Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

June 6, 2025

Contents

Chapter 8

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.

8.1 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

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

probability of this sequence can be defined using the chain rule

$$\begin{aligned}\Pr(x_0,...,x_m) &= \Pr(x_0)\cdot\Pr(x_1|x_0)\cdot\Pr(x_2|x_0,x_1)\cdots\Pr(x_m|x_0,...,x_{m-1}) \\ &= \prod_{i=0}^{m}\Pr(x_i|x_0,...,x_{i-1}) \end{aligned} \tag{8.1}$$

or alternatively in a logarithmic form

$$\log\Pr(x_0,...,x_m) = \sum_{i=0}^{m}\log\Pr(x_i|x_0,...,x_{i-1}) \tag{8.2}$$

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 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

$$\hat{x}_i = \arg\max_{x_i\in\mathcal{V}}\Pr(x_i|x_0,...,x_{i-1}) \tag{8.3}$$

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. (8.1) and (8.2). To illustrate, consider the generation of the following three words given the prefix '$\langle s\rangle$ a', as shown in Table 8.1. Now we discuss how LLMs are constructed, trained, and applied.

8.1.1 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

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

[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 $\log\Pr(x_0)=0$.

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

Table 8.1: 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.

blocks (or layers). Each Transformer block has two stacked sub-layers, one for self-attention modeling and one for FFN modeling. These sub-layers can be defined using the post-norm architecture

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input}) + \text{input}) \tag{8.4}$$

or the pre-norm architecture

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input})) + \text{input} \tag{8.5}$$

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

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

$$\text{Att}_{\text{qkv}}(\mathbf{Q},\mathbf{K},\mathbf{V}) \quad = \quad \text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\text{T}}}{\sqrt{d}} + \mathbf{Mask})\mathbf{V} \tag{8.6}$$

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

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

$$F(\mathbf{H}) \quad = \quad \text{Merge}(\text{head}_1,...,\text{head}_\tau)\mathbf{W}^{\text{head}} \tag{8.7}$$

where $\text{Merge}(\cdot)$ representees a concatenation of its inputs, and $\mathbf{W}^{\text{head}} \in \mathbb{R}^{d\times d}$ represents a

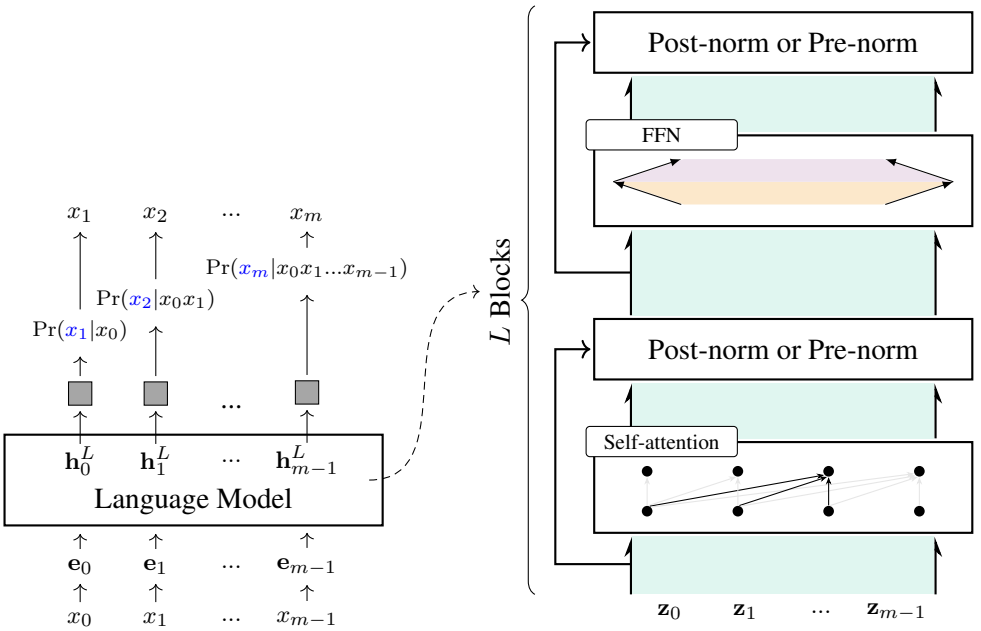


Figure 8.1: The Transformer-decoder architecture for language modeling. The central components are L stacked Transformer blocks, each comprising a self-attention sub-layer and an FFN sub-layer. To prevent the model from accessing the right-context, a masking variable is incorporated into self-attention. 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.

parameter matrix. head_j is the output of QKV attention on a sub-space of representation

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{Q}^{[j]}, \mathbf{K}^{[j]}, \mathbf{V}^{[j]}) \tag{8.8}$$

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

$$\mathbf{Q}^{[j]} = \mathbf{H}\mathbf{W}_j^q \tag{8.9}$$
$$\mathbf{K}^{[j]} = \mathbf{H}\mathbf{W}_j^k \tag{8.10}$$
$$\mathbf{V}^{[j]} = \mathbf{H}\mathbf{W}_j^v \tag{8.11}$$

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

$$\begin{bmatrix} \Pr(\cdot|x_0,...,x_{m-1}) \\ \vdots \\ \Pr(\cdot|x_0,x_1) \\ \Pr(\cdot|x_0) \end{bmatrix} = \text{Softmax}(\mathbf{H}^L\mathbf{W}^o) \tag{8.12}$$

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 8.1 shows the Transformer architecture for language modeling. Applying this language 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 8.2 shows the model sizes for a few LLMs, as well as their model setups.

8.1.2 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

$$\mathcal{L}_\theta(\mathbf{x}) = \sum_{i=1}^{m} \log \Pr_\theta(x_i|x_0,...,x_{i-1}) \tag{8.13}$$

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

$$\hat{\theta} = \arg\max_\theta \sum_{\mathbf{x}\in\mathcal{D}} \mathcal{L}_\theta(\mathbf{x}) \tag{8.14}$$

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 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,

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., 2023]	7B	32	4,096	32/32

Table 8.2: 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).

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 8.2 we will present more discussions on these issues.

8.1.3 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 $\mathbf{x}$:

$$\begin{aligned} \hat{\mathbf{y}} &= \arg\max_{\mathbf{y}} \log \Pr(\mathbf{y}|\mathbf{x}) \\ &= \arg\max_{\mathbf{y}} \sum_{i=1}^{n} \log \Pr(y_i|x_0, ..., x_m, y_1, ..., y_{i-1}) \end{aligned} \tag{8.15}$$

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. (8.2). 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 sub-sequences 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 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.

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

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

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

or using an instruction-like template

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

or using a code-like template.

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

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

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

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

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

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

Yes

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

Yes

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

Yes

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

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

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

Show me a recipe for making ice cream.

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

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

$6.00

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

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

To acquire instruction-following abilities, a certain amount of fine-tuning data is required. This data may include diverse instructions and possible responses. It has been found that scaling the number of fine-tuning tasks is beneficial for improving the performance of LLMs [Chung et al., 2022]. Note that although more fine-tuning data is favorable, the amount of this data is generally orders of magnitude smaller than that of the pre-training data. For example, LLMs can be fine-tuned with tens or hundreds of thousands of samples, or even fewer if these samples are of high quality [Zhou et al., 2023; Chen et al., 2023a], 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. (8.14) to obtain the objective of fine-tuning

$$\tilde{\theta} \quad = \quad \arg\max_{\hat{\theta}^+} \sum_{\text{sample} \in \mathcal{D}_{\text{tune}}} \mathcal{L}_{\hat{\theta}^+}(\text{sample}) \tag{8.16}$$

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

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

$$\text{sample} \quad = \quad [\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}] \tag{8.17}$$

We then define the loss function to be

$$\mathcal{L}_{\hat{\theta}^+}(\text{sample}) \quad = \quad -\log \Pr_{\hat{\theta}^+}(\mathbf{y}_{\text{sample}} | \mathbf{x}_{\text{sample}}) \tag{8.18}$$

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

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

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

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

While we focus on instruction fine-tuning for an illustrative example here, fine-tuning techniques play an important role in developing various LLMs and are more widely used. Examples include fine-tuning LLMs as chatbots using dialog data, and adapting these models to handle very long sequences. The wide application of fine-tuning has led researchers to improve these techniques, such as designing more efficient fine-tuning algorithms. While the

research on fine-tuning is fruitful, in this section we just give a flavour of the key steps involved. We will see more detailed discussions on this topic in the following chapters.

8.1.4 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.

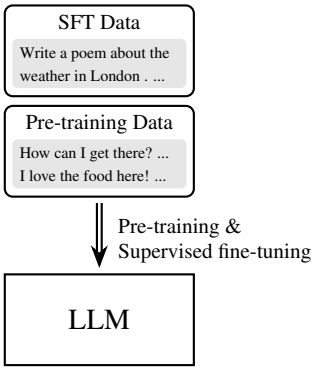
- Build an initial policy using pre-training and instruction fine-tuning.
- Use the policy to generate multiple outputs for each input, and then collect human feedback on these outputs (e.g., comparisons of the outputs).
- Learn a reward model from the human feedback.
- Fine-tune the policy with the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

Figure 8.2 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.

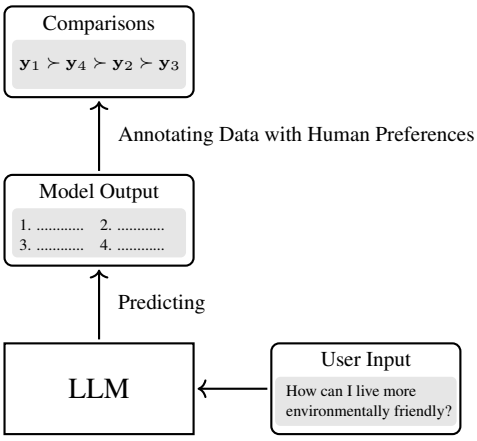
Suppose we have trained an LLM via pre-training and instruction fine-tuning. This LLM is deployed to respond to requests from users. For example, a user may input

> How can I live a more environmentally friendly life?

