Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

June 12, 2025

Tong Xiao and Jingbo Zhu
June, 2025

Chapter 11

Inference

Once we have pre-trained and fine-tuned an LLM, we can apply it to make predictions on new data. This process is called inference, in which the LLM computes the probabilities of different possible outputs given an input, and selects the output that maximizes the probability. The inference problem is generally expressed in the following form:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{x}) \tag{11.1}$$

where $\mathbf{x}$ is the input sequence, $\mathbf{y}$ is a possible output sequence, and $\hat{\mathbf{y}}$ is the best output sequence.

This is perhaps one of the most widely adopted formulas in NLP, and dates back to the early days of speech recognition and machine translation systems based on probabilistic models. Although for some applications, such as predicting a token using a very small language model, solving this optimization problem seems trivial, for most situations the computational challenges arise from both calculating $\Pr(\mathbf{y}|\mathbf{x})$ and performing the $\arg\max$ operation. The problems we therefore wish to address in this chapter involve: 1) computing the prediction probability efficiently given a trained LLM, and 2) devising an efficient (suboptimal) search for $\hat{\mathbf{y}}$.

At a high level, these are fundamental issues in artificial intelligence, which have been extensively studied. So many well-established techniques can be directly applied, for example, one can use greedy search algorithms to implement an efficient inference system. On the other hand, model-specific optimizations, such as efficient attention models for Transformers, can be considered to further improve efficiency. But, in many practical applications, we still need to make a trade-off between accuracy and efficiency, by carefully combining various techniques.

The importance of the inference problem in LLMs also lies in the fact that many application scenarios require processing extremely long sequences. Recent studies have found that injecting additional prompts and contextual information, such as long chain-of-thought prompts, during inference can significantly improve the performance of LLMs. This provides a new approach to scaling LLMs: better results can be achieved by increasing the compute at inference time. For instance, through inference-time scaling, OpenAI [2024]'s o1 and Deepseek [2025]'s R1

systems have demonstrated impressive performance on complex reasoning and programming tasks. This, in turn, has encouraged the NLP field to focus more on the issue of efficient inference.

In this chapter, we will introduce basic concepts and algorithms of LLM inference, including prefilling-decoding frameworks, search (decoding) algorithms, and evaluation metrics of inference performance. We will then present methods for improving the efficiency of LLM inference, covering a range of techniques for speeding up the system and compressing the model. Finally, we will discuss inference-time scaling, which is considered an important application of inference optimization.

11.1 Prefilling and Decoding

In this section, we present the prefilling-decoding framework, which is the most commonly used for interpreting and implementing LLM inference processes. We first introduce the notation and background knowledge, and then describe the details of the framework, such as the decoding algorithms for LLM inference.

11.1.1 Preliminaries

Although we have described LLMs many times in this book, we begin by briefly defining the notation to facilitate the subsequent discussion, and to make this chapter self-contained.

- $\mathbf{x}$: The input token sequence. It is conceptually equivalent to a "prompt", which includes instructions, user inputs, and any additional context intended as input to the LLM. $\mathbf{x}$ comprises $m+1$ tokens, denoted by $x_0 ... x_m$, where x_0 is the start symbol $\langle \text{SOS} \rangle$.
- $\mathbf{y}$: The output token sequence, also called the response to the input. $\mathbf{y}$ comprises n tokens, denoted by $y_1 ... y_n$.
- $\mathbf{y}_{<i}$: The output tokens that precede position i, that is, $\mathbf{y}_{<i} = y_1 ... y_{i-1}$.
- $\Pr(\mathbf{y}|\mathbf{x})$: The probability of generating $\mathbf{y}$ given $\mathbf{x}$ using the LLM. If the LLM is parameterized by θ, we can write it as $\Pr_\theta(\mathbf{y}|\mathbf{x})$.
- $[\mathbf{x}, \mathbf{y}]$: The concatenated token sequence of $\mathbf{x}$ and $\mathbf{y}$. That is, $[\mathbf{x}, \mathbf{y}] = x_0 ... x_m y_1 ... y_n$. Occasionally, we use the notation $\text{seq}_{\mathbf{x},\mathbf{y}}$ to represent $[\mathbf{x}, \mathbf{y}]$.
- $\Pr([\mathbf{x}, \mathbf{y}])$: The probability of generating the token sequence $[\mathbf{x}, \mathbf{y}]$ using the LLM.

As described in Eq. (11.1), the goal of LLM inference is to maximize $\Pr(\mathbf{y}|\mathbf{x})$. Modeling this conditional probability is common in NLP. At first glance, it seems to be a sequence-to-sequence problem, where we transform a sequence into another using encoding-decoding models. However, we are not discussing sequence-to-sequence problems or encoding-decoding architectures. Instead, as discussed in earlier chapters, this modeling problem can be addressed

by using decoder-only models. To do this, we can interpret the log-scale probability $\log \Pr(\mathbf{y}|\mathbf{x})$ as the difference between $\log \Pr([\mathbf{x},\mathbf{y}])$ and $\log \Pr(\mathbf{x})$

$$\log \Pr(\mathbf{y}|\mathbf{x}) \quad = \quad \log \Pr([\mathbf{x},\mathbf{y}]) - \log \Pr(\mathbf{x}) \tag{11.2}$$

where $\log \Pr([\mathbf{x},\mathbf{y}])$ and $\log \Pr(\mathbf{x})$ can be obtained by running the LLM on the sequences $[\mathbf{x},\mathbf{y}]$ and $\mathbf{x}$, respectively. For example, we can calculate the probability of generating $\mathbf{x}$ using the chain rule

$$\begin{aligned}
\log \Pr(\mathbf{x}) &= \log \Pr(x_0 ... x_m) \\
&= \log \left[\Pr(x_0) \Pr(x_1|x_0) \cdots \Pr(x_m|x_0 ... x_{m-1}) \right] \\
&= \underbrace{\log \Pr(x_0)}_{=0} + \sum_{j=1}^{m} \log \Pr(x_j|\mathbf{x}_{<j}) \\
&= \sum_{j=1}^{m} \log \Pr(x_j|\mathbf{x}_{<j})
\end{aligned} \tag{11.3}$$

In other words, we calculate the token prediction log-probability at each position of $\mathbf{x}$, and sum all these log-probabilities.

