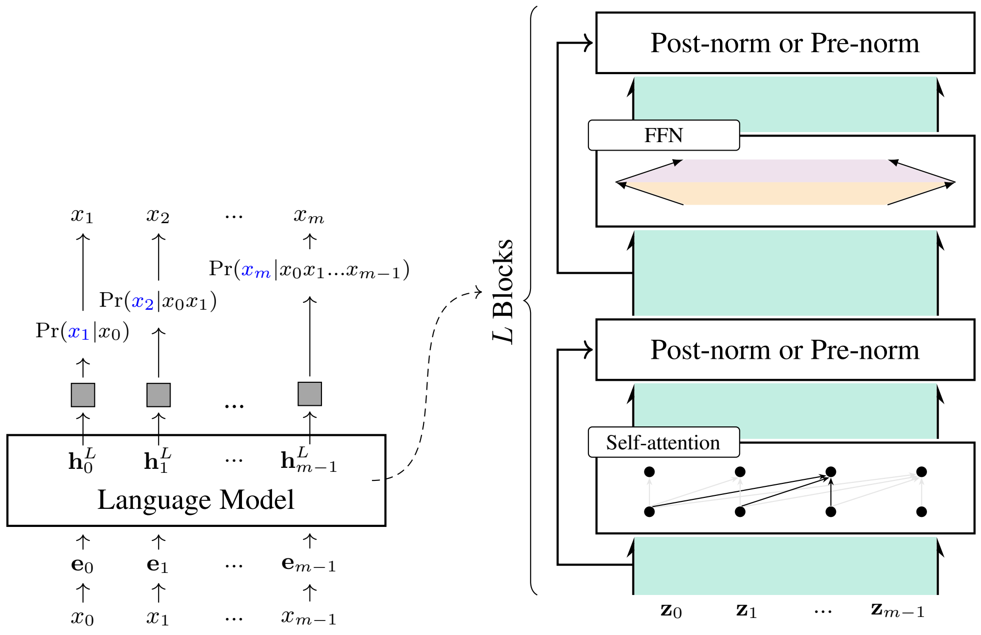


Fig. 2.1: The Transformer-decoder architecture for language modeling. The central components are L stacked Transformer blocks, each comprising a self-attention sub-layer and an FFN sub-layer. To prevent the model from accessing the right-context, a masking variable is incorporated into self-attention.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

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

2.1.2 Training LLMs

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

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

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

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

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

LLM	# of Parameters	Depth L	Width d	# of Heads (Q/KV)
GPT-1 [Radford et al., 2018]	0.117B	12	768	12/12
GPT-2 [Radford et al., 2019]	1.5B	48	1,600	25/25
GPT-3 [Brown et al., 2020]	175B	96	12,288	96/96
LLaMA2 [Touvron et al., 2023b]	7B	32	4,096	32/32
	13B	40	5,120	40/40
	70B	80	8,192	64/64
LLaMA3/3.1 [Dubey et al., 2024]	8B	32	4,096	32/8
	70B	80	8,192	64/8
	405B	126	16,384	128/8
Gemma2 [Team et al., 2024]	2B	26	2,304	8/4
	9B	42	3,584	16/8
	37B	46	4,608	32/16
Qwen2.5 [Yang et al., 2024]	0.5B	24	896	14/2
	7B	28	3,584	28/4
	72B	80	8,192	64/8
DeepSeek-V3 [Liu et al., 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 $\mathbf{x} = x_0...x_m$ denote a token sequence of context given by users, and $\mathbf{y} = y_1...y_n$ denote a token sequence following the context. Then, the inference of LLMs can be defined as a problem of finding the most likely sequence $\mathbf{y}$ based on $\mathbf{x}$:

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

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

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

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

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

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

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

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

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

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

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

or using an instruction-like template

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

or using a code-like template.

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

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

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

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

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

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

Yes

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

Yes

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

Yes

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

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

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

Show me a recipe for making ice cream.

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

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

\$6.00

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

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

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

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

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

For each 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,

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

We then define the loss function to be

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

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

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

The loss is calculated and back propagated only for <u>The result is 4 .</u>.

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 $\{\mathbf{y}_1, ..., \mathbf{y}_4\}$) 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.

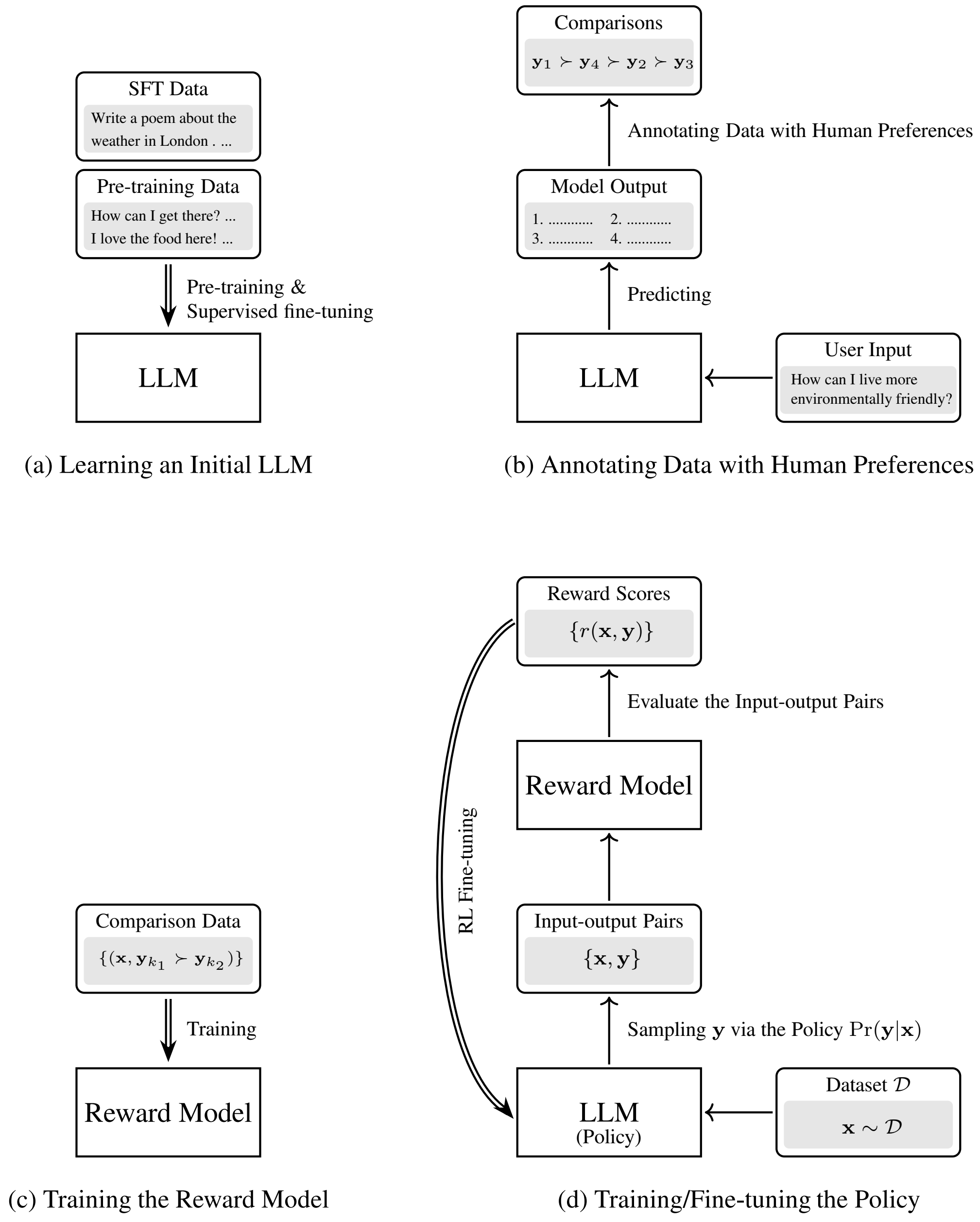

SFT Data
Write a poem about the weather in London . ...
Pre-training Data
How can I get there? ...
I love the food here! ...
Pre-training & Supervised fine-tuning
LLM
(a) Learning an Initial LLM
Comparisons
$\mathbf{y}_1 \succ \mathbf{y}_4 \succ \mathbf{y}_2 \succ \mathbf{y}_3$
Annotating Data with Human Preferences
Model Output
1. .... 2. ....
3. .... 4. ....
Predicting
LLM
User Input
How can I live more environmentally friendly?
(b) Annotating Data with Human Preferences
Comparison Data
$\{(\mathbf{x}, \mathbf{y}_{k_1} \succ \mathbf{y}_{k_2})\}$
Training
Reward Model
(c) Training the Reward Model
Reward Scores
$\{r(\mathbf{x}, \mathbf{y})\}$
Evaluate the Input-output Pairs
Reward Model
RL Fine-tuning
Input-output Pairs
$\{\mathbf{x}, \mathbf{y}\}$
Sampling $\mathbf{y}$ via the Policy $\Pr(\mathbf{y}|\mathbf{x})$
LLM
(Policy)
Dataset $\mathcal{D}$
$\mathbf{x} \sim \mathcal{D}$
(d) Training/Fine-tuning the Policy