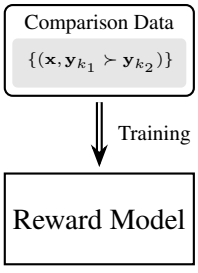
We use the LLM to generate 4 different outputs (denoted by $\{\mathbf{y}_1, ..., \mathbf{y}_4\}$) by sampling the output space



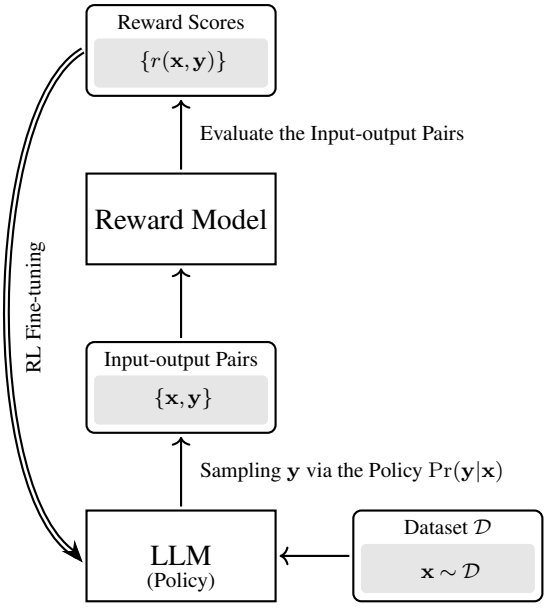
(a) Learning an Initial LLM



(b) Annotating Data with Human Preferences



(c) Training the Reward Model



(d) Training/Fine-tuning the Policy

Figure 8.2: 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.

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

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

$$\text{Loss}_\omega(\mathcal{D}_r) \quad = \quad -\mathbb{E}_{(\mathbf{x},\mathbf{y}_{k_1},\mathbf{y}_{k_2})\sim\mathcal{D}_r} \log(\text{Sigmoid}(R_\omega(\mathbf{x},\mathbf{y}_{k_1}) - R_\omega(\mathbf{x},\mathbf{y}_{k_2}))) \quad (8.19)$$

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

$$\begin{aligned} & \text{Loss}_\omega(\mathcal{D}_r) \\ = \quad & -\sum \Pr(\mathbf{x}) \cdot \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}|\mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x},\mathbf{y}_{k_1}) - R_\omega(\mathbf{x},\mathbf{y}_{k_2}))) \\ = \quad & -\frac{1}{K}\sum \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}|\mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x},\mathbf{y}_{k_1}) - R_\omega(\mathbf{x},\mathbf{y}_{k_2}))) \end{aligned} \quad (8.20)$$

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

$$\hat{\omega} \quad = \quad \arg\min_\omega \text{Loss}_\omega(\mathcal{D}_r) \quad (8.21)$$

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. (8.16), we obtain a simple training objective for RL fine-tuning

$$\tilde{\theta} \quad = \quad \arg\max_{\hat{\theta}^+} \mathbb{E}_{(\mathbf{x},\mathbf{y}_{\hat{\theta}^+})\sim\mathcal{D}_{\text{rlft}}} R_{\hat{\omega}}(\mathbf{x},\mathbf{y}_{\hat{\theta}^+}) \tag{8.22}$$

where the optimal parameters $\tilde{\theta}$ are obtained by fine-tuning the pre-trained parameters $\hat{\theta}$. $\mathcal{D}_{\text{rlft}}$ is the RL fine-tuning dataset. For each sample $(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+})$, $\mathbf{x}$ is sampled from a prepared dataset of input sequences, and $\mathbf{y}_{\hat{\theta}^+}$ is sampled from the distribution $\Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x})$ given by the policy.

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

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

8.1.5 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].

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

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].

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.

[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.

> 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 9.

8.2 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.

8.2.1 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 8.3), 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	# 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 8.3: Amounts of training data used in some LLMs in terms of the number of tokens.

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

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

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

Another issue with collecting large-scale data is the privacy concern. If LLMs are trained on data from extensive sources, this potentially leads to risks regarding the exposure of sensitive information, such as intellectual property and personal data. This is particularly

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

8.2.2 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.

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 8.1.1, a sub-layer can be based on either the post-norm architecture, in which layer normalization is performed right after a residual block, or the pre-norm architecture, in which layer normalization is performed inside a residual block. While both of these architectures are widely used in Transformer-based systems [Wang et al., 2019], the pre-norm architecture has proven to be especially useful in training deep Transformers. Given this, most LLMs are based on the pre-norm architecture, expressed as $\text{output} = \text{LNorm}(F(\text{input})) + \text{input}$.

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

$$\text{LNorm}(\mathbf{h}) \quad = \quad \alpha \cdot \frac{\mathbf{h} - \mu}{\sigma + \epsilon} + \beta \tag{8.23}$$

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

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

normalization function is given by

$$\text{LNorm}(\mathbf{h}) = \alpha \cdot \frac{\mathbf{h}}{\sigma_{\text{rms}} + \epsilon} + \beta \tag{8.24}$$

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

2. 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

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_f + \mathbf{b}_f \tag{8.25}$$

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

$$\sigma_{\text{relu}}(\mathbf{h}) = \max(0, \mathbf{h}) \tag{8.26}$$

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

$$\begin{aligned} \sigma_{\text{gelu}}(\mathbf{h}) &= \mathbf{h}\Pr(h \leq \mathbf{h}) \\ &= \mathbf{h}\Phi(\mathbf{h}) \end{aligned} \tag{8.27}$$

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

Another family of activation functions which is popular in LLMs is **gated linear unit**

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

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

(**GLU**)-based functions. The basic form of GLUs is given by

$$\sigma_{\text{glu}}(\mathbf{h}) \quad = \quad \sigma(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{8.28}$$

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

$$\sigma_{\text{geglu}}(\mathbf{h}) \quad = \quad \sigma_{\text{gelu}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{8.29}$$

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

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

$$\sigma_{\text{swiglu}}(\mathbf{h}) \quad = \quad \sigma_{\text{swish}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{8.30}$$

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.

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. (8.25) to obtain an FFN with no bias terms

$$\text{FFN}(\mathbf{h}) \quad = \quad \sigma(\mathbf{h}\mathbf{W}_h)\mathbf{W}_f \tag{8.31}$$

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.

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 8.3.

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.

8.2.3 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

$$\theta_{t+1} \quad = \quad \theta_t - lr \cdot \frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \tag{8.32}$$

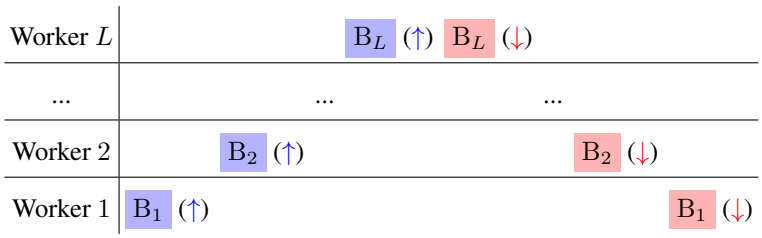
  where the new parameters θ_{t+1} is obtained by updating the latest parameters θ_t with a small step lr in the direction of the negative loss gradient. $\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t}$ is the gradient of the loss with respect to the parameters θ_t, and is computed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

$$\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \quad = \quad \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^1)}{\partial \theta_t}}_{\text{worker 1}} + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^2)}{\partial \theta_t}}_{\text{worker 2}} + \cdots + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^N)}{\partial \theta_t}}_{\text{worker } N} \tag{8.33}$$

  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.



Here B_l denotes the computation of block l, and the symbols ↑ and ↓ 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. (8.25)). We can slice the matrix $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ vertically to a sequence of M sub-matrices

$$\mathbf{W}_h = \begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \tag{8.34}$$

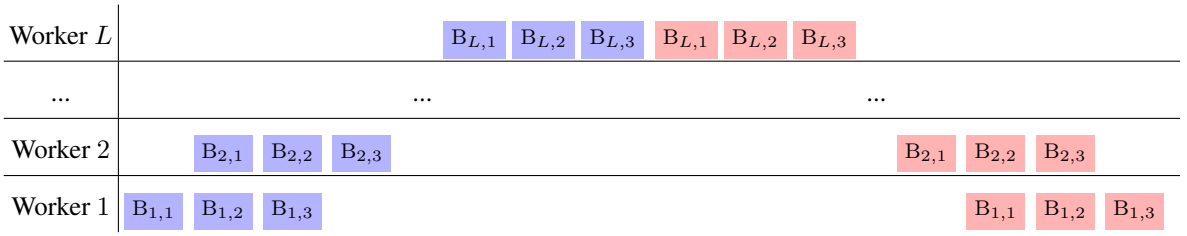
where each sub-matrix $\mathbf{W}_h^k$ has a shape of $d \times \frac{d_h}{M}$. The multiplication of $\mathbf{h}$ with $\mathbf{W}_h$ can be expressed as

$$\begin{aligned} \mathbf{h}\mathbf{W}_h &= \mathbf{h}\begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{h}\mathbf{W}_h^1 & \mathbf{h}\mathbf{W}_h^2 & ... & \mathbf{h}\mathbf{W}_h^M \end{bmatrix} \end{aligned} \tag{8.35}$$

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.



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.