In common implementations of LLMs, however, we do not need to compute the log-probability of the input sequence, but use the LLM to directly compute the log-probability of the output sequence in the following form

$$\log \Pr(\mathbf{y}|\mathbf{x}) \quad = \quad \sum_{i=1}^{n} \log \Pr(y_i|\mathbf{x},\mathbf{y}_{<i}) \tag{11.4}$$

where $[\mathbf{x},\mathbf{y}_{<i}]$ represents the context for predicting y_i. We use $\Pr(y_i|\mathbf{x},\mathbf{y}_{<i})$ to denote $\Pr(y_i|[\mathbf{x},\mathbf{y}_{<i}])$, following the commonly used notation in the literature.

Now, we have two sub-problems in addressing the inference issue described in Eq. (11.1):

- **Model Computation**: we model $\Pr(y_i|\mathbf{x},\mathbf{y}_{<i})$ and compute it in an efficient manner.
- **Search**: we find the optimal (or sub-optimal) output sequence in terms of $\log \Pr(\mathbf{y}|\mathbf{x})$.

The second sub-problem is a classic issue in NLP. We will show in Section 11.1.3 that there are several well-studied algorithms that can be applied to efficiently search the space of possible output sequences. The first sub-problem requires a language model to produce a distribution over a vocabulary V given a sequence of context tokens. We can do this by training a Transformer decoder, which outputs the distribution

$$\Pr(\cdot|\mathbf{x},\mathbf{y}_{<i}) \quad = \quad \text{Softmax}(\mathbf{H}\mathbf{W}^o)_{m+i} \tag{11.5}$$

$$\mathbf{H} \quad = \quad \text{Dec}([\mathbf{x},\mathbf{y}_{<i}]) \tag{11.6}$$

Here $\text{Dec}(\cdot)$ produces a sequence of representations, each corresponding to a position of the input sequence. So, if we input $[\mathbf{x},\mathbf{y}_{<i}]$ to the LLM, $\mathbf{H}$ is an $i' \times d$ matrix, where d is the

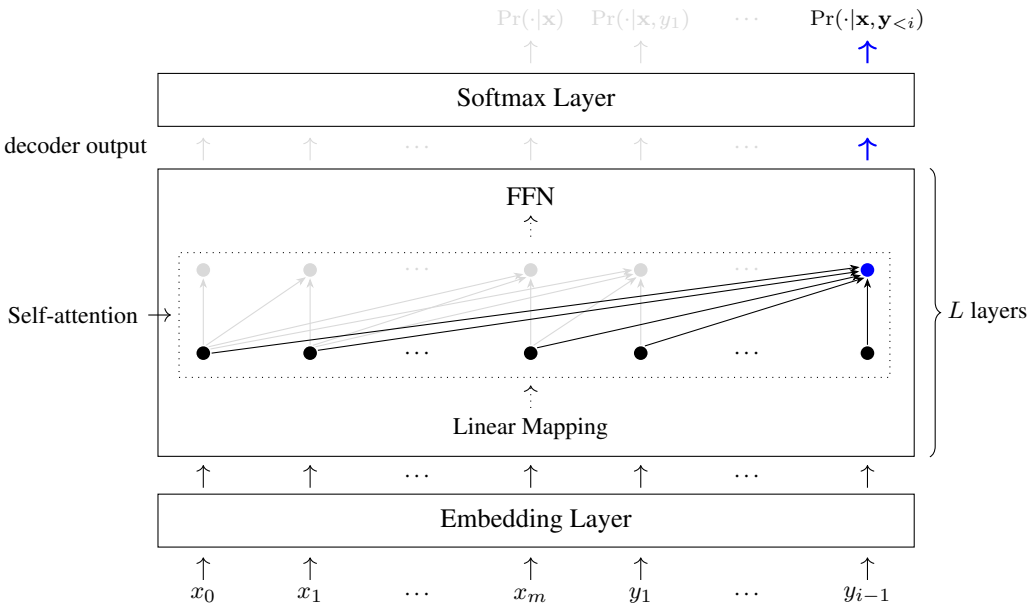


Figure 11.1: The decoder-only architecture for LLMs. The decoder consists of an embedding layer and a stack of Transformer layers. In each Transformer layer, the input passes through a linear mapping, a self-attention network, and an FFN. The output of the decoder is a sequence of representations that are taken as input to a Softmax layer, which generates a distribution of tokens for each position.

dimensionality of each representation, and $i' = m + i$ is the number of context tokens. We can then use a Softmax layer to transform these representations into distributions of tokens. $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$ is the linear mapping matrix of the Softmax layer, and $\mathbf{H}\mathbf{W}^o$ transforms the d-dimensional representations in $\mathbf{H}$ into the $|V|$-dimensional representations. The use of the subscript $m + i$ indicates that the Softmax function is performed only on the representation at position $m + i$. See Figure 11.1 for an illustration of this architecture.

$\mathrm{Dec}(\cdot)$ is a Transformer decoding network that consists of an embedding network and a number of stacked self-attention and FFN networks. We will not discuss Transformers in detail here, as readers can easily learn about these models from the literature. However, it is worth pointing out that the difficulty of inference is in part from the use of the self-attention mechanism in Transformers. Recall that a general form of single-head self-attention is given by

$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{q}_{i'}, \mathbf{K}, \mathbf{V}) \quad = \quad \mathrm{Softmax}(\frac{\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}})\mathbf{V} \tag{11.7}$$

where $\mathbf{q}_{i'} \in \mathbb{R}^d$ is the query at the position i' (i.e., position of y_i), and $\mathbf{K}$ and $\mathbf{V} \in \mathbb{R}^{i' \times d}$ are the keys and values up to i', respectively.

At each step during inference, we call the self-attention function $\mathrm{Att}_{\mathrm{qkv}}(\cdot)$, followed by

an FFN, to generate a d-dimensional representation that integrates information from both the current token and its left context. This process is repeated through L layers of self-attention and FFN, forming a stack of Transformer layers. The output of the L-th layer in this stack is the final representation.

Each time, the model attends position i' to all previous positions, which results in $2i'$ vector products (i' times for $\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}$ and i' times for the product of $\mathrm{Softmax}(\frac{\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}})$ and $\mathbf{V}$). Hence, generating a sequence of length len has a time complexity of $O(L \times len^2)$ for the self-attention network. Clearly, the inference of this model is slow for long sequences due to its quadratic time complexity with respect to sequence length. Therefore, many improvements to Transformers and alternative models have focused on efficient methods that are faster than this quadratic time complexity, such as sparse attention mechanisms and linear-time models. A detailed discussion of efficient Transformers can be found in the previous chapters, and this section will focus on the standard Transformer architecture.