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

$$\mathbf{y}_1 \succ \mathbf{y}_4 \succ \mathbf{y}_2 \succ \mathbf{y}_3$$

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

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

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

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

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

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

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

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

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 $\mathrm{Pr}_{\hat{\theta}+}(\mathbf{y}|\mathbf{x})$ 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 debugging[6].

Fix the bugs in this C language program.

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

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

Here is the corrected version of the program:

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

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

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

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

> Hi! I'm an assistant. How can I help you?
>
> Who won the FIFA World Cup 2022?
>
> Argentina won the FIFA World Cup 2022.
>
> Where was it held?
>
> The 2022 FIFA World Cup was held in Qatar.
>
> How many times has Argentina won the World Cup?
>
> Argentina has won the FIFA World Cup three times.
>
> ...

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

> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> The answer is 10.

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

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

> Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?
>
> The answer is 17.
>
> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> The answer is 12.

But the LLM still made mistakes this time. A reason for this might be that solving math problems does not only involve problem-answer mappings but also, to a larger extent, the underlying logical inference in multiple steps. A method to improve the inference abilities of LLMs is **chain-of-thought prompting** (**COT prompting**) [Wei et al., 2022c]. In COT prompting, we decompose complex reasoning problems into multiple problem-solving intermediate steps. These steps are demonstrated in prompts so that LLMs can be prompted to learn to mimic the underlying thought processes in solving the problems. COT prompting has been found to be very useful for achieving good results on challenging tasks, such as the GSM8K mathematical reasoning benchmark.

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

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

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

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

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

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

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

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

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

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

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

In the end, Jack has 7 apples left.

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

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

Example 1: “I had an amazing day at the park!”
Sentiment: Positive

Example 2: “The service at the restaurant was terrible.”
Sentiment: Negative

Example 3: “I think it’s going to rain today.”
Sentiment: Neutral

Text: “This movie was a fantastic journey through imagination.”
Sentiment: Positive

Translate the following Chinese phrases into English.

Example 1: “你好”
Translation: “Hello”

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

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

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

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

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 $\sim$ 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.

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 $\mathrm{output} = \mathrm{LNorm}(F(\mathrm{input})) + \mathrm{input}$.

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

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

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

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

where σ_{rms} is the root mean square of $\mathbf{h}$, that is, $\sigma_{\mathrm{rms}} = (\frac{1}{d}\sum_{k=1}^{d} h_k^2)^{\frac{1}{2}}$. 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 degeneration[8] [Dong et al., 2021]. A standard form of the FFNs used in these sub-layers can be expressed as

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

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 **rectified linear unit** (**ReLU**), given by

$$\sigma_{\mathrm{relu}}(\mathbf{h}) = \max(0, \mathbf{h}) \tag{2.26}$$

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 $\Pr(h \le \mathbf{h})$. Here h is a d-dimensional vector whose entries are drawn from the standard normal distribution $\mathrm{Gaussian}(0, 1)$[9]. Specifically, the GeLU function is defined to be

$$\begin{aligned} \sigma_{\mathrm{gelu}}(\mathbf{h}) &= \mathbf{h}\Pr(h \le \mathbf{h}) \\ &= \mathbf{h}\Phi(\mathbf{h}) \end{aligned} \tag{2.27}$$

where $\Phi(\mathbf{h})$ is the cumulative distribution function of $\mathrm{Gaussian}(0, 1)$, 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

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

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

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

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

As another example, consider $\sigma(\cdot)$ to be the Swish function $\sigma_{\mathrm{swish}}(\mathbf{h}) = \mathbf{h} \odot \mathrm{Sigmoid}(c\mathbf{h})$

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

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

[Ramachandran et al., 2017]. Then, the SwiGLU function is given by

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

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.

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

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

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

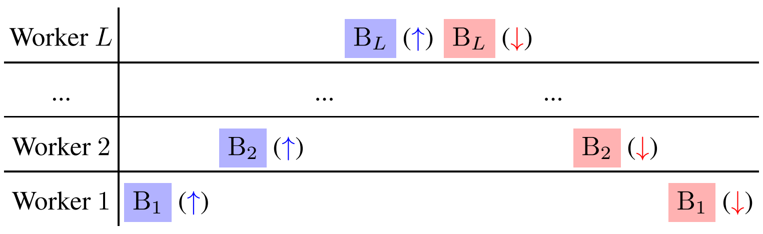
$$\theta_{t+1} = \theta_t - lr \cdot \frac{\partial L_{\theta_t}(\mathcal{D}_{\mathrm{mini}})}{\partial \theta_t} \tag{2.32}$$

  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. $\frac{\partial L_{\theta_t}(\mathcal{D}_{\mathrm{mini}})}{\partial \theta_t}$ is the gradient of the loss with respect to the parameters θ_t, and is computed on a minibatch of training sample $\mathcal{D}_{\mathrm{mini}}$. In data parallelism, we divide $\mathcal{D}_{\mathrm{mini}}$ into N smaller batches, denoted by $\{\mathcal{D}^1, ..., \mathcal{D}^N\}$. 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

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

  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

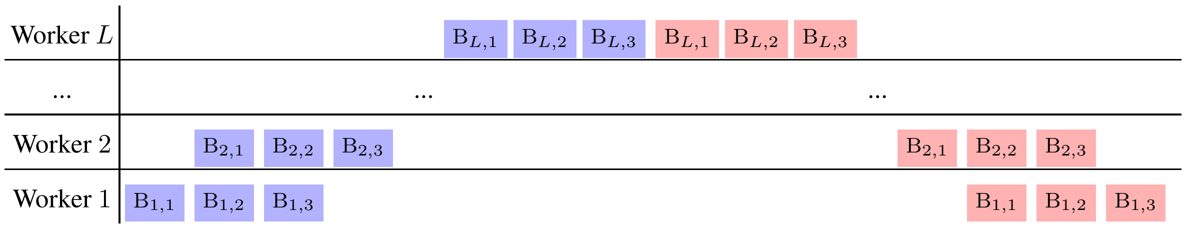
$$\mathbf{W}_h = \begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \tag{2.34}$$

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

$$\begin{aligned} \mathbf{h}\mathbf{W}_h &= \mathbf{h}\begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{h}\mathbf{W}_h^1 & \mathbf{h}\mathbf{W}_h^2 & ... & \mathbf{h}\mathbf{W}_h^M \end{bmatrix} \end{aligned} \tag{2.35}$$

We can perform matrix multiplications $\{\mathbf{h}\mathbf{W}_h^1, \mathbf{h}\mathbf{W}_h^2, ..., \mathbf{h}\mathbf{W}_h^M\}$ 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
...
Worker 2
Worker 1
$\mathrm{B}_{L,1}$ $\mathrm{B}_{L,2}$ $\mathrm{B}_{L,3}$ $\mathrm{B}_{L,1}$ $\mathrm{B}_{L,2}$ $\mathrm{B}_{L,3}$
$\mathrm{B}_{2,1}$ $\mathrm{B}_{2,2}$ $\mathrm{B}_{2,3}$ $\mathrm{B}_{2,1}$ $\mathrm{B}_{2,2}$ $\mathrm{B}_{2,3}$
$\mathrm{B}_{1,1}$ $\mathrm{B}_{1,2}$ $\mathrm{B}_{1,3}$ $\mathrm{B}_{1,1}$ $\mathrm{B}_{1,2}$ $\mathrm{B}_{1,3}$