8.2.4 Scaling Laws

The success of LLMs reveals that training larger language models using more resources can lead to improved model performance. Researchers have explained this as **scaling laws** of LLMs. More specifically, scaling laws describe the relationships between the performance of LLMs and the attributes of LLM training, such as the model size, the amount of computation used for training, and the amount of training data. For example, Hestness et al. [2017] show that the performance of deep neural networks is a power-law-like function of the training data size. In the beginning, when the amount of training data is not large, the performance of the model improves slowly. Afterward, when more training data is used, the model enters a phase of rapid performance improvement, and the performance curve resembles a power-law curve. Ultimately, the improvement in performance becomes slow again, and more data does not lead to significant gains. Figure 8.3 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

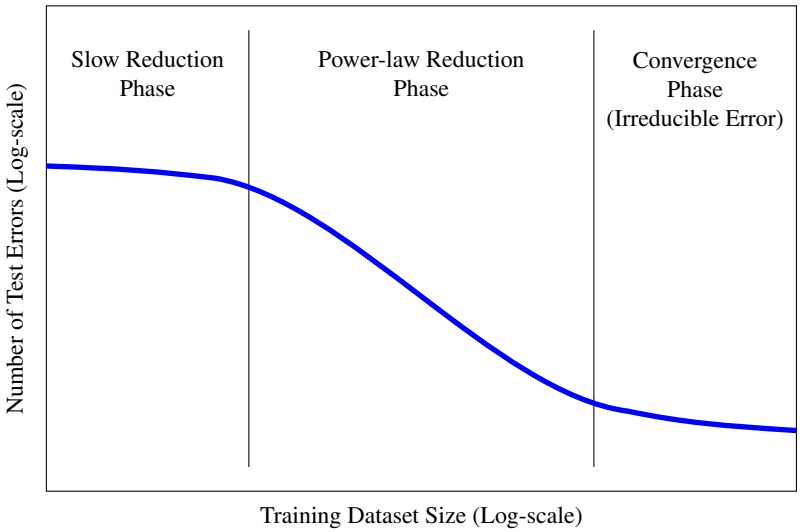


Figure 8.3: A scaling law of test error against a variable of interest (e.g., training dataset size) [Hestness et al., 2017]. The curve of the scaling law can be divided into three phases. At the beginning, the number of test errors decreases slowly when more training data is used, but this only lasts 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.

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

$$\mathcal{L}(x) \quad = \quad ax^b \tag{8.36}$$

where a and b are parameters that are estimated empirically. Despite its simplicity, this function has successfully interpreted the scaling ability of language models and machine translation systems in terms of model size (denoted by N) and training dataset size (denoted by D) [Gordon et al., 2021; Hestness et al., 2017]. For example, Kaplan et al. [2020] found that the performance of their language model improves as a power law of either N or D after an initial transient period, and expressed these relationships using $\mathcal{L}(N) = \left(\frac{N}{8.8\times10^{13}}\right)^{-0.076}$ and $\mathcal{L}(D) = \left(\frac{D}{5.4\times10^{13}}\right)^{-0.095}$ (see Figure 8.4).

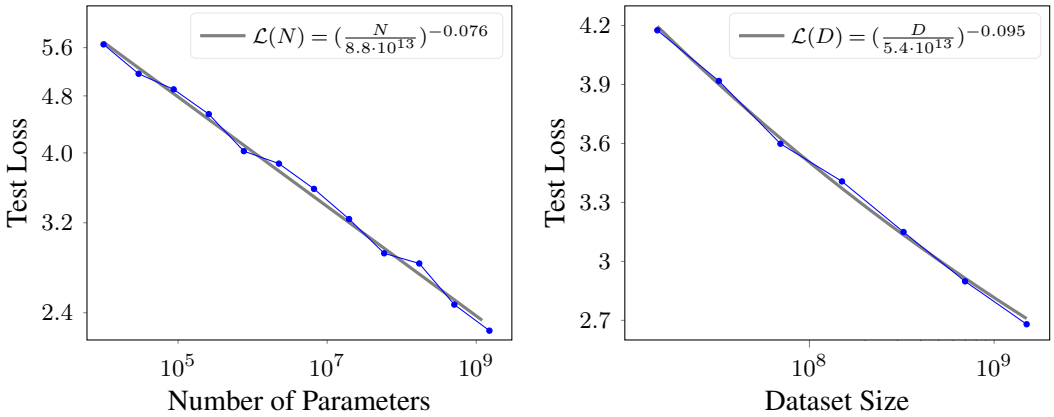


Figure 8.4: Test loss against model size (N) and training dataset size (D) (data points are plotted for illustrative purposes). We plot test loss as a function of N, which is defined as $\mathcal{L}(N) = \left(\frac{N}{8.8 \times 10^{13}}\right)^{-0.076}$, and a function of D, which is defined as $\mathcal{L}(D) = \left(\frac{D}{5.4 \times 10^{13}}\right)^{-0.095}$ [Kaplan et al., 2020].

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

$$\mathcal{L}(x) \quad = \quad ax^b + \epsilon_\infty \tag{8.37}$$

where ϵ_∞ is the irreducible error that accounts for the error due to unknown variables, which is present even as $x \to \infty$. Eq. (8.37) 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

$$\mathcal{L}(N, D) \quad = \quad aN^b + cD^d + \epsilon_\infty \tag{8.38}$$

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

$$\mathcal{L}(N, D) \quad = \quad \underbrace{\frac{406.4}{N^{0.34}}}_{\text{model scaling}} + \underbrace{\frac{410.7}{D^{0.28}}}_{\text{dataset scaling}} + \underbrace{1.69}_{\text{irreducible error}} \tag{8.39}$$

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.

8.3 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 Chapter 6 extensively covers many methods in this strand. 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.

8.3.1 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., 2023; 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

$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \sum_{j=0}^{m-1} \alpha_{i,j} \mathbf{v}_j \tag{8.40}$$

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

$$\beta_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} + \mathrm{Mask}(i,j) \tag{8.41}$$

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

$$\begin{aligned} \alpha_{i,j} &= \mathrm{Softmax}(\beta_{i,j}) \\ &= \frac{\exp(\beta_{i,j})}{\sum_{j'} \exp(\beta_{i,j'})} \end{aligned} \tag{8.42}$$

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. (8.42) (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. (8.42) 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. (8.42) as

$$\alpha_{i,j} = \frac{\overbrace{\exp(\beta_{i,j})}^{\text{node } u}}{\underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[1]}} \exp(\beta_{i,j'})}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[n_u]}} \exp(\beta_{i,j'})}_{\text{node } n_u}} \tag{8.43}$$

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. (8.40). The problem can be re-expressed as

$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[1]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node 1}} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } u} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[n_u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{node } n_u} \tag{8.44}$$

Like Eq. (8.43), Eq. (8.44) 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 8.2.3. As a result, the implementation of these methods can be based on the same parallel processing library.

8.3.2 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

$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \alpha(\mathbf{Q},\mathbf{K})\mathbf{V} \tag{8.45}$$

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

$$\begin{aligned}\alpha(\mathbf{Q},\mathbf{K}) &= \mathrm{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}}+\mathbf{Mask}) \\ &= \begin{bmatrix} \alpha_{0,0} & 0 & 0 & ... & 0 \\ \alpha_{1,0} & \alpha_{1,1} & 0 & ... & 0 \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & ... & \alpha_{m-1,m-1} \end{bmatrix}\end{aligned} \tag{8.46}$$

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

$$\begin{aligned}\mathrm{Att}_{\mathrm{qkv}}(\mathbf{q}_i,\mathbf{K}_{\le i},\mathbf{V}_{\le i}) &= \begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix}\begin{bmatrix}\mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i\end{bmatrix} \\ &= \sum_{j=0}^{i}\alpha_{i,j}\mathbf{v}_j\end{aligned} \tag{8.47}$$

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

$$\text{Att}_{\text{sparse}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j \in G} \alpha'_{i,j} \mathbf{v}_j \tag{8.48}$$

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

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &\approx \text{Att}_{\text{linear}}(\mathbf{q}'_i, \mathbf{K}'_{\leq i}, \mathbf{V}_{\leq i}) \\ &= \frac{\mathbf{q}'_i \mu_i}{\mathbf{q}'_i \nu_i} \end{aligned} \tag{8.49}$$

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

$$\mu_i = \mu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \mathbf{v}_i \tag{8.50}$$

$$\nu_i = \nu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \tag{8.51}$$

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

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

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 8.5 shows a comparison of the models discussed in this subsection. Since these models, along with others not mentioned here, have been intensively discussed in Chapter 6 and in related surveys [Tay et al., 2020], a detailed discussion of them is precluded here.

8.3.3 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.

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

$$\mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}) \quad = \quad \mathrm{Att}_{\mathrm{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \tag{8.52}$$

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.

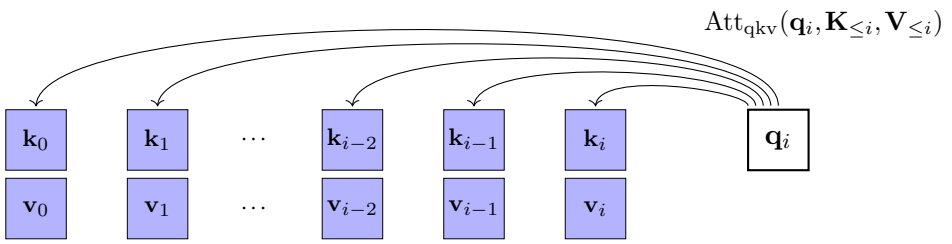
- One of the simplest methods is to consider a fixed-size window of previous keys and values. Mem is therefore given by

$$\mathrm{Mem} \quad = \quad (\mathbf{K}_{[i-n_c+1,i]}, \mathbf{V}_{[i-n_c+1,i]}) \tag{8.53}$$

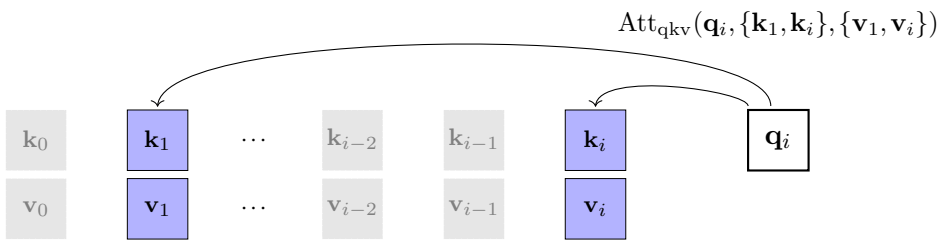
  where n_c denotes the size of the window. The notation $\mathbf{K}_{[i-n_c+1,i]}$ and $\mathbf{V}_{[i-n_c+1,i]}$ denote the keys and values over positions from $i-n_c+1$ to i.[11] This model can be seen as a type of local attention model.