Note that in self-attention, the queries, keys, and values of a layer are linear mappings from the same input (i.e., the output of the previous layer). Once a new key-value pair is generated, it is repeatedly used in subsequent inference steps. Rather than regenerating these key-value pairs during inference, a more desirable way is to store them in a structure, called the **key-value cache**, or the **KV cache**. Thus, $(\mathbf{K}, \mathbf{V})$ can straightforwardly be considered a KV cache. This cache is updated as follows

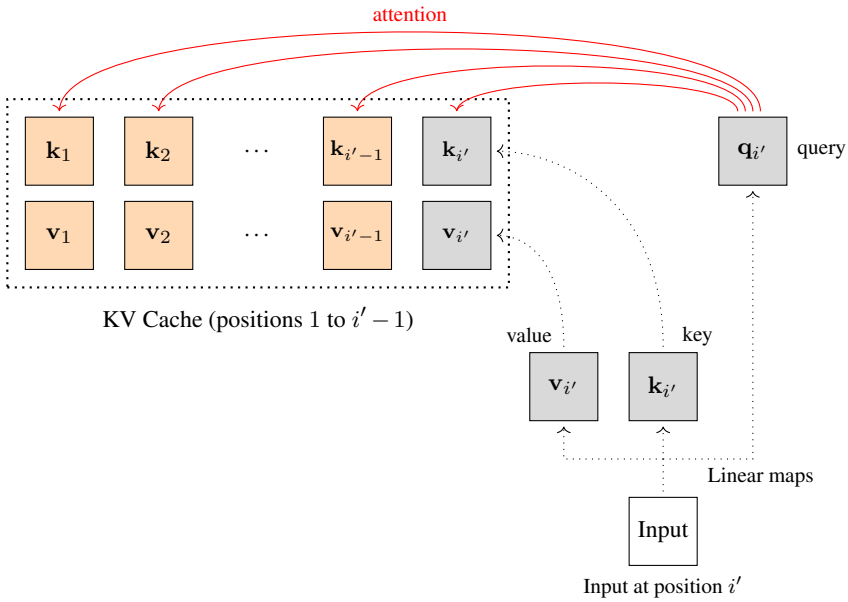
$$\mathbf{K} = \mathrm{Append}(\mathbf{K}, \mathbf{k}_{i'}) \tag{11.8}$$

$$\mathbf{V} = \mathrm{Append}(\mathbf{V}, \mathbf{v}_{i'}) \tag{11.9}$$

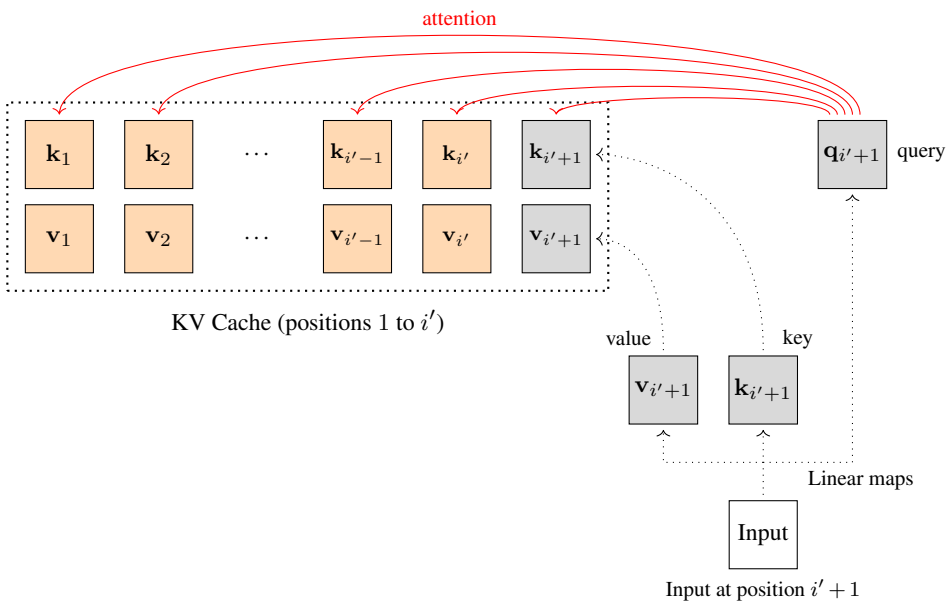
where $(\mathbf{k}_{i'}, \mathbf{v}_{i'})$ is the newly generated key-value pair at position i', and $\mathrm{Append}(\mathbf{a}, \mathbf{b})$ denotes a function that appends a row vector $\mathbf{b}$ to a matrix $\mathbf{a}$. Figure 11.2 shows how a Transformer decoder works with a KV cache.

Finally, the process of computing $\log \Pr(\mathbf{y}|\mathbf{x})$ is summarized as follows:

1. We concatenate $\mathbf{x}$ and $\mathbf{y}$ into a sequence $[\mathbf{x}, \mathbf{y}]$. For each position i' of this sequence, we perform the following steps.
   (a) We compute the embedding of the token at position i', and feed the resulting embedding as an initial representation into the stack of Transformer layers.
   (b) In each Transformer layer, we pass the input representation through the self-attention network first and then through an FFN. In the self-attention network, the input representation is transformed into $\mathbf{q}_{i'}$, $\mathbf{k}_{i'}$, and $\mathbf{v}_{i'}$. Then, we update the KV cache $(\mathbf{K}, \mathbf{V})$ using $\mathbf{k}_{i'}$ and $\mathbf{v}_{i'}$ (see Eqs. (11.8-11.9)). Then, we compute the output of the attention model by attending $\mathbf{q}_{i'}$ to $(\mathbf{K}, \mathbf{V})$ (see Eq. (11.7)).
   (c) If $i' > m$ (i.e., $i = i' - m \geq 0$), we take the output of the Transformer stack and compute the token prediction probability $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ via the Softmax layer (see Eq. (11.5)).
2. When reaching the end of the sequence, we obtain $\log \Pr(\mathbf{y}|\mathbf{x})$ by summing $\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ over $i \in [1, n]$ (see Eq. (11.4)).