Here $\mathrm{B}_{l,k}$ 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

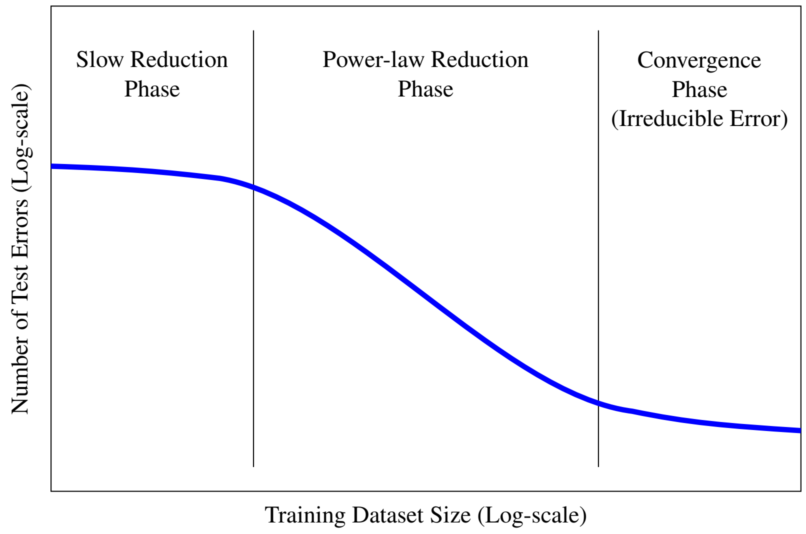


Fig. 2.3: A scaling law of test error against a variable of interest (e.g., training dataset size) [Hestness et al., 2017]. The curve of the scaling law can be divided into three phases. At the beginning, the number of test errors decreases slowly when more training data is used, but this only lasts 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.

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 **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

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

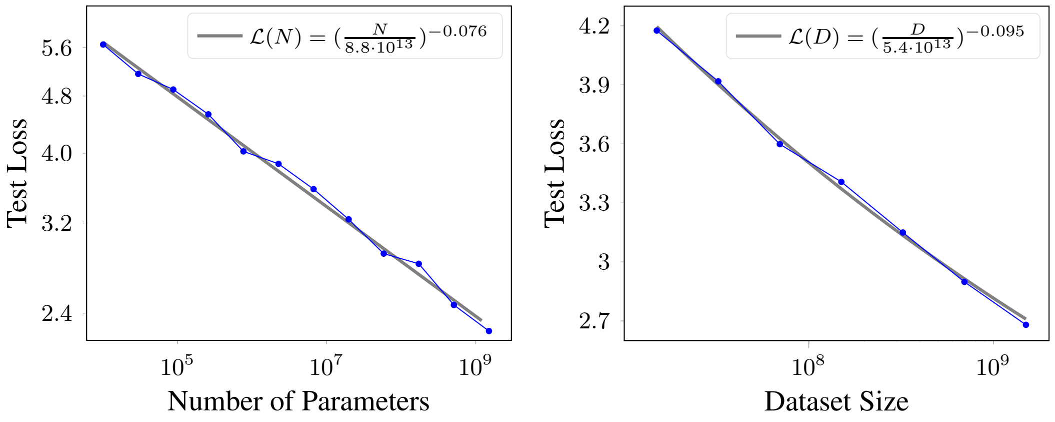


Fig. 2.4: Test loss against model size (N) and training dataset size (D) (data points are plotted for illustrative purposes). We plot test loss as a function of N, which is defined as $\mathcal{L}(N) = \big(\frac{N}{8.8\times 10^{13}}\big)^{-0.076}$, and a function of D, which is defined as $\mathcal{L}(D)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

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 $\mathcal{L}(N) = \big(\frac{N}{8.8\times 10^{13}}\big)^{-0.076}$ and $\mathcal{L}(D) = \big(\frac{D}{5.4\times 10^{13}}\big)^{-0.095}$ (see Figure 2.4).

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

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

where ϵ_∞ is the irreducible error that accounts for the error due to unknown variables, which is present even as $x \to \infty$. 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

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

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

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

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 $\{\mathbf{K}^{[1]}, ..., \mathbf{K}^{[n_u]}\}$, each corresponding to a segment of the sequence. Similarly, we can obtain the sub-matrices of $\mathbf{V}$, denoted by $\{\mathbf{V}^{[1]}, ..., \mathbf{V}^{[n_u]}\}$. 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

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

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

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

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

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

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

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

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

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

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

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

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

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

Each row vector $\begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i} & 0 & ... & 0\end{bmatrix}$ 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

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

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

In the original version of self-attention $\begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix}$ is assumed to be dense, that is, most of the values are non-zero. In sparse attention, some of the entries of $\begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix}$ are considered non-zero, and the remaining entries are simply ignored in computation. Suppose $G \subseteq \{0,...,i\}$ 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

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

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 transformations[10], the form of the resulting attention model is given by

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

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

$$\mu_i = \mu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \mathbf{v}_i \tag{2.50}$$

$$\nu_i = \nu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \tag{2.51}$$

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.