- 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

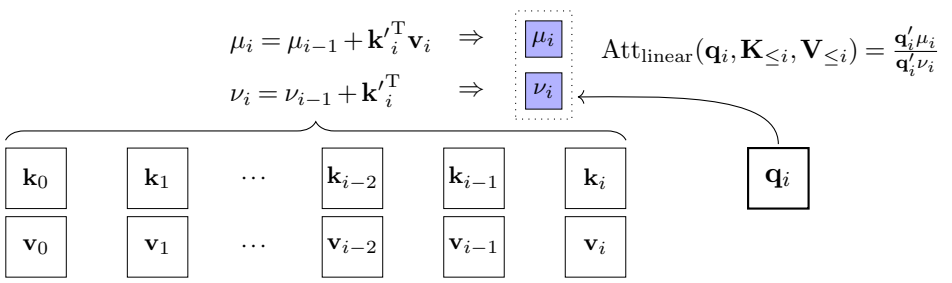
[11] More formally, we write $\mathbf{K}_{[i-n_c+1,i]} = \begin{bmatrix} \mathbf{k}_{i-n_c+1} \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$ and $\mathbf{V}_{[i-n_c+1,i]} = \begin{bmatrix} \mathbf{v}_{i-n_c+1} \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$. Sometimes we denote $\mathbf{K}_{[i-n_c+1,i]}$ by $\{\mathbf{k}_{i-n_c+1},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



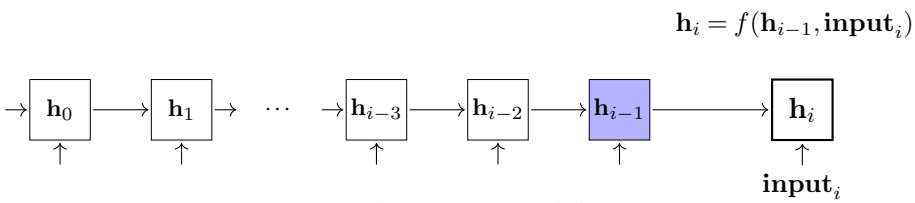
(a) Standard Self-attention



(b) Sparse Attention



(c) Linear Attention



(d) Recurrent Models

Figure 8.5: 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.

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

$$\mathrm{Mem} = \left(\frac{\sum_{j=i-n_c+1}^{i} \mathbf{k}_j}{n_c}, \frac{\sum_{j=i-n_c+1}^{i} \mathbf{v}_j}{n_c} \right) \tag{8.54}$$

Alternatively, we can use a weighted version of moving average

$$\text{Mem} = \Big(\frac{\sum_{j=i-n_c+1}^{i}\beta_{j-i+n_c}\mathbf{k}_j}{\sum_{j=1}^{n_c}\beta_j}, \frac{\sum_{j=i-n_c+1}^{i}\beta_{j-i+n_c}\mathbf{v}_j}{\sum_{j=1}^{n_c}\beta_j}\Big) \tag{8.55}$$

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 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

$$\text{Mem} = \Big(\frac{\sum_{j=0}^{i}\mathbf{k}_j}{i+1}, \frac{\sum_{j=0}^{i}\mathbf{v}_j}{i+1}\Big) \tag{8.56}$$

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

$$\text{Mem}_i = \frac{(\mathbf{k}_i, \mathbf{v}_i) + i \cdot \text{Mem}_{i-1}}{i+1} \tag{8.57}$$

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

$$\text{Mem} = \text{Update}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{8.58}$$

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. (8.58) can be expressed as an RNN-like model

$$\text{Mem} = f((\mathbf{k}_i, \mathbf{v}_i), \text{Mem}_{\text{pre}}) \tag{8.59}$$

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

$$\text{Mem} = \text{FIFO}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{8.60}$$

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

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

$$\text{Att}_{\text{com}}(\mathbf{q}_i, \text{Mem}, \text{CMem}) = \text{Att}_{\text{qkv}}(\mathbf{q}_i, [\text{Mem}, \text{CMem}]) \tag{8.61}$$

where $[\text{Mem}, \text{CMem}]$ is a combined memory of Mem and CMem. As with other segment-level models, the compressive Transformer model operates on segments of the sequence. Each segment is a sequence of n_s consecutive tokens, and we denote S^k_{kv} 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^k_{kv} to Mem, and then pop the n_s oldest key-value pairs from Mem, which is given by

$$\text{Mem} = \text{FIFO}(S^k_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{8.62}$$

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

$$\text{CMem} = \text{FIFO}(C^k_{\text{kv}}, \text{CMem}_{\text{pre}}) \tag{8.63}$$

where C^k_{kv} 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 8.6 shows illustrations of the above approaches. Note that, while we focus on optimization of the KV cache here, this issue is closely related to those discussed in the previous section. All of the methods we have mentioned so far can broadly be categorized as efficient attention approaches, which are widely used in various Transformer variants.

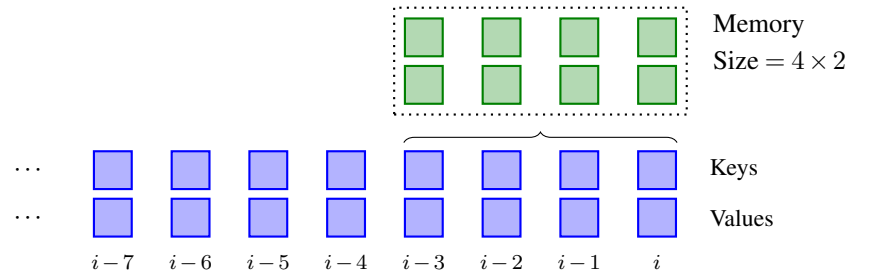
2. Memory-based Models

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.

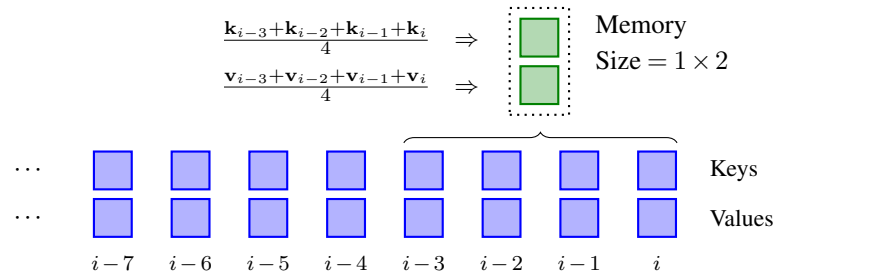
Here, we consider the k**-nearest neighbors** (k**-NN**) method which is one of the most popular memory-based methods. Since our focus is language modeling in this section, we define a sample in the datastore as a key-value pair corresponding to some context state. Note that "context" is a broad concept 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

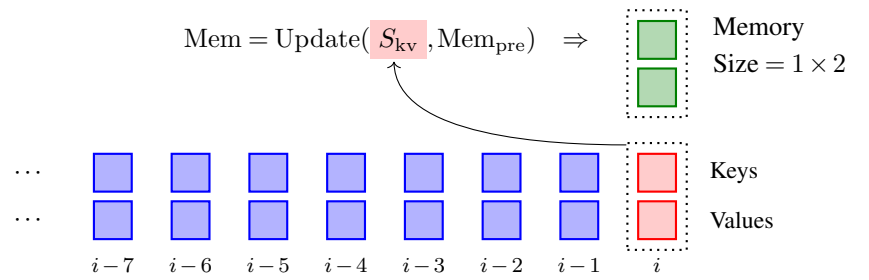
[12] A vector database, or vector store, is a database that provides highly optimized retrieval interfaces for finding stored vectors that closely match a query vector.



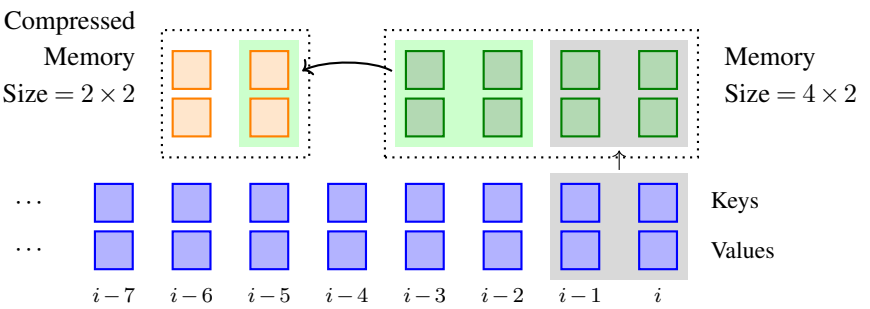
(a) Window-based Cache



(b) Moving Average-based Cache



(c) Recurrent Network as Cache



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

Figure 8.6: 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.