(a) Updating the KV Cache at Position i'



(b) Updating the KV Cache at Position $i'+1$

Figure 11.2: Illustration of the KV cache. We update the KV cache at a position, perform the attention operation, and then move to the next position to repeat the process.

11.1.2 A Two-phase Framework

As we have seen, language modeling is a standard autoregressive process, where each token is generated one at a time, conditioned on the previous tokens. For Transformers, this requires the model to maintain a KV cache that stores past representations, and attend the newly generated representation to this KV cache. If we think of the model $\Pr(\mathbf{y}|\mathbf{x})$ from the perspective of computing the KV cache, it is natural to divide inference into two phases:

- **Prefilling**. The prefilling phase computes the KV cache for the input sequence $\mathbf{x}$. It is called prefilling because the model prepares and stores the key-value pairs for each token in the input before the actual inference begins. The process of prefilling in an LLM can be expressed as

$$\text{cache} \quad = \quad \text{Dec}_{\text{kv}}(\mathbf{x}) \tag{11.10}$$

  where $\text{Dec}_{\text{kv}}(\cdot)$ is the decoding network (i.e., the same as $\text{Dec}(\cdot)$), but it returns the KV cache in self-attention instead of the output representations. cache is a list, given by

$$\text{cache} \quad = \quad \{\text{cache}^1, ..., \text{cache}^L\} \tag{11.11}$$

  where cache^l represents the key-value pairs for the l-th layer.

- **Decoding**. The decoding phase continues generating tokens based on the KV cache, as illustrated in Figure 11.2. When a new token is input into the decoder, we update the KV cache in each layer by adding the new key-value pair. The updated cache is then used for self-attention computation. The token generation stops when some stopping criterion is met, such as when the generated token is the end symbol. The goal of decoding is to find the best predicted sequence, which is given by

$$\hat{\mathbf{y}} \quad = \quad \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\text{cache}) \tag{11.12}$$

  Here we use $\Pr(\mathbf{y}|\text{cache})$ instead of $\Pr(\mathbf{y}|\mathbf{x})$ to emphasize that the decoding process actually relies on the KV cache rather than $\mathbf{x}$.

The prefilling and decoding processes are illustrated in Figure 11.3. Note that both these processes are autoregressive. However, as shown in Table 11.1, they differ in several aspects, which lead to very different implementations in practice.

In essence, while the underlying model of prefilling is based on token prediction, it can be considered an encoding process. This is because our goal is not to generate tokens, but to build a context representation (i.e., the KV cache) for the subsequent steps in the decoding phase. In this sense, it is similar to BERT, where we encode the input sequence into a sequence of contextualized token representations. On the other hand, unlike BERT which generates bidirectional sequence representations, prefilling is based on standard language modeling tasks, and is thus unidirectional. Note that, since the entire sequence $\mathbf{x}$ is input into the model all at once, all queries can be packed together and the self-attention operation is performed on $\mathbf{x}$

	Prefilling	Decoding
Goal	Set up initial context $\mathbf{x}$.	Continue generating tokens $\mathbf{y}$ after the initial input.
All-at-once Visibility	Tokens in $\mathbf{x}$ are presented all at once.	Tokens in $\mathbf{y}$ are presented sequentially, that is, predicting a token requires waiting for the previous tokens to be predicted first.
Context Use	Build the context or encoded representation of the input.	Use the cached key-value pairs (from prefilling) to generate further tokens.
Resource Limitation	Compute-bound	Memory-bound
Computational Cost	High	Very High

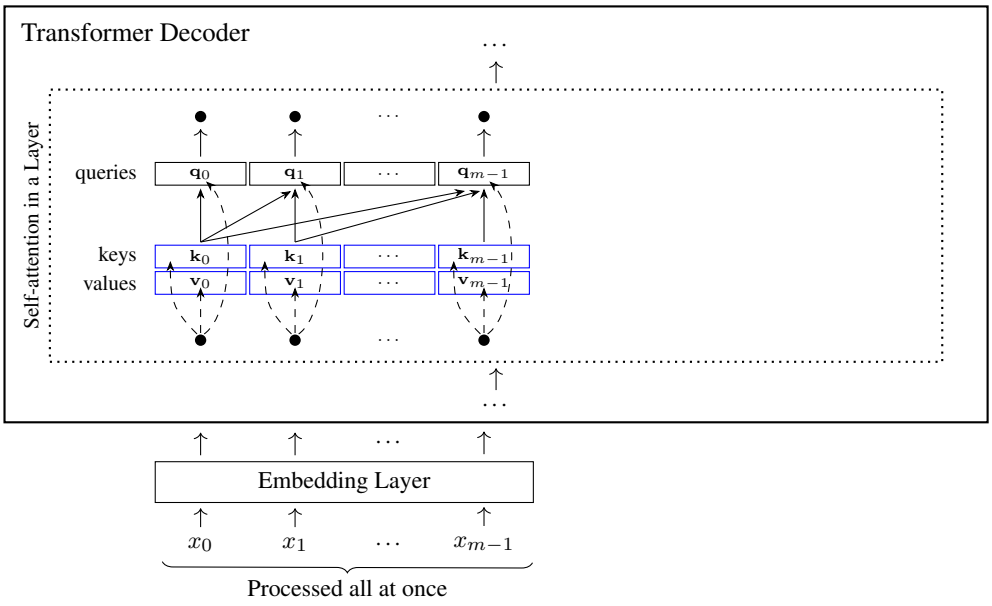
Table 11.1: Prefilling vs Decoding.

in parallel. Let $\mathbf{Q}$ be the queries that are packed into one matrix. The self-attention model in prefilling can be defined as