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

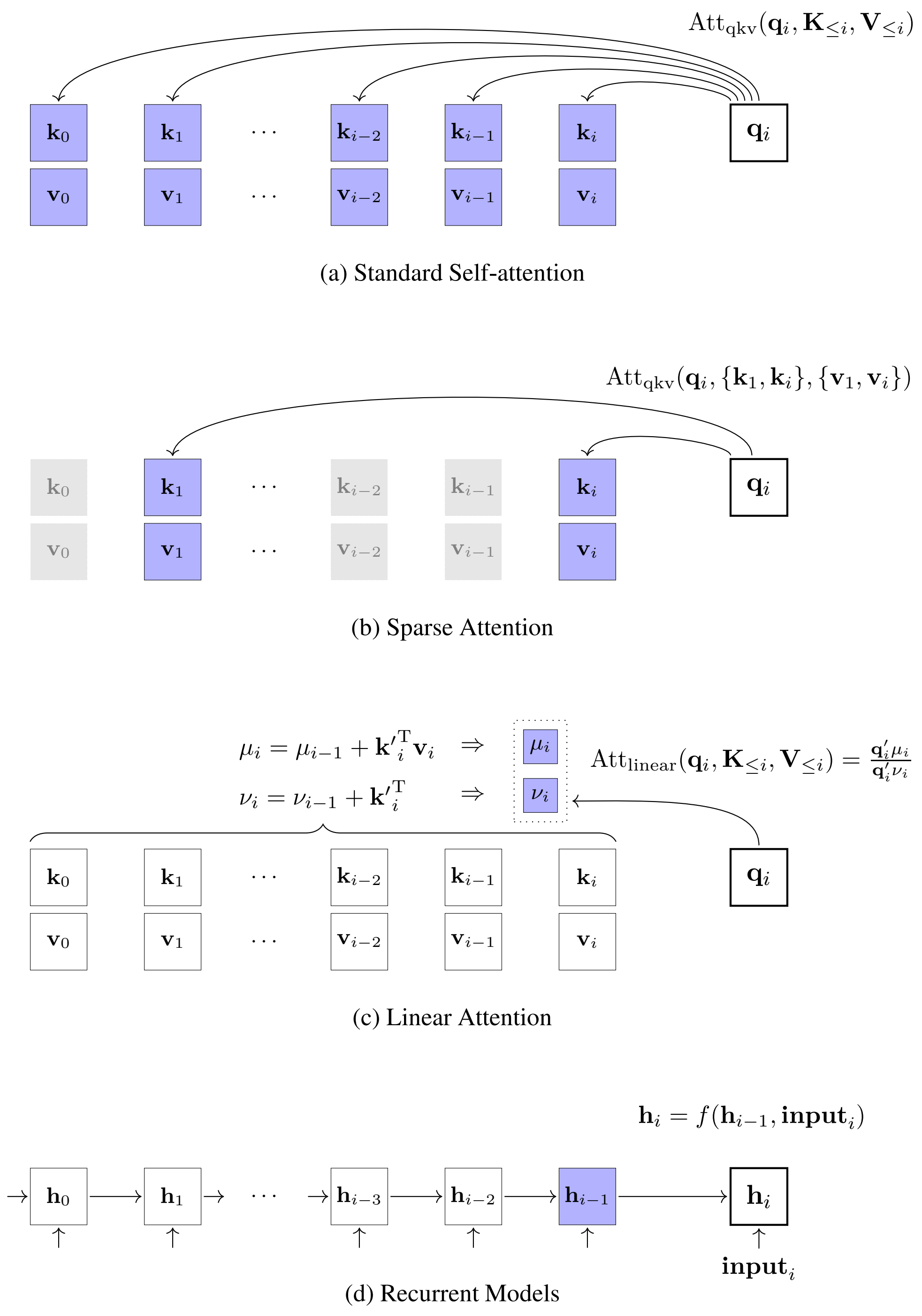

$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\le i}, \mathbf{V}_{\le i})$
$\mathbf{k}_0$ $\mathbf{k}_1$ ... $\mathbf{k}_{i-2}$ $\mathbf{k}_{i-1}$ $\mathbf{k}_i$ $\mathbf{q}_i$
$\mathbf{v}_0$ $\mathbf{v}_1$ ... $\mathbf{v}_{i-2}$ $\mathbf{v}_{i-1}$ $\mathbf{v}_i$
(a) Standard Self-attention
$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \{\mathbf{k}_1, \mathbf{k}_i\}, \{\mathbf{v}_1, \mathbf{v}_i\})$
(b) Sparse Attention
$\mu_i = \mu_{i-1} + \mathbf{k'}_i^{\text{T}} \mathbf{v}_i \Rightarrow \mu_i$
$\nu_i = \nu_{i-1} + \mathbf{k'}_i^{\text{T}} \Rightarrow \nu_i$
$\text{Att}_{\text{linear}}(\mathbf{q}_i, \mathbf{K}_{\le i}, \mathbf{V}_{\le i}) = \frac{\mathbf{q}'_i \mu_i}{\mathbf{q}'_i \nu_i}$
(c) Linear Attention
$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{input}_i)$
$\mathbf{h}_0$ $\mathbf{h}_1$ ... $\mathbf{h}_{i-3}$ $\mathbf{h}_{i-2}$ $\mathbf{h}_{i-1}$ $\mathbf{h}_i$
$\mathbf{input}_i$
(d) Recurrent Models

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.

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 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

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

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

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

  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

$$\mathrm{Mem} = \Big(\frac{\sum_{j=i-n_c+1}^{i} \mathbf{k}_j}{n_c}, \frac{\sum_{j=i-n_c+1}^{i} \mathbf{v}_j}{n_c}\Big) \tag{2.54}$$

  Alternatively, we can use a weighted version of moving average

$$\mathrm{Mem} = \Big(\frac{\sum_{j=i-n_c+1}^{i} \beta_{j-i+n_c}\mathbf{k}_j}{\sum_{j=1}^{n_c} \beta_j}, \frac{\sum_{j=i-n_c+1}^{i} \beta_{j-i+n_c}\mathbf{v}_j}{\sum_{j=1}^{n_c} \beta_j}\Big) \tag{2.55}$$

  Here $\{\beta_1, ..., \beta_{n_c}\}$ 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., $\beta_1 < \beta_2 < ... < \beta_{n_c-1} < \beta_{n_c}$) in order to give larger weight to positions that are closer to

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

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

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

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

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

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

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

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

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

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

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

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

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

where $[\mathrm{Mem}, \mathrm{CMem}]$ 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

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

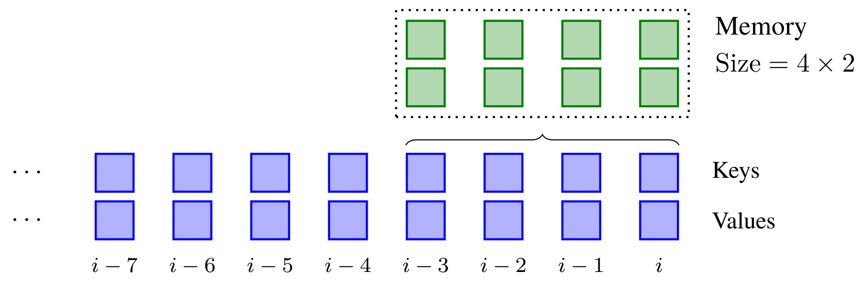
The popped key-value pairs are then used to update the compressive memory CMem. These n_s key-value pairs are compressed into $\frac{n_s}{c}$ key-value pairs via a compression network. CMem is an FIFO which appends the compressed $\frac{n_s}{c}$ key-value pairs to the tail of the queue, and drops the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

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

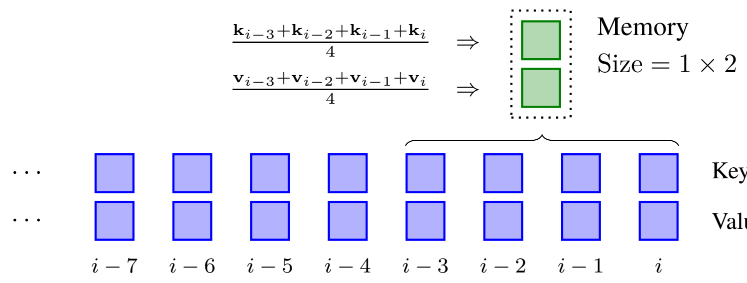
where C_{kv}^k represents the set of compressed key-value pairs. Implicit in the compressive Transformer model is that local context should be represented explicitly with minimal information loss, while long-range context can be more compressed.

- 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.

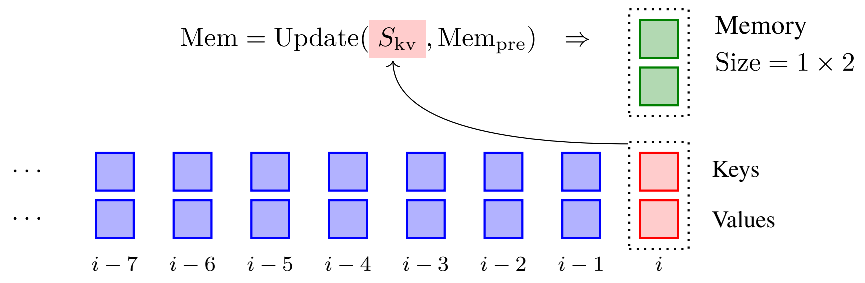
Figure 2.6 shows illustrations of the above approaches. Note that, while we focus on optimization of the KV cache here, this issue is closely related to those discussed in the previous section. All of the methods we have mentioned so far can broadly be categorized as efficient attention approaches, which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



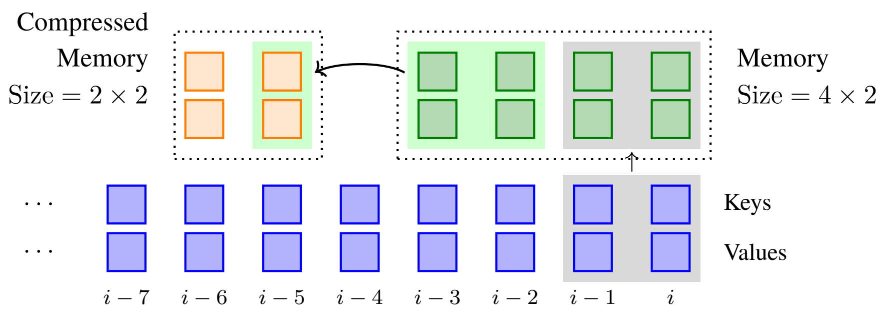
(a) Window-based Cache


$\frac{\mathbf{k}_{i-3}+\mathbf{k}_{i-2}+\mathbf{k}_{i-1}+\mathbf{k}_i}{4} \Rightarrow$
$\frac{\mathbf{v}_{i-3}+\mathbf{v}_{i-2}+\mathbf{v}_{i-1}+\mathbf{v}_i}{4} \Rightarrow$
Memory
Size $= 1 \times 2$
Keys
Values
$i-7$ $i-6$ $i-5$ $i-4$ $i-3$ $i-2$ $i-1$ i

(b) Moving Average-based Cache


$\mathrm{Mem} = \mathrm{Update}(S_{\mathrm{kv}}, \mathrm{Mem}_{\mathrm{pre}}) \Rightarrow$
Memory
Size $= 1 \times 2$
Keys
Values
$i-7$ $i-6$ $i-5$ $i-4$ $i-3$ $i-2$ $i-1$ i

(c) Recurrent Network as Cache



(d) Hybrid Cache (Compressed Memory + Local Memory)

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.