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

$$\begin{aligned} \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}, \mathrm{Mem}_{k\mathrm{nn}}) &= \mathbf{g} \odot \mathrm{Att}_{\mathrm{local}} + (1 - \mathbf{g}) \odot \mathrm{Att}_{k\mathrm{nn}} && (8.64) \\ \mathrm{Att}_{\mathrm{local}} &= \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}) && (8.65) \\ \mathrm{Att}_{k\mathrm{nn}} &= \mathrm{Att}(\mathbf{q}_i, \mathrm{Mem}_{k\mathrm{nn}}) && (8.66) \end{aligned}$$

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 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,

$$\Pr_{k\mathrm{nn}}(\cdot|\mathbf{h}_i) = \mathrm{Softmax}(\begin{bmatrix} -d_0 & \cdots & -d_{|V|} \end{bmatrix}) \quad (8.67)$$

where d_v equals the distance between $\mathbf{h}_i$ and $\mathbf{z}_j$ if w_j equals the v-th entry of V, and equals 0 otherwise. We use a linear function with a coefficient λ that interpolates between the

retrieval-based distribution $\Pr_{knn}(\cdot|\mathbf{h}_i)$ and the LLM output distribution $\Pr_{\text{lm}}(\cdot|\mathbf{h}_i)$

$$\Pr(\cdot|\mathbf{h}_i) \quad = \quad \lambda \cdot \Pr_{knn}(\cdot|\mathbf{h}_i) + (1-\lambda) \cdot \Pr_{\text{lm}}(\cdot|\mathbf{h}_i) \tag{8.68}$$

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

As with information retrieval (IR) systems, the datastore can also manage texts and provide access to relevant texts for a query. For example, we can store a collection of text documents in a search engine with full-text indexing, and then search it for documents that match a given text-based query. Applying IR techniques to LLMs leads to a general framework called **retrieval-augmented generation** (**RAG**). The RAG framework works as follows. We use the context $\mathbf{x}$ as the query and find the k most relevant document pieces $\{\mathbf{c}_1, ..., \mathbf{c}_k\}$ from the datastore via efficient IR techniques[13]. These search results are combined with the original context via a prompting template $g(\cdot)$[14], resulting in an augmented input for the LLM

$$\mathbf{x}' \quad = \quad g(\mathbf{c}_1, ..., \mathbf{c}_k, \mathbf{x}) \tag{8.69}$$

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 8.7 shows a comparison of the use of different external memories in LLMs.

3. Memory Capacity

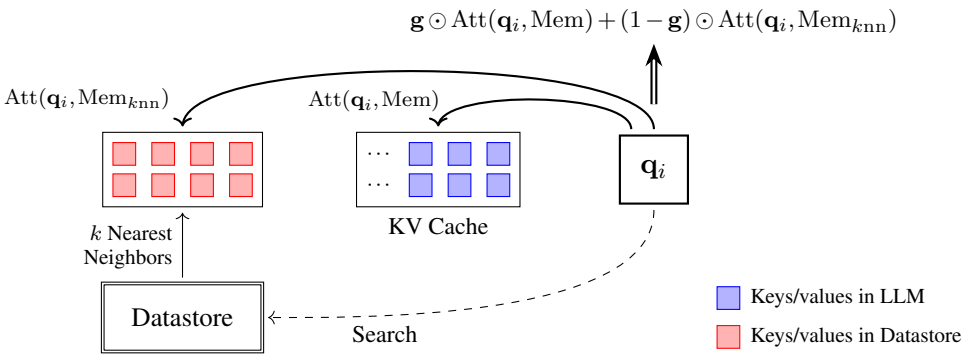
A memory model in LLMs, in the form of a simple key-value cache or a datastore, can broadly be seen as an encoder of contextual information. Ideally, before we say that a memory model is representative of the entire context in token prediction, we need to make sure that the model can accurately represent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

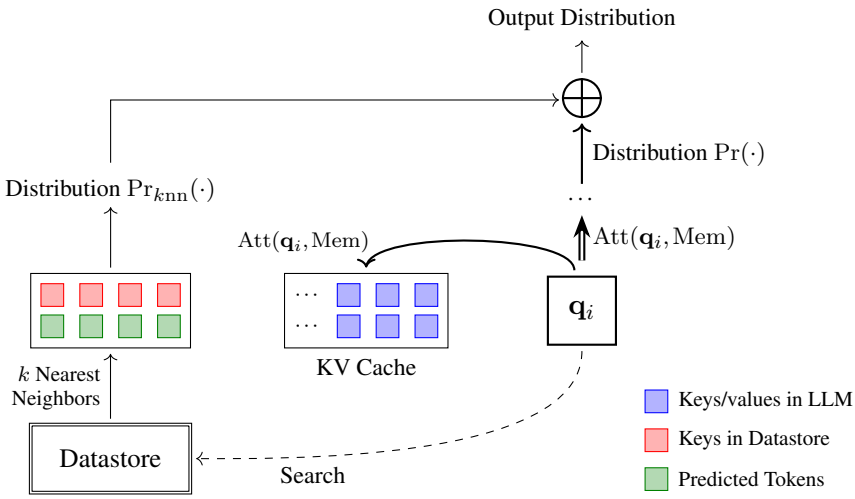
[13] In piratical applications, queries are typically generated using a query generation system, which may expand it with variations of tokens and query intent.

[14] For example, the template could be:

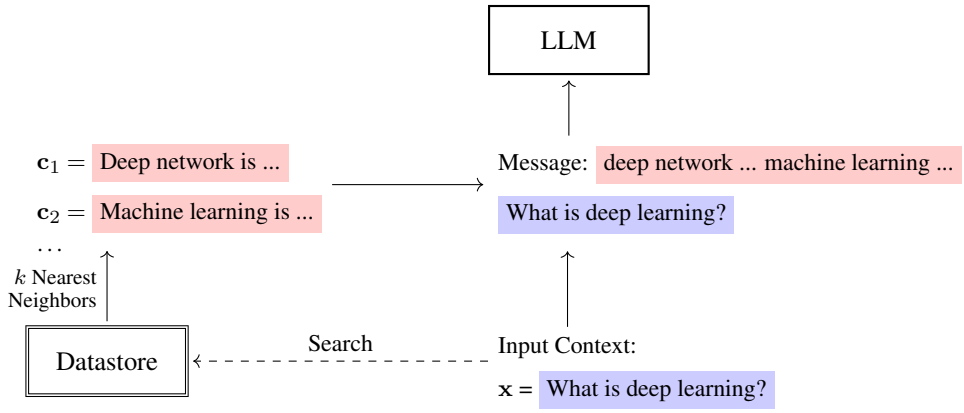
message = {*$\mathbf{c}_1$*} ... {*$\mathbf{c}_k$*}
input: {*$\mathbf{x}$*}
output: ____



(a) k-NN Search Augmented Attention



(b) k-NN Language Modeling



(c) Retrieval-augmented Generation

Figure 8.7: Illustrations of external memories (or datastores) for language modeling.

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

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

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

8.3.4 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 8.1.1 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

$$\text{Output} \quad = \quad \text{Merge}(\text{head}_1, ..., \text{head}_\tau)\mathbf{W}^{\text{head}} \tag{8.70}$$

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

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[j]}, \mathbf{V}_{\leq i}^{[j]}) \tag{8.71}$$

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 8.8 (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

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \tag{8.72}$$

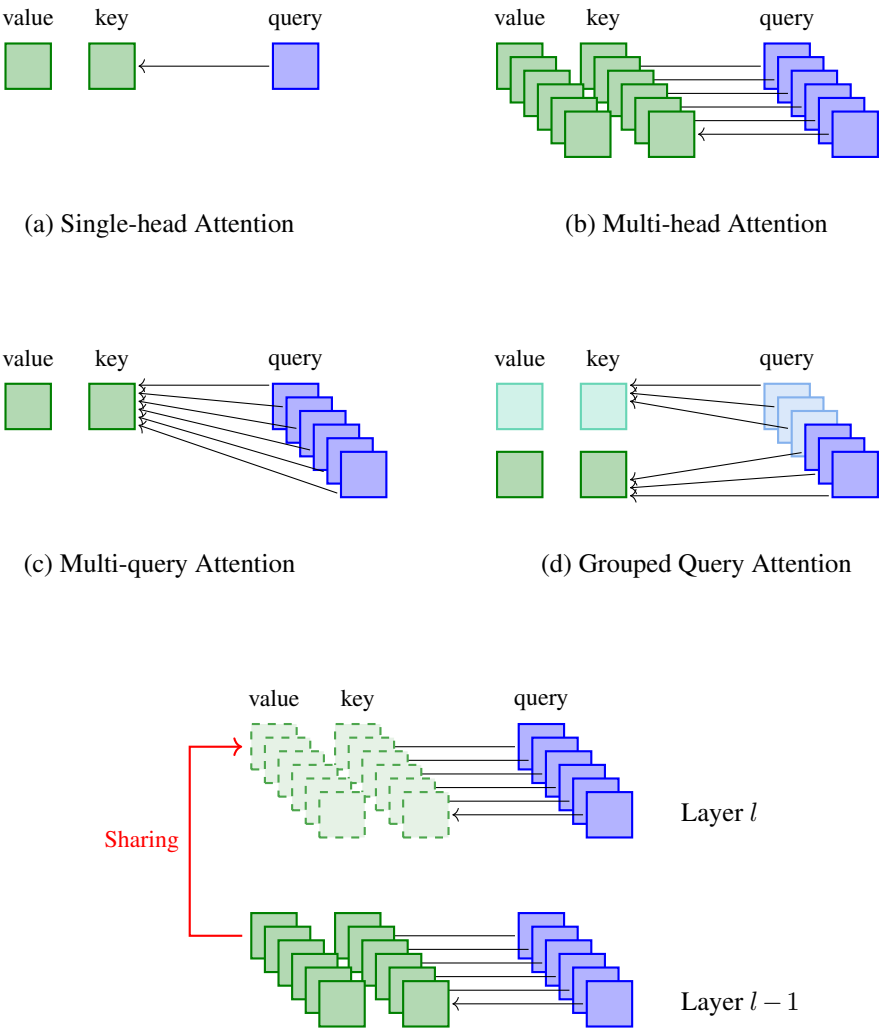
Figure 8.8 (c) illustrates this model. By sharing keys and values, the size of the KV cache would be $O(L \cdot d_h \cdot m)$.