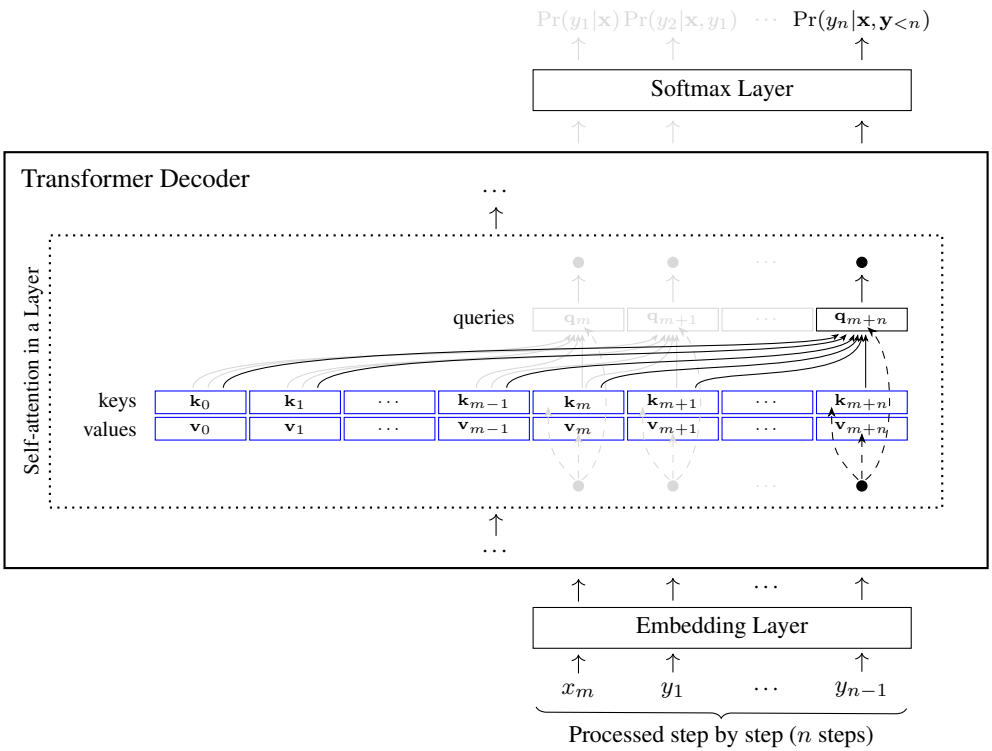
$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{Q},\mathbf{K},\mathbf{V}) \quad = \quad \mathrm{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}}+\mathbf{Mask})\mathbf{V} \tag{11.13}$$

where $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{d\times(m+1)}$. $\mathbf{Mask} \in \mathbb{R}^{(m+1)\times(m+1)}$ is a mask that ensures that each token only attends to itself and the tokens that precede it in the sequence. It is represented by setting the values in the mask corresponding to future tokens to a large negative number, for example, for the query $\mathbf{q}_i$ and the key $\mathbf{k}_j$, we set the value of the entry (i,j) to $-\infty$ if $i<j$. One advantage of processing the sequence with a single self-attention computation is that we can make better use of the parallel computing capabilities of modern GPUs, and so speed up prefilling. In general, the prefilling process is considered compute-bound. This is because merging multiple computational operations into one operation reduces the number of data transfers and the performance bottleneck usually comes from the computational capacity rather than memory bandwidth.

Decoding is a standard left-to-right text generation process. The token sequence is generated autoregressively by predicting one token at a time based on the KV cache. Each time a new token is generated, we need to attend it to previous tokens, following Eq. (11.7). Therefore, the decoding process is memory-bound due to its frequent access to the KV cache. The cost of decoding grows significantly as more tokens are generated. In most cases, decoding is computationally more expensive than prefilling. Note that this is not just because, in decoding, the LLM generates tokens one by one and repeatedly updates the KV cache. As we will see in the following subsection, we may need to explore multiple different token sequences during decoding, which makes the problem more complex and increases its cost further.



(a) Prefilling



(b) Decoding (at the n-th step)

Figure 11.3: Illustration of the prefilling and decoding processes. In prefilling, the entire input sequence is processed together and the KV cache is filled. In decoding, the LLM generates the output sequence step by step based on the prefilled KV cache.

11.1.3 Decoding Algorithms

So far our discussion of LLM inference has primarily focused on the model computation problem, that is, how to compute $\Pr(\mathbf{y}|\mathbf{x})$. Now we turn to the discussion of the search problem. The problem can be stated as: given an LLM $\Pr(\mathbf{y}|\mathbf{x})$, how do we efficiently search for the best output sequence $\hat{\mathbf{y}}$ given the input sequence $\mathbf{x}$ (or the generated KV cache)? Naively, we can consider all of the output sequences, compute the prediction probability for each, and then select the output sequence having the highest probability. This method can guarantee the globally optimal solution, but direct exhaustive search is impractical for LLMs as the number of possible output sequences grows exponentially with the length of $\mathbf{y}$.

In practice, various heuristic search algorithms, such as greedy search and sampling-based search, are commonly employed to approximate the solution. Each of these methods offers trade-offs between search quality and computational efficiency. The search problem, therefore, becomes a balancing act between exploration and exploitation, where the goal is to find an efficient strategy that produces high-quality outputs without exploring the entire space.

Before giving a more detailed discussion of these methods, let us first informally define what a search space is and how it is represented. In LLM inference, we define a hypothesis as a tuple of input and output sequences. Since $\mathbf{x}$ is fixed during inference, we can simply consider each hypothesis as an output sequence. The search space, denoted by $\mathcal{Y}$, is then the set of all possible hypotheses (i.e., output sequences) that the model can generate. The search problem for LLM inference can be re-expressed as

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}\in\mathcal{Y}} \Pr(\mathbf{y}|\mathbf{x}) \tag{11.14}$$

In NLP, $\mathcal{Y}$ is commonly represented in a tree data structure to facilitate search. Figure 11.4 shows an example of the search tree resulting from a small vocabulary. In this example, a node represents a prefix subsequence that can be shared by many sequences. The search starts with the root of the tree, which can be regarded as the beginning of all sequences that can be generated[1]. Each child node extends the prefix of its parent node by adding one token from the vocabulary to the sequence, along with the probability of predicting the token given the prefix. This process continues as each node further branches out into additional child nodes, each representing a new possible extension of the sequence with another token. The search tree thus grows deeper and wider, representing an ever-increasing number of potential sequences as more tokens are appended. This structure allows us to efficiently traverse through possible sequences, evaluating each in terms of the log-probability accumulated over the path from the root to that node. For example, in Figure 11.4, the path from the root to the node 17 corresponds to the output sequence "*Cats are playful.*". The prediction log-probability $\log\Pr(\mathbf{y}|\mathbf{x})$ is the sum of the log-probabilities of all the nodes on this path.

In general, the search tree is organized as levels, where each level consists of all nodes that are the same distance from the root node. Thus, a breadth-first search over the tree essentially performs left-to-right generation of tokens. Nodes in the same level correspond to sequences

[1] Here, since the predictions in LLMs are based on $\mathbf{x}$, we can think of the root as a representation of $\mathbf{x}$.