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

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

Suppose we have a set of keys $\{\mathbf{k}_j\}$ with corresponding values $\{\mathbf{v}_j\}$, and suppose we store these key-value pairs in a vector database[12]. 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 $\mathrm{Mem}_{k\mathrm{nn}}$. 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 $\mathrm{Mem}_{k\mathrm{nn}}$. There are, of course, several ways to incorporate Mem and $\mathrm{Mem}_{k\mathrm{nn}}$ into the attention model. For example, we might simply combine them to form a single KV cache $[\mathrm{Mem}, \mathrm{Mem}_{k\mathrm{nn}}]$, and attend $\mathbf{q}_i$ to $[\mathrm{Mem}, \mathrm{Mem}_{k\mathrm{nn}}]$ via standard QKV attention. Or we might use Mem and $\mathrm{Mem}_{k\mathrm{nn}}$ in separate attention steps. An example of such approaches is the model developed by Wu et al. [2021]. It linearly combines the two types of attention, given by

$$\begin{aligned}
\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}, \mathrm{Mem}_{k\mathrm{nn}}) &= \mathbf{g} \odot \mathrm{Att}_{\mathrm{local}} + (1 - \mathbf{g}) \odot \mathrm{Att}_{k\mathrm{nn}} && (2.64) \\
\mathrm{Att}_{\mathrm{local}} &= \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}) && (2.65) \\
\mathrm{Att}_{k\mathrm{nn}} &= \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}_{k\mathrm{nn}}) && (2.66)
\end{aligned}$$

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

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

generalized context. A problem with this approach is that the computational cost would be large if many sequences are involved. Since these sequences are part of our training data, we can build and optimize an index for the vectors in the datastore before running the LLMs. As a result, the retrieval of similar vectors can be very efficient, as in most vector databases.

In fact, all the above-mentioned methods can be viewed as instances of a retrieval-based approach. Instead of using retrieval results to improve attention, we can apply this approach in other ways as well. One application of k-NN-based search is **k-NN language modeling** (or **k-NN LM**) [Khandelwal et al., 2020]. The idea is that, although it is attempting to extend the context used in self-attention by incorporating nearest neighbors in representation learning, in practice, similar hidden states in Transformers are often highly predictive of similar tokens in subsequent positions. In k-NN LM, each item in the datastore is a key-value tuple $(\mathbf{z}, w)$, where $\mathbf{z}$ represents a hidden state of the LLM at a position, and w represents the corresponding prediction. A typical way to create the datastore is to collect the output vector of the Transformer layer stack and the corresponding next token for each position of each sequence in a training dataset. During inference, we have a representation $\mathbf{h}_i$ given a prefix. Given this representation, we first search the datastore for k closest matching data items $\{(\mathbf{z}_1, w_1), ..., (\mathbf{z}_k, w_k)\}$. Here $\{w_1, ..., w_k\}$ 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,

$$\mathrm{Pr}_{k\mathrm{nn}}(\cdot|\mathbf{h}_i) \quad = \quad \mathrm{Softmax}(\begin{bmatrix} -d_0 & \cdots & -d_{|V|} \end{bmatrix}) \tag{2.67}$$

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 $\mathrm{Pr}_{k\mathrm{nn}}(\cdot|\mathbf{h}_i)$ and the LLM output distribution $\mathrm{Pr}_{\mathrm{lm}}(\cdot|\mathbf{h}_i)$

$$\mathrm{Pr}(\cdot|\mathbf{h}_i) \quad = \quad \lambda \cdot \mathrm{Pr}_{k\mathrm{nn}}(\cdot|\mathbf{h}_i) + (1 - \lambda) \cdot \mathrm{Pr}_{\mathrm{lm}}(\cdot|\mathbf{h}_i) \tag{2.68}$$

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

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 $\{\mathbf{c}_1, ..., \mathbf{c}_k\}$ from the datastore via efficient IR techniques[13]. These search results are combined with the original context via a prompting

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

template $g(\cdot)$[14], resulting in an augmented input for the LLM

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

Then, we use $\mathbf{x}'$ as the context and predict the following text using the model $\Pr(\mathbf{y}|\mathbf{x}')$. 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.

2.3.3.3 Memory Capacity

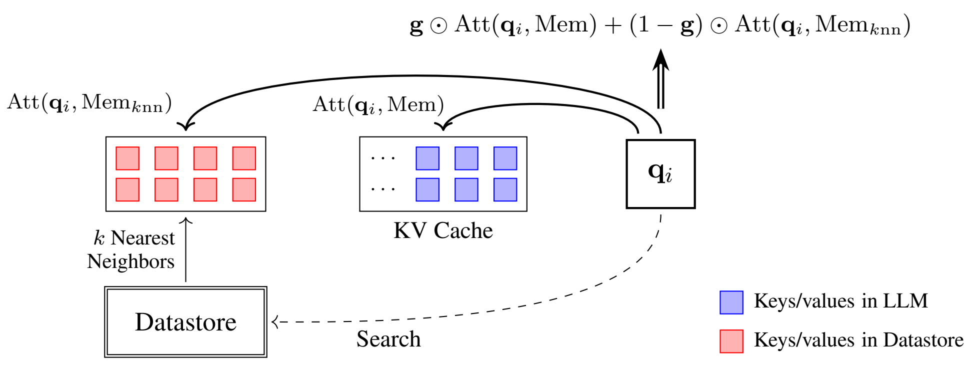
A memory model in LLMs, in the form of a simple key-value cache or a datastore, can broadly be seen as an encoder of contextual information. Ideally, before we say that a memory model is representative of the entire context in token prediction, we need to make sure that the model can accurately represent 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].

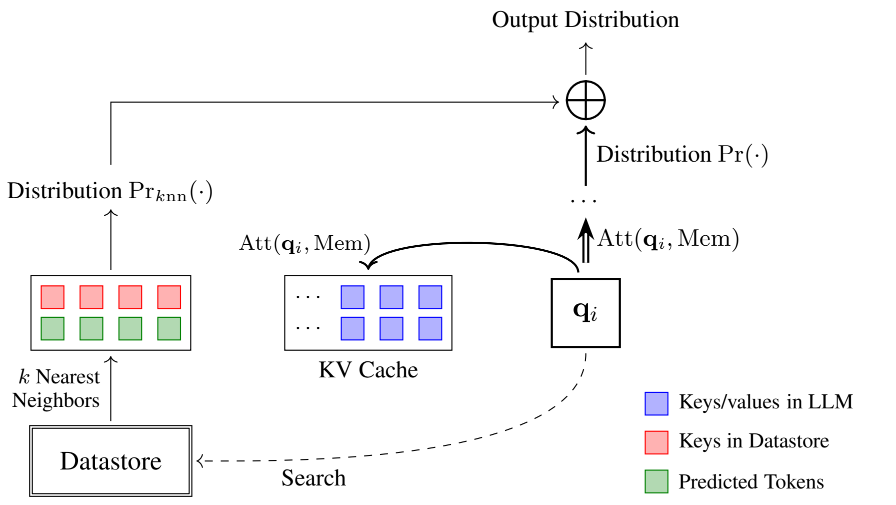
[14] For example, the template could be:

```
message = {*$\mathbf{c}_1$*} ... {*$\mathbf{c}_k$*}
input: {*$\mathbf{x}$*}
output: ____
```