Grouped query attention (**GQA**) is a natural extension to multi-head attention and MQA [Ainslie et al., 2023]. In GQA, heads are divided into n_g groups, each corresponding to a shared set of keys and values. Hence we have n_g sets of keys and values $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), ..., (\mathbf{K}_{\leq i}^{[n_g]}, \mathbf{V}_{\leq i}^{[n_g]})\}$. See Figure 8.8 (d) for an illustration. Let $g(j)$ be the group id for the j-th head. The GQA model can be expressed as

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[g(j)]}, \mathbf{V}_{\leq i}^{[g(j)]}) \tag{8.73}$$

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 8.8 (e) shows an illustration of this method, where a query in a layer directly accesses the KV cache of a lower-level layer.



(a) Single-head Attention

(b) Multi-head Attention

(c) Multi-query Attention

(d) Grouped Query Attention

(e) Cross-layer Multi-head Attention

Figure 8.8: Illustration of QKV attention based on different multi-head and sharing mechanisms. (a) = single-head attention, and (b-e) = attention with multiple heads.

8.3.5 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

$$\mathbf{e}_i \quad = \quad \mathbf{x}_i + \mathrm{PE}(i) \tag{8.74}$$

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 8.9 (a) for an illustration where the learned embedding model cannot model data points outside the training domain if it lacks the ability to extrapolate.

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

- **Extrapolation**. The model learned on observed data points (i.e., positions) can be directly employed to assign meaningful values to data points beyond the original range. For example, suppose we have a series of numbers $1, 2, ..., 10$, and we want to understand the meaning of a new number, 15. Knowing that these numbers are natural numbers used for ordering, we can easily infer that 15 is a number that follows 10, even though 15 has not been observed before. Figure 8.9 (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 8.9 (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.

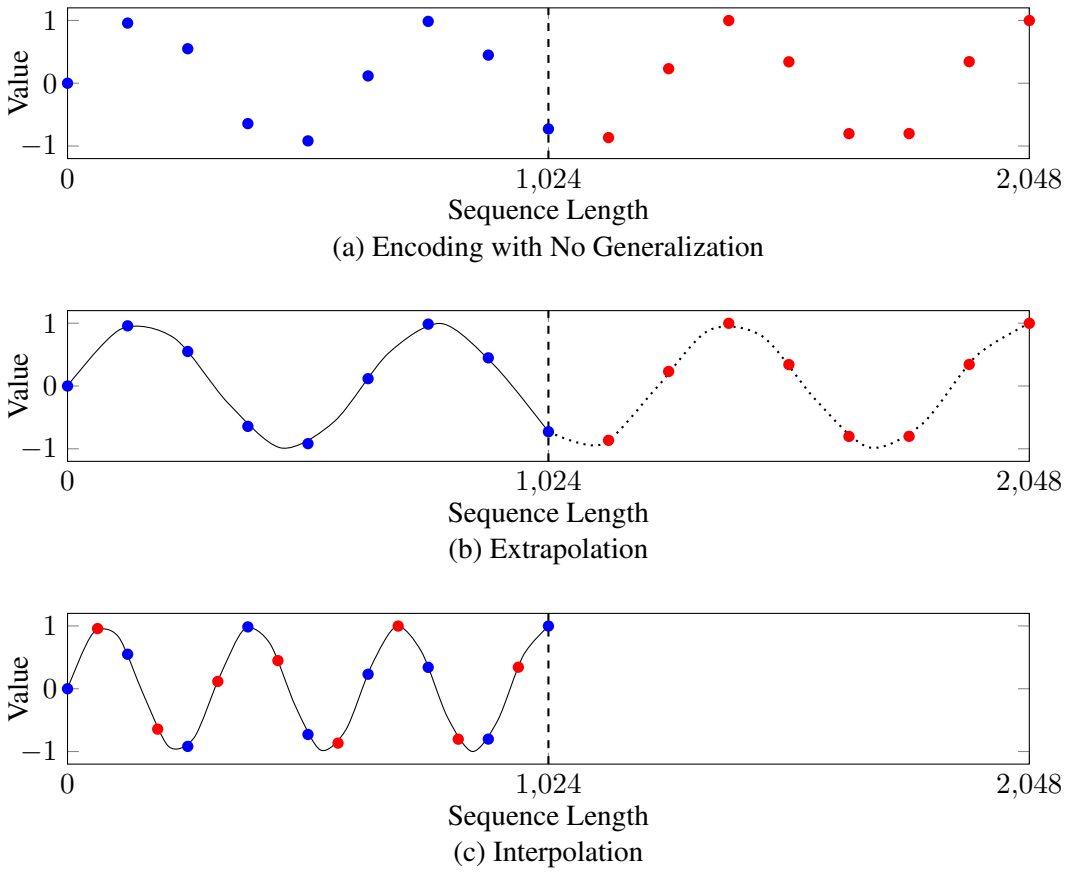


Figure 8.9: Illustrations of different positional embedding methods for a range of positions. Blue points represent the positions that have been observed during training, and red points represent the positions that are newly observed at test time. In sub-figure (a), the encoding model only memorizes the points seen during training, and cannot generalize. In sub-figures (b) and (c), the model can generalize through extrapolation and interpolation.

1. Attention with Learnable Biases

One problem with Eq. (8.74) is that the embedding model treats each token independently and therefore ignores the distance between different tokens. A common improvement to this model, called relative positional embedding, is to consider the pairwise relationship between tokens [Shaw et al., 2018]. The general idea behind this is to obtain the offset between any pair of positions and incorporate it into the self-attention model. One of the simplest forms of self-attention with relative positional embedding is given by

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j=0}^{i} \alpha(i,j)\mathbf{v}_j \tag{8.75}$$

$$\alpha(i,j) = \text{Softmax}(\frac{\mathbf{q}_i \mathbf{k}_j^{\text{T}} + \text{PE}(i,j)}{\sqrt{d}} + \text{Mask}(i,j)) \tag{8.76}$$

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]

$$d(i,j) \quad = \quad i-j \tag{8.77}$$

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

$$b(i-j) \quad = \quad \frac{n_b+1}{2} + \lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\mathrm{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \rfloor \tag{8.78}$$

  where the parameter $\mathrm{dist}_{\max}$ is typically set to a relatively large number to indicate the maximum offset we may encounter.
- When $i-j > \mathrm{dist}_{\max}$, 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

$$\begin{aligned} & b(i-j) \\ = & \begin{cases} i-j & 0 \leq i-j < \frac{n_b+1}{2} \\ \min(n_b, \frac{n_b+1}{2} + \lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\mathrm{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \rfloor) & i-j \geq \frac{n_b+1}{2} \end{cases} \end{aligned} \tag{8.79}$$

Figure 8.10 shows an illustration of these buckets. We see that in the first half of 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 $i < j$.

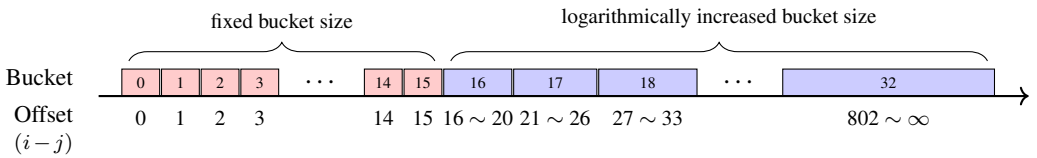


Figure 8.10: Illustration of distributing query-key offsets into buckets in the T5 model ($n_b = 32$ and $\text{dist}_{\max} = 1024$). Boxes represent buckets. In the first half of the buckets, we use a fixed bucket size. In the second half of the buckets, we increase the bucket size logarithmically. The last bucket contains all the query-key offsets that are not covered by previous buckets.

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. (8.76), the attention weight for $\mathbf{q}_i$ and $\mathbf{k}_j$ becomes[16]

$$\alpha(i,j) \quad = \quad \text{Softmax}(\frac{\mathbf{q}_i \mathbf{k}_j^{\mathrm{T}} + u_{b(i-j)}}{\sqrt{d}} + \text{Mask}(i,j)) \tag{8.81}$$

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.

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

[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

$$\alpha(i,j) \quad = \quad \text{Softmax}(\mathbf{q}_i \mathbf{k}_j^{\mathrm{T}} + u_{b(i-j)} + \text{Mask}(i,j)) \tag{8.80}$$

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]	$f\left(\psi(\ i-j\ )/\psi(\max(m_{\text{len}}, i))\right)$

Table 8.4: 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.

scaled query-key offset

$$\begin{aligned} \mathrm{PE}(i,j) &= -\beta \cdot (i-j) \\ &= \beta \cdot (j-i) \end{aligned} \tag{8.82}$$

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

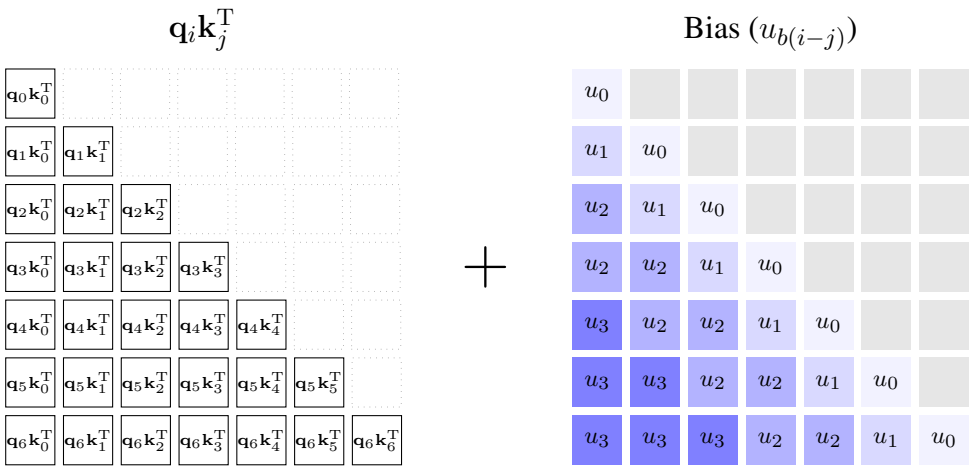
$$\alpha(i,j) = \mathrm{Softmax}\left(\frac{\mathbf{q}_i \mathbf{k}_j^{\mathrm{T}} + \beta \cdot (j-i)}{\sqrt{d}} + \mathrm{Mask}(i,j)\right) \tag{8.83}$$

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 8.11 for a comparison of the T5 bias and the ALiBi bias.