Path: node $0 \rightarrow$ node $3 \rightarrow$ node $9 \rightarrow$ node $11 \rightarrow$ node 17
Output: cats are playful.

Probability:
node $0 \rightarrow 0$
node $3 \rightarrow \log \Pr(\text{"cats"}|\mathbf{x})$
node $9 \rightarrow \log \Pr(\text{"are"}|\mathbf{x}, \text{"cats"})$
node $11 \rightarrow \log \Pr(\text{"playful"}|\mathbf{x}, \text{"cats are"})$
node $17 \rightarrow \log \Pr(\text{"."}|\mathbf{x}, \text{"cats are playful"})$

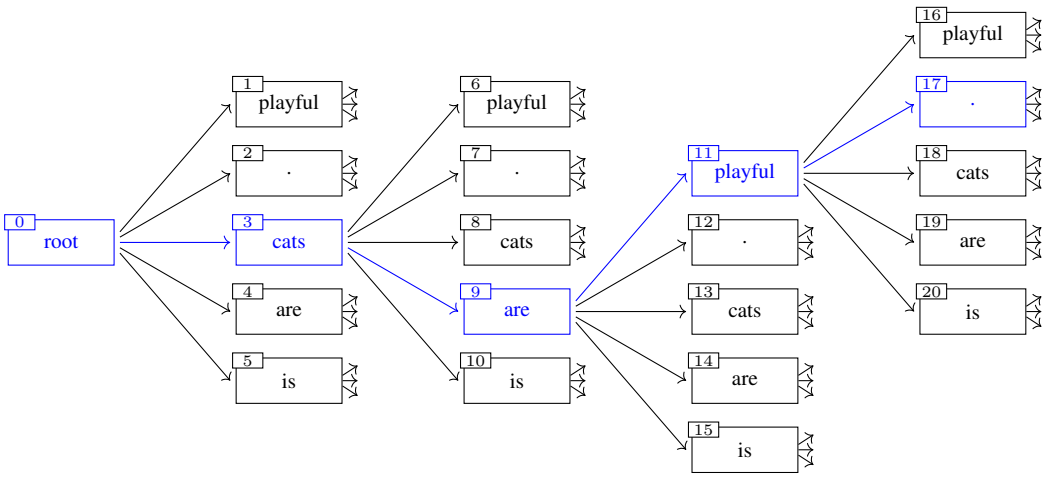


Figure 11.4: A search tree for decoding. At each node, we expand the tree by considering all possible tokens, each leading to a new node representing a potential continuation of the text. Here we highlight a path through nodes 0, 3, 9, 11, and 17. The path represents the output sequence "*cats are playful.*", whose log-probability can be computed by accumulating the log-probabilities of these nodes.

of the same length. As the search progresses, new tokens are appended to these sequences, expanding them incrementally.

Let Y_i be the set of the sequences that the LLM generates at step i. Y_i can be obtained by expanding each sequence in Y_{i-1} with all possible next tokens in the vocabulary V, given in the following recursive form

$$Y_i \quad = \quad Y_{i-1} \times V \tag{11.15}$$

where $Y_{i-1} \times V$ denotes the Cartesian product of Y_{i-1} and V (i.e., each sequence in Y_{i-1} is concatenated with each token in V). Note that if a sequence in Y_{i-1} is complete (e.g., ending with the $\langle \text{EOS} \rangle$ token), it will not be expanded any further. Let $\Psi(Y_i)$ be the set of all complete sequences in Y_i. Then, the search space can be expressed as

$$\mathcal{Y} \quad = \quad \Psi(Y_1) \bigcup \Psi(Y_2) \bigcup \cdots \bigcup \Psi(Y_{n_{\max}}) \tag{11.16}$$

where $n_{\max}$ is the maximum length of a sequence.

Most decoding algorithms follow this level-by-level search process. However, $\mathcal{Y}$ consists

of an exponentially large number of sequences, and a direct search in such a vast space is computationally infeasible. Therefore, practical decoding algorithms often rely on strategies to prune the search space and avoid exploring low-quality sequences. For example, at each decoding step, Y_i can be obtained in the following way

$$Y_i \quad = \quad \mathrm{Prune}(Y_{i-1} \times V) \tag{11.17}$$

where $\mathrm{Prune}(\cdot)$ is a function that selectively removes sequences less likely to result in high-quality outcomes. In general, we expect that $|Y_i| << |Y_{i-1}| \cdot |V|$. Thus we can drastically reduce the number of sequences under consideration at each step, ensuring that the computational load does not grow exponentially with the sequence length.

Next, we will introduce these decoding algorithms. Some of them have already been discussed in sequence-to-sequence models (see Chapter 5), while others are more commonly used in LLMs.

1. Greedy Decoding

Greedy search (or greedy decoding) is one of the most widely used decoding methods in NLP, particularly in text generation tasks like machine translation. The idea behind greedy search is straightforward: at each step in generation, it selects the next token that has the highest prediction probability. For each sequence $\mathbf{y} = y_1 ... y_i \in Y_{i-1} \times V$, we can evaluate it using $\log \Pr(\mathbf{y}|\mathbf{x})$. This log-probability can be easily computed by noting that

$$\begin{aligned} \log \Pr(\mathbf{y}|\mathbf{x}) &= \log \Pr(y_1 ... y_i|\mathbf{x}) \\ &= \underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{accumulated up to the parent node}} + \underbrace{\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}_{\text{newly computed for the current node}} \end{aligned} \tag{11.18}$$

Here the first term is the sum of the log-probabilities of the path from the root to the parent node, which has been computed in the previous decoding steps. At step i, we only need to compute the second term which is the standard token prediction log-probability produced by the LLM.

The "best" token at step i is then chosen as

$$\begin{aligned} y_i^{\mathrm{top1}} &= \arg\max_{y_i \in V} \log \Pr(y_1 ... y_i|\mathbf{x}) \\ &= \arg\max_{y_i \in V} \big[\underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{fixed wrt. } y_i} + \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \big] \\ &= \arg\max_{y_i \in V} \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \end{aligned} \tag{11.19}$$

Thus, the "best" sequence generated up to step i is given by

$$\mathbf{y}^{\mathrm{top1}} \quad = \quad y_1 ... y_{i-1} y_i^{\mathrm{top1}} \tag{11.20}$$

Finally, Y_i contains only this sequence

$$Y_i = \{\mathbf{y}^{\text{top1}}\} \tag{11.21}$$

The greedy choice in one decoding step is illustrated in Figure 11.5 (a). Greedy search offers computational efficiency and simplicity in implementation for LLM inference. Its primary disadvantage, however, lies in its suboptimal nature — high-quality sequences are likely pruned at early stages of decoding. Therefore, greedy search is appealing for tasks that demand speed and simplicity. For tasks that require better search results, alternative strategies such as beam search, which explores multiple potential paths simultaneously, are preferable.