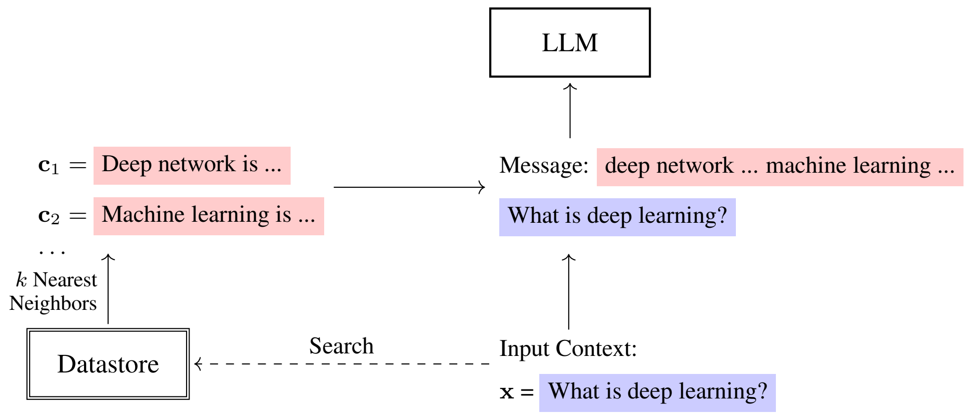
$\mathbf{g} \odot \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}) + (1-\mathbf{g}) \odot \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}_{knn})$
$\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}_{knn})$
$\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem})$
$\mathbf{q}_i$
KV Cache
k Nearest Neighbors
Datastore
Search
Keys/values in LLM
Keys/values in Datastore

(a) k-NN Search Augmented Attention



Output Distribution
Distribution $\mathrm{Pr}(\cdot)$
Distribution $\mathrm{Pr}_{knn}(\cdot)$
$\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem})$
$\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem})$
$\mathbf{q}_i$
KV Cache
k Nearest Neighbors
Datastore
Search
Keys/values in LLM
Keys in Datastore
Predicted Tokens

(b) k-NN Language Modeling



LLM
$\mathbf{c}_1$ = Deep network is ...
$\mathbf{c}_2$ = Machine learning is ...
...
Message: deep network ... machine learning ...
What is deep learning?
k Nearest Neighbors
Datastore
Search
Input Context:
$\mathbf{x}$ = What is deep learning?

(c) Retrieval-augmented Generation

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

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

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

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

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

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, $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), ..., (\mathbf{K}_{\leq i}^{[\tau]}, \mathbf{V}_{\leq i}^{[\tau]})\}$.

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 $\{\mathbf{q}_i^{[1]}, ..., \mathbf{q}_i^{[\tau]}\}$, each corresponding to a different head. For each head, we have

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

Figure 2.8 (c) illustrates this model. By sharing keys and values, the size of the KV cache would

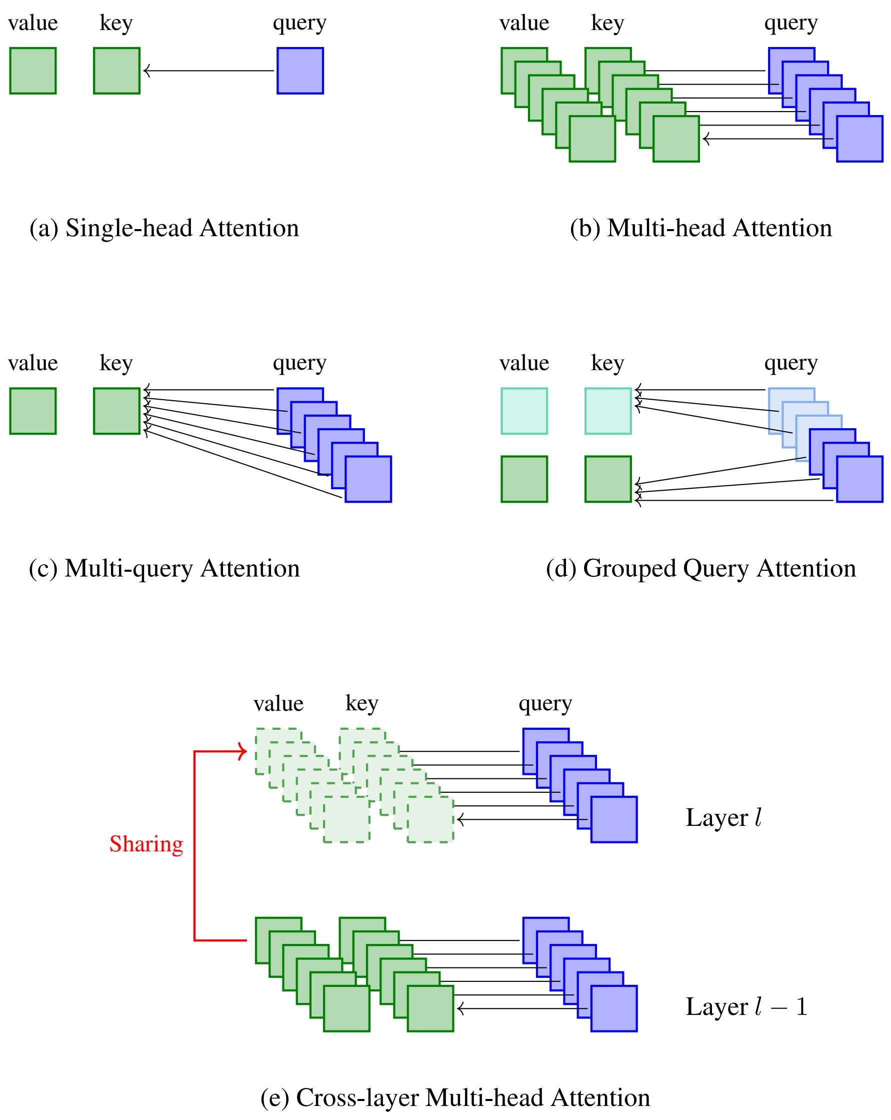


Fig. 2.8: Illustration of QKV attention based on different multi-head and sharing mechanisms. (a) = single-head attention, and (b-e) = attention with multiple heads.

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 $\{(\mathbf{K}_{\le i}^{[1]}, \mathbf{V}_{\le i}^{[1]}), ..., (\mathbf{K}_{\le i}^{[n_g]}, \mathbf{V}_{\le i}^{[n_g]})\}$. 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

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

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 GQA model.

Sharing can also be performed across layers. Such a method falls into the family of shared weight and shared activation methods, which have been extensively used in Transformers [Dehghani et al., 2018; Lan et al., 2020]. For example, one can share KV activations or attention weights across layers to reduce both computation and memory footprints [Xiao et al., 2019; Brandon et al., 2024]. Figure 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

$$\mathbf{e}_i = \mathbf{x}_i + \mathrm{PE}(i) \tag{2.74}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ denotes the token embedding, and $\mathrm{PE}(i) \in \mathbb{R}^d$ denotes the positional embedding. In general, the token embedding $\mathbf{x}_i$ is a position-independent vector, and so the positional embedding $\mathrm{PE}(i)$ is used to encode the positional context. A straightforward approach is to treat $\mathrm{PE}(i)$ 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 ($m >> m_l$). 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