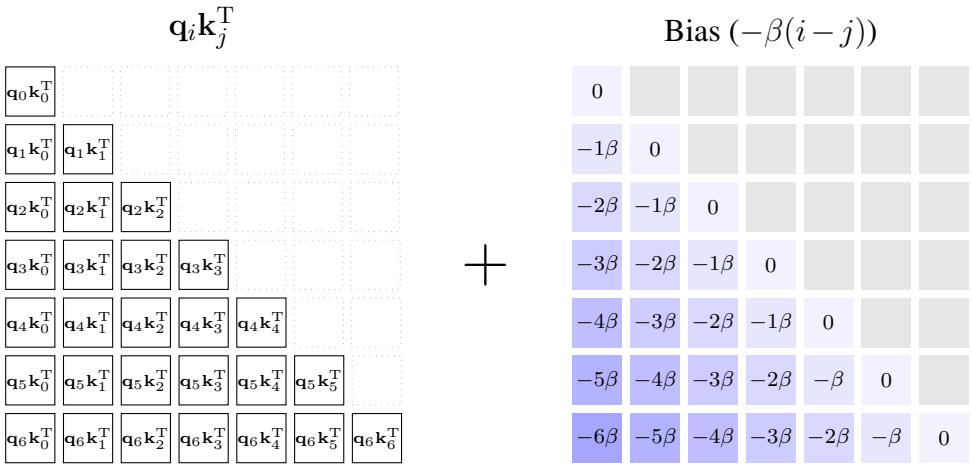
In general, the scalar β should be tuned on a validation dataset. However, Press et al. [2022] found that setting β to values decreasing geometrically by a factor of $\frac{1}{2^a}$ for multi-head attention performs well on a variety of tasks. Specifically, for a self-attention sub-layer involving n_{head}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

$$\beta_k = \frac{1}{2^{\frac{8}{k}}} \tag{8.84}$$

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 8.4 shows a comparison of such biases. As an aside it is worth noting that the form of the right-hand side of Eq. (8.82) 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].



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



(b) The ALiBi bias

Figure 8.11: 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.

3. Rotary Positional Embedding

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

of multiplicative embeddings

$$\mathbf{e}_i = \mathbf{x}_i R(i) \tag{8.85}$$

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

For simplicity, we will first consider embeddings with only two dimensions and return to a discussion of the more general formulation later. Suppose we have a 2-dimensional token embedding $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$. We can represent it as a vector in a plane, originating at the origin $(0,0)$ and terminating at (x_1, x_2). A counterclockwise rotation of this vector refers to an operation of moving the vector around the origin while maintaining its magnitude, as shown in Figure 8.12 (a). The degree of rotation is usually defined by a specific angle, denoted by θ. The rotation can be expressed mathematically in the form

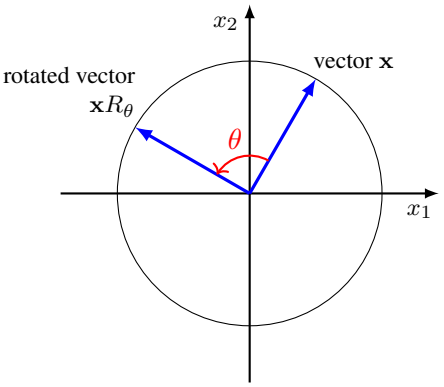
$$\begin{aligned} \mathrm{Ro}(\mathbf{x}, \theta) &= \mathbf{x} R_\theta \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta \cdot x_1 - \sin\theta \cdot x_2 & \sin\theta \cdot x_1 + \cos\theta \cdot x_2 \end{bmatrix} \end{aligned} \tag{8.86}$$

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

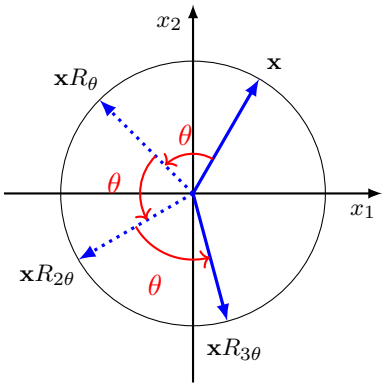
$$\begin{aligned} \mathrm{Ro}(\mathbf{x}, t\theta) &= \mathbf{x} R_{t\theta} \\ &= \begin{bmatrix} \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 & \sin t\theta \cdot x_1 + \cos t\theta \cdot x_2 \end{bmatrix} \end{aligned} \tag{8.87}$$

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 8.12 (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.

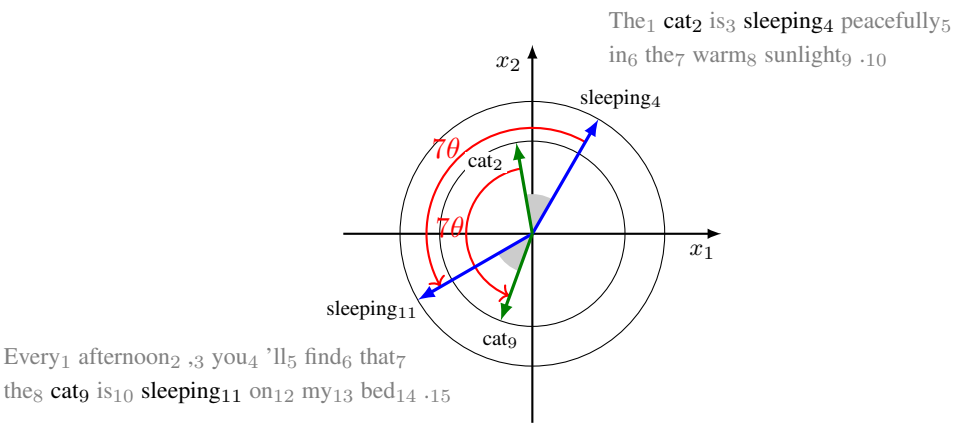
A popular way to understand vector rotation is to define it in complex spaces. It is easy to transform each vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ in the 2D Euclidean space $\mathbb{R}^2$ to a complex number $\mathbf{x}' = x_1 + \mathtt{i} x_2$ in the complex space $\mathbb{C}$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



(a) Single-step Rotation



(b) Multi-step Rotation



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

Figure 8.12: 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.

rotation operation can be re-expressed in the form

$$\begin{aligned} \mathbf{x}R_{t\theta} &\mapsto \mathbf{x}'e^{\mathtt{i}t\theta} \\ &= (x_1 + \mathtt{i}x_2)(\cos t\theta + \mathtt{i}\sin t\theta) \\ &= \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 + \mathtt{i}(\sin t\theta \cdot x_1 + \cos t\theta \cdot x_2) \end{aligned} \tag{8.88}$$

Here we denote the token representation $\mathbf{x}'e^{\mathtt{i}t\theta}$ by $C(\mathbf{x}, t\theta)$. The inner product of the represen-

tations of the tokens at positions t and s can be written as

$$\langle C(\mathbf{x}, t\theta), C(\mathbf{y}, s\theta)\rangle = (\mathbf{x}'\overline{\mathbf{y}'})e^{\mathbf{i}(t-s)\theta} \tag{8.89}$$

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 $(t-s)\theta$

$$\begin{aligned}\mathrm{Ro}(\mathbf{x}, t\theta)[\mathrm{Ro}(\mathbf{y}, s\theta)]^{\mathrm{T}} &= \mathbf{x}R_{t\theta}[\mathbf{y}R_{s\theta}]^{\mathrm{T}} \\ &= \mathbf{x}R_{t\theta}[R_{s\theta}]^{\mathrm{T}}\mathbf{y}^{\mathrm{T}} \\ &= \mathbf{x}R_{(t-s)\theta}\mathbf{y}^{\mathrm{T}}\end{aligned} \tag{8.90}$$

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 $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_d \end{bmatrix}$, we can treat it as a $\frac{d}{2}$-dimensional complex vector $\mathbf{x}' = \begin{bmatrix} x'_1 & x'_2 & \ldots & x'_{d/2} \end{bmatrix} = \begin{bmatrix} x_1 + \mathbf{i}x_2 & x_3 + \mathbf{i}x_4 & \ldots & x_{d-1} + \mathbf{i}x_d \end{bmatrix}$, where each consecutive pair of items forms a complex number. Then, the rotary positional embedding in the complex space is given by

$$C(\mathbf{x}, t\theta) = \sum_{k=1}^{d/2} x'_k e^{\mathbf{i}t\theta_k}\vec{e}_k \tag{8.91}$$

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

$$\mathrm{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 & x_2 & \ldots & x_d \end{bmatrix} \begin{bmatrix} R_{t\theta_1} & & & \\ & R_{t\theta_2} & & \\ & & \ddots & \\ & & & R_{t\theta_{d/2}} \end{bmatrix} \tag{8.92}$$

where $R_{t\theta_k} = \begin{bmatrix} \cos t\theta_k & \sin t\theta_k \\ -\sin t\theta_k & \cos t\theta_k \end{bmatrix}$. $\theta = \begin{bmatrix} \theta_1, \ldots, \theta_{d/2} \end{bmatrix}$ 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. (8.92) can be rewritten into a form that relies solely on

the element-wise product and addition of vectors.