2. Beam Decoding

Beam search (or beam decoding) is a natural extension of greedy search. Instead of selecting the single most probable token at each step, beam search maintains a fixed number of the best candidates at each step, known as the "beam width". See Figure 11.5 (b) for an illustration of beam search.

Let K be the beam width. Given a parent node, which corresponds to the prefix $y_1...y_{i-1}$, we can select the top-K next tokens by

$$\{y_i^{\text{top1}}, ..., y_i^{\text{top}K}\} = \underset{y_i \in V}{\text{argTopK}} \ \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{11.22}$$

where argTopK is a function that ranks the prediction probabilities of all possible next tokens and selects the top K candidates. Given these tokens, the top-K sequences for step i are given by

$$\mathbf{y}^{\text{top1}} = y_1...y_{i-1}y_i^{\text{top1}} \tag{11.23}$$

$$\vdots$$

$$\mathbf{y}^{\text{top}K} = y_1...y_{i-1}y_i^{\text{top}K} \tag{11.24}$$

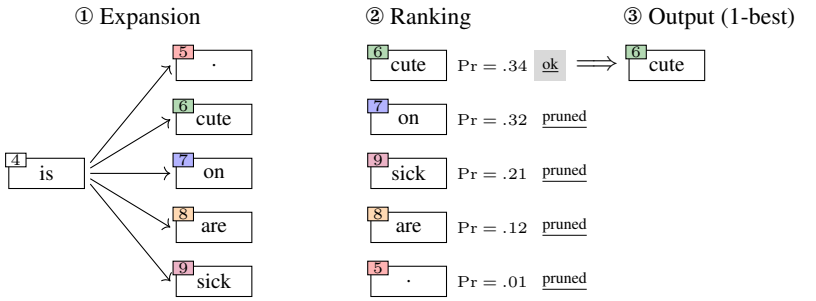
Then, we can define Y_i as

$$Y_i = \{\mathbf{y}^{\text{top1}}, ..., \mathbf{y}^{\text{top}K}\} \tag{11.25}$$

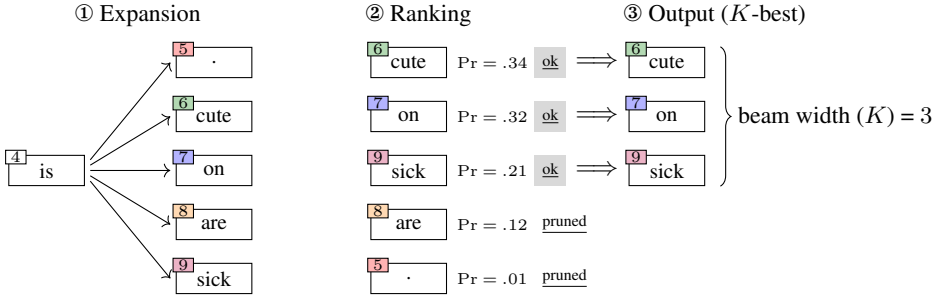
We can adjust the beam width K to balance search efficiency and accuracy. But a very large beam width might not be helpful. In many practical applications, selecting a relatively small number for K, such as $K = 2$ or $K = 4$, is often sufficient to achieve satisfactory performance in LLM inference.

3. Sampling-based Decoding

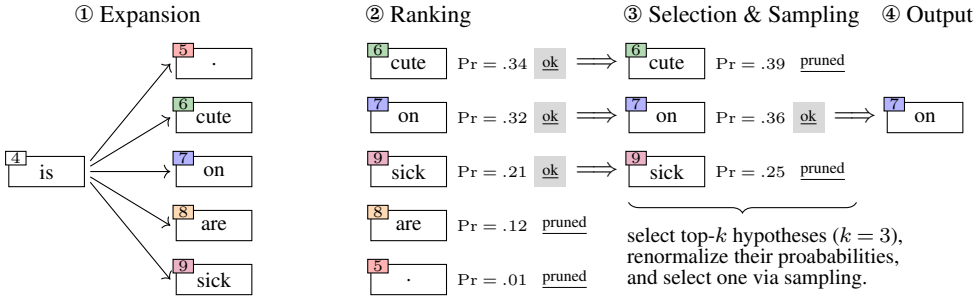
Both greedy and beam search generate deterministic outputs, that is, given an LLM, the output of the model will always be the same every time it processes the same input. The deterministic nature of greedy and beam search ensures predictability and reliability in applications where



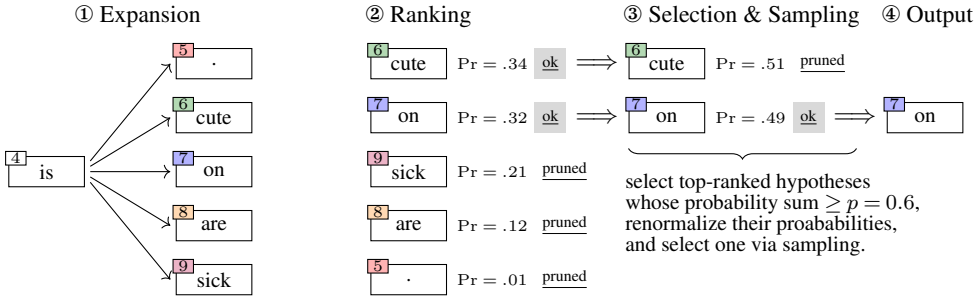
(a) Greedy search



(b) Beam search



(c) Top-k Sampling



(d) Top-p Sampling

Figure 11.5: Illustrations of greedy decoding, beam decoding, top-k decoding and top-p decoding methods (in one decoding step).

consistent outcomes are critical, such as in formal document generation, where varying outputs could cause confusion or errors. On the other hand, one disadvantage of these methods is the lack of diversity and flexibility. For example, in creative tasks like story generation or conversational agents, generic or repetitive outputs generated by deterministic systems are often less engaging.