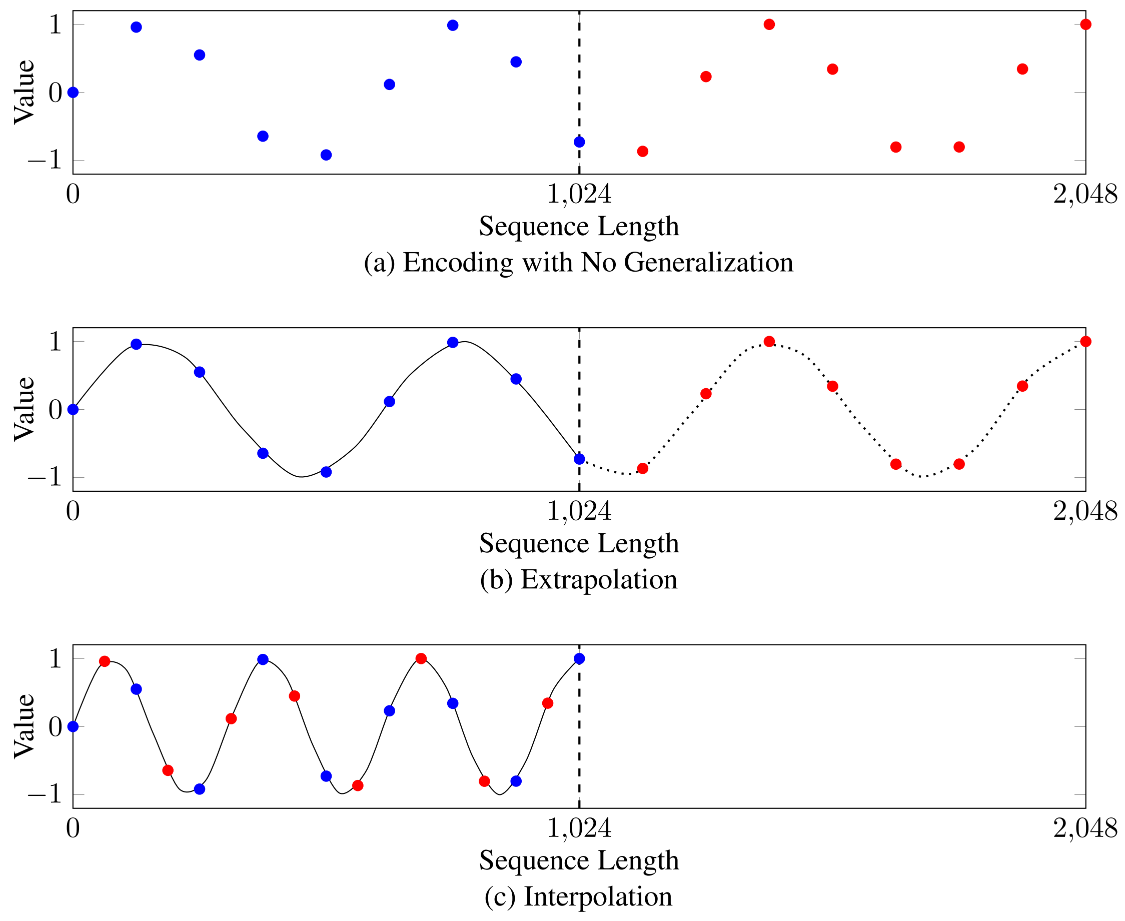


Fig. 2.9: Illustrations of different positional embedding methods for a range of positions. Blue points represent the positions that have been observed during training, and red points represent the positions that are newly observed at test time. In sub-figure (a), the encoding model only memorizes the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

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.

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

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

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

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

$\mathrm{PE}(i,j)$ can be defined in several different ways. Here, we consider the T5 version of relative positional embedding, called the T5 bias [Raffel et al., 2020]. For each pair of query $\mathbf{q}_i$ and key $\mathbf{k}_j$, the offset between them is defined to be[15]

$$d(i,j) = i - j \tag{2.77}$$

A simple design for the bias $\mathrm{PE}(i,j)$ is to share the same learnable variable for all query-key pairs with the same offset, i.e., $\mathrm{PE}(i,j) = u_{i-j}$, 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 $b(\mathrm{i}-\mathrm{j}) = i - j$.
- 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

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

where the parameter $\mathrm{dist}_{\max}$ is typically set to a relatively large number to indicate the

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

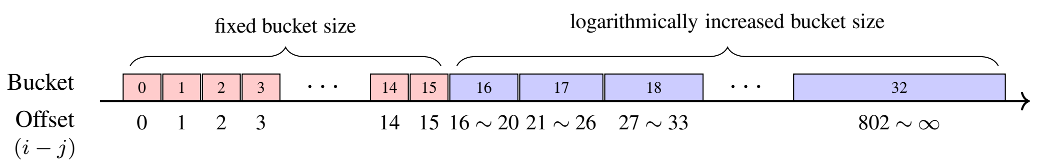


Fig. 2.10: Illustration of distributing query-key offsets into buckets in the T5 model ($n_b = 32$ and $\text{dist}_{\max} = 1024$). Boxes represent buckets. In the first half of the buckets, we use a fixed bucket size. In the second half of the buckets, we increase the bucket size logarithmically. The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

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

$$\begin{aligned} & b(i-j) \\ = & \begin{cases} i-j & 0 \le i-j < \frac{n_b+1}{2} \\ \min(n_b, \frac{n_b+1}{2} + \lfloor \frac{\log(i-j)-\log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max})-\log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \rfloor) & i-j \ge \frac{n_b+1}{2} \end{cases} \end{aligned} \tag{2.79}$$

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.

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 $\mathbf{k}_j$ becomes[16]

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

The parameters $\{u_0, ..., u_{n_b}\}$ 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.

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

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

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

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

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

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

This model can be interpreted as adding a fixed penalty to $\mathbf{q}_i\mathbf{k}_j^{\mathrm{T}}$ 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.

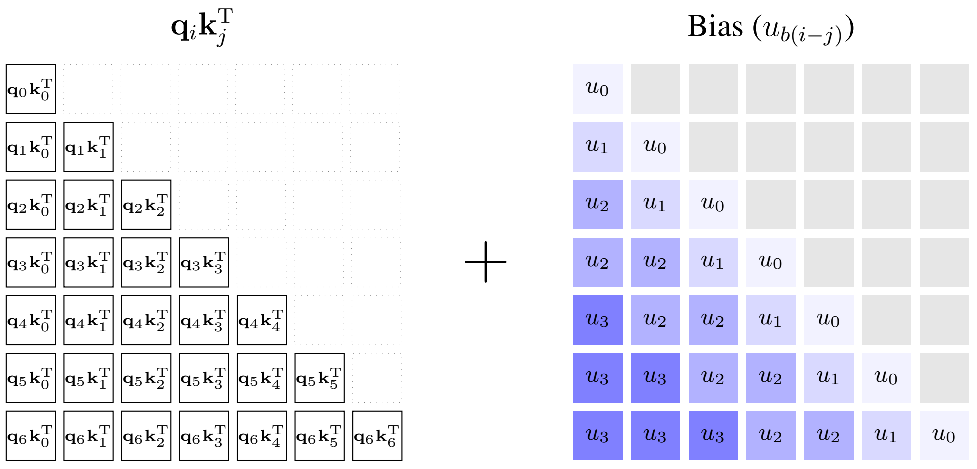
In general, the scalar β should be tuned on a validation dataset. However, Press et al. [2022] found that setting β to values decreasing geometrically by a factor of $\frac{1}{2^a}$ for multi-head attention performs well on a variety of tasks. Specifically, for a self-attention sub-layer involving n_{head}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

$$\beta_k = \frac{1}{2^{\frac{8}{k}}} \tag{2.84}$$

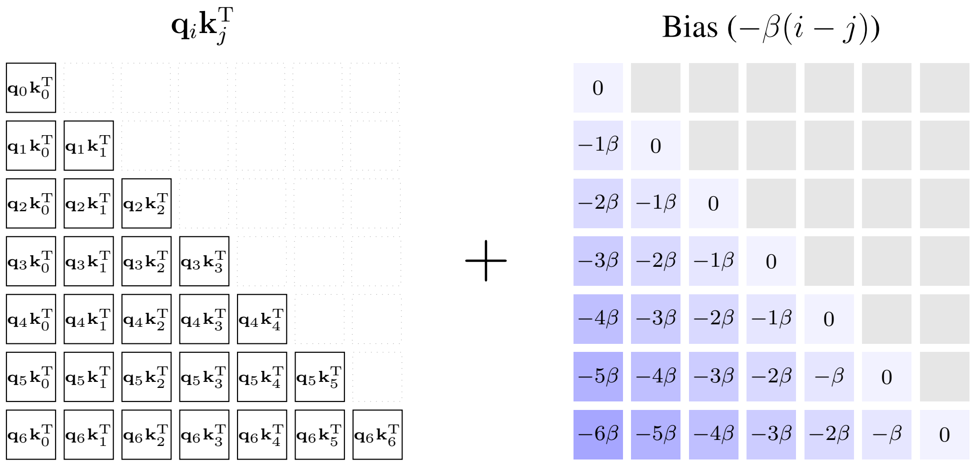
The ALiBi approach provides a simple form of relative positional embeddings. There are other similar methods for designing query-key biases using the offset $i-j$. Table 2.4 shows a comparison of such biases. As an aside it is worth noting that the form of the right-hand side of Eq. (2.82) is very similar 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



(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.