$$\mathrm{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d-1} \\ x_d \end{bmatrix}^{\mathrm{T}} \odot \begin{bmatrix} \cos t\theta_1 \\ \cos t\theta_1 \\ \vdots \\ \cos t\theta_{d/2} \\ \cos t\theta_{d/2} \end{bmatrix}^{\mathrm{T}} + \begin{bmatrix} -x_2 \\ x_1 \\ \vdots \\ -x_d \\ x_{d-1} \end{bmatrix}^{\mathrm{T}} \odot \begin{bmatrix} \sin t\theta_1 \\ \sin t\theta_1 \\ \vdots \\ \sin t\theta_{d/2} \\ \sin t\theta_{d/2} \end{bmatrix}^{\mathrm{T}} \tag{8.93}$$

Finally, we rewrite Eq. (8.85) to obtain the form of the embedding at position i

$$\mathbf{e}_i = \mathrm{Ro}(\mathbf{x}_i, i\theta) \tag{8.94}$$

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 $\theta = \left[\theta_1, ..., \theta_{d/2}\right]$ 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. (8.93))

$$\cos i\theta = \left[\cos i\theta_1 \quad ... \quad \cos i\theta_{d/2}\right] \tag{8.95}$$

$$\sin i\theta = \left[\sin i\theta_1 \quad ... \quad \sin i\theta_{d/2}\right] \tag{8.96}$$

θ_k is a exponential function of k and takes the form

$$\theta_k = b^{-\frac{2(k-1)}{d}} \tag{8.97}$$

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

$$T_k = 2\pi \cdot b^{\frac{2(k-1)}{d}} \tag{8.98}$$

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

$$T'_k = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(k-1)}{d}} \tag{8.99}$$

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., 2023b]. The new model with

linear positional interpolation is given by

$$\text{Ro}'(\mathbf{x}_i, i\theta) \quad = \quad \text{Ro}(\mathbf{x}_i, \frac{m_l}{m} i\theta) \tag{8.100}$$

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

$$2\pi \cdot (\lambda b)^{\frac{2(\frac{d}{2}-1)}{d}} \quad = \quad \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(\frac{d}{2}-1)}{d}} \tag{8.101}$$

Solving this equation, we obtain

$$\begin{aligned} \lambda \quad &= \quad (\frac{m}{m_l})^{\frac{d}{2(\frac{d}{2}-1)}} \\ &= \quad (\frac{m}{m_l})^{\frac{d}{d-2}} \end{aligned} \tag{8.102}$$

This gives an embedding model

$$\text{Ro}'(\mathbf{x}_i, i\theta) \quad = \quad \text{Ro}(\mathbf{x}_i, i\theta') \tag{8.103}$$

where

$$\theta' = \left[(\lambda b)^{-\frac{0}{d}}, (\lambda b)^{-\frac{2}{d}}, ..., (\lambda b)^{-\frac{d-2}{d}}\right] \tag{8.104}$$

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].

8.3.6 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.

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 8.3.3, 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

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

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. 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.

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., 2023b], 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., 2024b]. 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

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

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.

8.4 Summary

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

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

Given their amazing power, LLMs have attracted considerable interest, both in terms of techniques and applications. As a result, the explosion of research interest in LLMs has also led to a vast number of new techniques and models. However, we do not attempt to provide a comprehensive literature review on all aspects of LLMs, given the rapid evolution of the field. Nevertheless, one can still gain knowledge about LLMs from general reviews [Zhao et al., 2023; Minaee et al., 2024] or more focused discussions on specific topics [Ruan et al., 2024].

Bibliography

[Ainslie et al., 2020] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. Etc: Encoding long and structured inputs in transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 268–284, 2020.

[Ainslie et al., 2023] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, 2023.

[Alabdulmohsin et al., 2022] Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting neural scaling laws in language and vision. *Advances in Neural Information Processing Systems*, 35:22300–22312, 2022.

[Almazrouei et al., 2023] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[Arjovsky et al., 2016] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International conference on machine learning*, pages 1120–1128, 2016.

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

[Biderman et al., 2021] Stella Biderman, Sid Black, Charles Foster, Leo Gao, Eric Hallahan, Horace He, Ben Wang, and Phil Wang. Rotary embeddings: A relative revolution. https://blog.eleuther.ai/rotary-embeddings/, 2021.

[Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Brandon et al., 2024] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*, 2024.

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

[Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke,

Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[Bulatov et al., 2022] Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. *Advances in Neural Information Processing Systems*, 35:11079–11091, 2022.

[Caballero et al., 2023] Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken neural scaling laws. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2023.

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

[Chen et al., 2023] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023b.

[Chi et al., 2022] Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. Kerple: Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems*, 35:8386–8399, 2022.

[Chi et al., 2023] Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. Dissecting transformer length extrapolation via the lens of receptive field analysis. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13522–13537, 2023.

[Chowdhery et al., 2022] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

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

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

[Dai et al., 2019] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan

Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

[Dao et al., 2022] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[Deletang et al., 2024] Gregoire Deletang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In *The Twelfth International Conference on Learning Representations*, 2024.

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

[Ding et al., 2024] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[Dong et al., 2021] Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In *International Conference on Machine Learning*, pages 2793–2803. PMLR, 2021.

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

[Fedus et al., 2022] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[Gemma Team, 2024] Google DeepMind Gemma Team. Gemma: Open Models Based on Gemini Research and Technology, 2024.

[Gordon et al., 2021] Mitchell A Gordon, Kevin Duh, and Jared Kaplan. Data and parameter scaling laws for neural machine translation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5915–5922, 2021.

[Gu and Dao, 2023] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[Gupta and Berant, 2020] Ankit Gupta and Jonathan Berant. Gmat: Global memory augmentation for transformers. *arXiv preprint arXiv:2006.03274*, 2020.

[Gupta et al., 2021] Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, pages 39–52, 2021.

[Harlap et al., 2018] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training.

arXiv preprint arXiv:1806.03377, 2018.

[Hendrycks and Gimpel, 2016] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[Hestness et al., 2017] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

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

[Hoffmann et al., 2022] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[Huang et al., 2019] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[Hutchins et al., 2022] DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. *Advances in neural information processing systems*, 35: 33248–33261, 2022.

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

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

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

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

[Katharopoulos et al., 2020] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[Khandelwal et al., 2020] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020.

[Kim et al., 2023] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*, 2023.

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

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

[Korthikanti et al., 2023] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[Kwon et al., 2023] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[Lan et al., 2020] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *Proceedings of International Conference on Learning Representations*, 2020.

[Li et al., 2024] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. In *The Twelfth International Conference on Learning Representations*, 2024.

[Li et al., 2023] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

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

[Liu et al., 2024] Xinyu Liu, Runsong Zhao, Pengcheng Huang, Chunyang Xiao, Bei Li, Jingang Wang, Tong Xiao, and Jingbo Zhu. Forgetting curve: A reliable method for evaluating memorization capability for long-context models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4667–4682, 2024b.

[Ma et al., 2023] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. Mega: Moving average equipped gated attention. In *The Eleventh International Conference on Learning Representations*, 2023.

[Ma et al., 2024] Xuezhe Ma, Xiaomeng Yang, Wenhan Xiong, Beidi Chen, Lili Yu, Hao Zhang, Jonathan May, Luke Zettlemoyer, Omer Levy, and Chunting Zhou. Megalodon: Efficient llm pretraining and inference with unlimited context length. *arXiv preprint arXiv:2404.08801*, 2024.

[Martins et al., 2022] Pedro Henrique Martins, Zita Marinho, and André FT Martins. ∞-former: Infinite memory transformer-former: Infinite memory transformer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5468–5485, 2022.

[Micikevicius et al., 2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *Proceedings of International Conference on Learning Representations*, 2018.

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

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

[Minaee et al., 2024] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

[Mohtashami and Jaggi, 2024] Amirkeivan Mohtashami and Martin Jaggi. Random-access infinite context length for transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

[Munkhdalai et al., 2024] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.

[Narayanan et al., 2021] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[OpenAI, 2024] OpenAI. Learning to reason with llms, September 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

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

[Pal et al., 2023] Koyena Pal, Jiuding Sun, Andrew Yuan, Byron C Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, pages 548–560, 2023.

[Parmar et al., 2018] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pages 4055–4064. PMLR, 2018.

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

[Peng et al., 2024] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

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

Human Language Technologies, Volume 1 (Long Papers), 2018.

[Press et al., 2022] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *Proceedings of International Conference on Learning Representations*, 2022.

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

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

[Rae et al., 2019] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2019.

[Raffel et al., 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[Ramachandran et al., 2017] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[Rosenfeld et al., 2020] Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. In *Proceedings of International Conference on Learning Representations*, 2020.

[Ruan et al., 2024] Junhao Ruan, Long Meng, Weiqiao Shan, Tong Xiao, and Jingbo Zhu. A survey of llm surveys. https://github.com/NiuTrans/ABigSurveyOfLLMs, 2024.

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

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

[Shaw et al., 2018] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.

[Shazeer, 2019] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[Shazeer, 2020] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[Stiennon et al., 2020] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human

feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[Su et al., 2024] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

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

[Tay et al., 2020] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020.

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

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

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

[Wang et al., 2019] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, 2019.

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

[Wei et al., 2022] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022b.

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

[Weng, 2021] Lilian Weng. How to train really large models on many gpus? *lil-*

ianweng.github.io, Sep 2021. URL https://lilianweng.github.io/posts/2021-09-25-train-large/.

[Wu et al., 2024] Wilson Wu, John X Morris, and Lionel Levine. Do language models plan for future tokens? *arXiv preprint arXiv:2404.00859*, 2024.

[Wu et al., 2021] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *Proceedings of International Conference on Learning Representations*, 2021.

[Xiao et al., 2024] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *Proceedings of The Twelfth International Conference on Learning Representations*, 2024.

[Xiao et al., 2019] Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. Sharing attention weights for fast transformer. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*, pages 5292–5298, 2019.

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

[Zaheer et al., 2020] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, C. Alberti, S. Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, L. Yang, and A. Ahmed. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33: 17283–17297, 2020.

[Zhang and Sennrich, 2019] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[Zhao et al., 2023] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie, and Ji rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

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