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

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

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

Entry	Query-Key Bias ($\mathrm{PE}(i,j)$)
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]	$\sum_{k=1}^{\bar{d}/2} \cos\left((\ i-j\ )/10000^{2k/\bar{d}}\right)$
FIRE [Li et al., 2024]	$f\left(\psi(\ i-j\ )/\psi(\max(m_{\mathrm{len}}, i))\right)$

Table 2.4: Query-key biases as relative positional embeddings. β, β_1, β_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.

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 expressed mathematically in the form

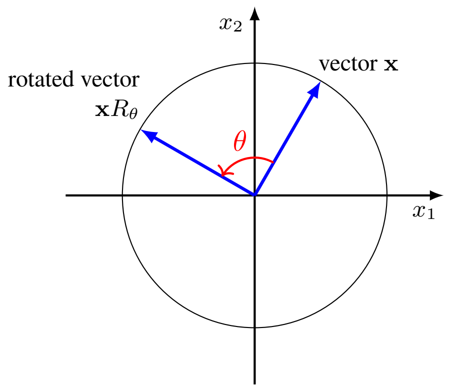
$$\begin{aligned}\mathrm{Ro}(\mathbf{x},\theta) &= \mathbf{x}R_\theta \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta \cdot x_1 - \sin\theta \cdot x_2 & \sin\theta \cdot x_1 + \cos\theta \cdot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

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

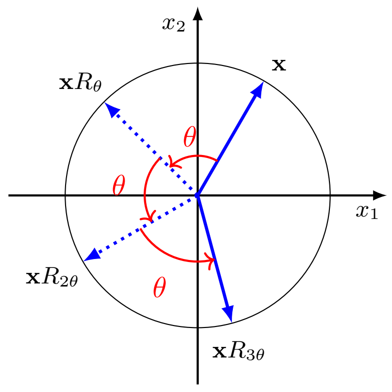
$$\begin{aligned}\mathrm{Ro}(\mathbf{x},t\theta) &= \mathbf{x}R_{t\theta} \\ &= \begin{bmatrix} \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 & \sin t\theta \cdot x_1 + \cos t\theta \cdot x_2 \end{bmatrix} \end{aligned} \tag{2.87}$$

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 $\mathrm{Ro}(\mathbf{x}, i\theta)$. 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.

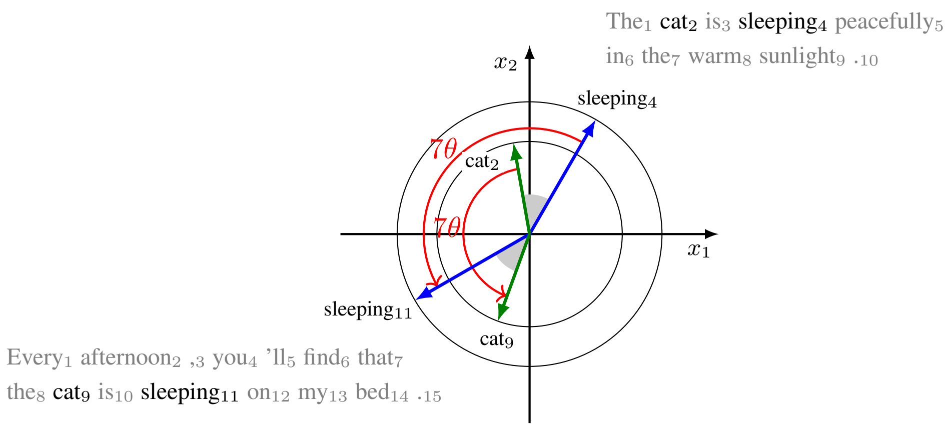
A popular way to understand vector rotation is to define it in complex spaces. It is easy to transform each vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ in the 2D Euclidean space $\mathbb{R}^2$ to a complex number $\mathbf{x}' = x_1 + \mathrm{i}x_2$ in the complex space $\mathbb{C}$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



(a) Single-step Rotation



(b) Multi-step Rotation


The$_1$ cat$_2$ is$_3$ sleeping$_4$ peacefully$_5$ in$_6$ the$_7$ warm$_8$ sunlight$_9$.$_{10}$
x_2
sleeping$_4$
7θ
cat$_2$
7θ
x_1
sleeping$_{11}$
cat$_9$
Every$_1$ afternoon$_2$,$_3$ you$_4$ 'll$_5$ find$_6$ that$_7$ the$_8$ cat$_9$ is$_{10}$ sleeping$_{11}$ on$_{12}$ my$_{13}$ bed$_{14}$.$_{15}$

(c) Angles between embeddings of two tokens at different positions

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.

operation can be re-expressed in the form

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

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

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

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 $\mathrm{Ro}(\mathbf{x}, t\theta)$ and $\mathrm{Ro}(\mathbf{y}, s\theta)$ is can be written as a function of $(t-s)\theta$

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

Given this result, if we consider $\mathrm{Ro}(\mathbf{x}, t\theta)$ and $\mathrm{Ro}(\mathbf{y}, s\theta)$ 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 $\mathbf{x}' = \begin{bmatrix} x'_1 & x'_2 & \dots & x'_{d/2} \end{bmatrix} = \begin{bmatrix} x_1 + \mathrm{i}x_2 & x_3 + \mathrm{i}x_4 & \dots & x_{d-1} + \mathrm{i}x_d \end{bmatrix}$, where each consecutive pair of items forms a complex number. Then, the rotary positional embedding in the complex space is given by

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

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

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

where $R_{t\theta_k} = \begin{bmatrix} \cos t\theta_k & \sin t\theta_k \\ -\sin t\theta_k & \cos t\theta_k \end{bmatrix}$. $\theta = \begin{bmatrix} \theta_1, \dots, \theta_{d/2} \end{bmatrix}$ 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.

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

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

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

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 $\mathrm{Ro}(\mathbf{x}_i, i\theta)$ in which $\theta = \begin{bmatrix}\theta_1, ..., \theta_{d/2}\end{bmatrix}$ are the parameters. $\mathrm{Ro}(\mathbf{x}_i, i\theta)$ can be cast in the form of a linear combination of two periodic functions (see Eq. (2.93))

$$\cos i\theta = \begin{bmatrix}\cos i\theta_1 & ... & \cos i\theta_{d/2}\end{bmatrix} \tag{2.95}$$

$$\sin i\theta = \begin{bmatrix}\sin i\theta_1 & ... & \sin i\theta_{d/2}\end{bmatrix} \tag{2.96}$$

θ_k is a exponential function of k and takes the form

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

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

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

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

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

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

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

Another method of positional interpolation is to scale the base[17]. 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

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

[17]This method was first proposed in https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware_scaled_rope_allows_llama_models_to_have/

Solving this equation, we obtain

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

This gives an embedding model

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

where

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

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-haystack[18] and passkey retrieval tasks [Mohtashami and Jaggi, 2024; Chen et al., 2023c], for instance, LLMs are required to identify and extract a small, relevant piece of information from a large volume of given text. The assumption here is that an LLM with sufficient memory should remember earlier parts of the text as it processes new information. This LLM can thus pick out the relevant details, which might be sparse and hidden among much irrelevant information, from the text. Alternatively, in copy memory tasks (or copy tasks for short), LLMs are used to repeat the input text or a specific segment multiple times. These tasks were initially proposed to test the extent to which recurrent models can retain and recall previously seen tokens [Hochreiter and Schmidhuber, 1997; Arjovsky et al., 2016], and have been adopted in evaluating recent LLMs [Bulatov et al., 2022; Gu and Dao, 2023].

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

Although many methods have been developed, there is still no general way to evaluate long-context LLMs [Liu et al., 2024c]. One problem is that most of these methods focus on specific aspects of LLMs, rather than their fundamental ability to model very long contexts. Even though an LLM can pick out the appropriate piece of text from the input, we cannot say that it truly understands the entire context. Instead, it might just remember some important parts of the context, or even simply recall the answer via the model learned in pre-training. Moreover, the data used in many tasks is small-scale and relatively preliminary, leading to discrepancies between evaluation results and actual application performance. A more interesting issue is that the results of LLMs are influenced by many other factors and experimental setups, for example, using different prompts can lead to very different outcomes. This makes evaluation even more 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.

[18] `https://github.com/gkamradt/LLMTest_NeedleInAHaystack`