To add variation into LLM outputs, we can use sampling-based decoding methods. There are two commonly used methods.

- **Top-k Sampling**. This method selects the next token from the top-k most likely candidates at each step of the generation process [Fan et al., 2018]. Let $\overline{V}_i$ be the selection pool for top-k sampling. We can define it as

$$\overline{V}_i = \{y_i^{\text{top1}}, ..., y_i^{\text{top}k}\} \tag{11.26}$$

  where $\{y_i^{\text{top1}}, ..., y_i^{\text{top}k}\}$ are the top-k tokens selected based on their prediction probabilities (see Eq. (11.22)). Once the selection pool is determined, we recompute the prediction probability distribution over $\overline{V}_i$. One of the simplest ways to do this is to renormalize their probabilities:

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}{\sum_{y_j \in \overline{V}_i} \Pr(y_j|\mathbf{x}, \mathbf{y}_{<i})} \tag{11.27}$$

  Alternatively, we can calculate the distribution by using the Softmax function:

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i})}{\sum_{y_j \in \overline{V}_i} \exp(u_{y_j})} \tag{11.28}$$

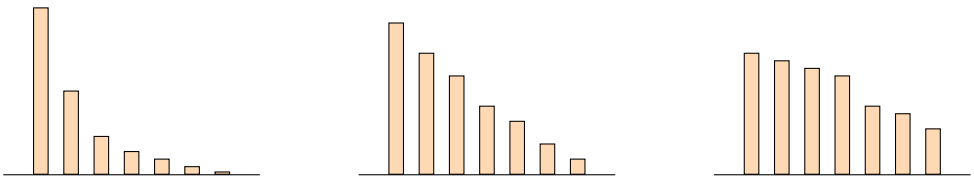
  where u_{y_i} is the logit for token y_i. Then, we sample a token $\bar{y}_i$ from this distribution:

$$\bar{y}_i \sim \overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{11.29}$$

  The corresponding sequence is $\bar{\mathbf{y}} = y_1 ... y_{i-1} \bar{y}_i$, and Y_i is given by

$$Y_i = \{\bar{\mathbf{y}}\} \tag{11.30}$$

- **Top-p Sampling**. This sampling method, also known as **nucleus sampling**, follows a procedure similar to that of top-k sampling. Instead of drawing from a fixed size candidate pool, it selects the next token from the smallest set of tokens that together have a cumulative probability higher than a predefined threshold p [Holtzman et al., 2020]. In this way we prevent the prediction from choosing from low-probability tokens in the long tail that could lead to incoherent or nonsensical outputs. To obtain the candidate pool in the top-p sampling method, we can sort all tokens by their predicted probabilities. Then, starting with the token with the highest probability, we continue to add tokens to the candidate pool until the cumulative probability of the tokens in the pool reaches or exceeds p (we denote the size of the candidate pool at this point as k_p). The candidate



(a) $\beta = 0.1$ (b) $\beta = 0.8$ (c) $\beta = 2.0$

Figure 11.6: Histogram estimates of the distributions generated by the Softmax function with different values of the temperature parameter β.

pool can then be expressed as

$$\overline{V}_i = \{y_i^{\text{top1}}, ..., y_i^{\text{top}k_p}\} \tag{11.31}$$

The subsequent steps, such as the renormalization of the distribution and sampling, are the same as in the top-k sampling method (see Eqs.(11.27-11.30)).

See Figure 11.5 (c-d) for illustrations of the top-k and top-p sampling methods. By limiting the choices to a smaller set of high-probability tokens, these methods strike a balance between randomness and coherence. They allow for more diverse outputs while still maintaining a reasonable level of relevance and fluency. However, the value of k or p must be carefully chosen: if k or p is too small, the output may still be overly deterministic (more like greedy decoding), and if k or p is too large, the LLM might produce degenerate outputs.

In order to further control the randomness of the token selection process, the renormalized distribution $\overline{\Pr}(\cdot)$ is typically obtained by using the Softmax function with the temperature parameter, given by

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i}/\beta)}{\sum_{y_j \in \overline{V}_i} \exp(u_{y_j}/\beta)} \tag{11.32}$$

Here β is a temperature parameter β that controls the sharpness of the probability distribution derived from logits. In Figure 11.6, we show simple examples involving distributions generated by the above function with different temperatures. When the temperature is set to a higher value, the resulting probability distribution becomes more uniform, as the differences between the logits are diminished. This means that each token in the candidate pool has a more equal chance of being selected, leading to greater diversity in the generated output. By contrast, when the temperature is set to a lower value, the distribution becomes sharper, making the high-probability tokens even more likely to be chosen, which often results in more deterministic outputs. For example, if we set p to 1 and β to a very small number (approaching zero), the top-p sampling method will become equivalent to the greedy search method.

4. Decoding with Penalty Terms

One common improvement to decoding methods in text generation is to modify the search objective. For example, one can replace maximum a posteriori (MAP) decoding with minimum

Bayes risk (MBR) decoding [Kumar and Byrne, 2004], where the focus shifts from selecting the single most probable output to choosing an output that minimizes the expected risk over a distribution of possible outputs. More details on MBR decoding can be found in Chapter 5. Here we explore methods that incorporate penalty terms into decoding. These methods offer a simple but effective way to make decoding more controllable.

Recall from Eq. (11.14) that the goal of decoding is to maximize the likelihood of the output sequence. With penalty terms, the objective is extended to include additional factors that penalize or reward certain behaviors in the generated text. A general form of the new objective is given by

$$\hat{\mathbf{y}} \quad = \quad \arg\max_{\mathbf{y} \in \mathcal{Y}} \left[\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y}) \right] \tag{11.33}$$

where $\text{Penalty}(\mathbf{x}, \mathbf{y})$ is a function that quantifies the degree to which the generated sequence $\mathbf{y}$ violates certain constraints or exhibits undesirable behaviors given the input $\mathbf{x}$. The design of $\text{Penalty}(\cdot)$ is very flexible, thus allowing us to incorporate a wide range of constraints or prior knowledge into it. Below, we present some common types of penalty functions.

- **Repetition Penalty**. A repetition penalty discourages the model from generating repetitive or redundant text. The penalty function might measure the frequency of repeated tokens or phrases in the generated sequence and impose a penalty proportional to their occurrence.
- **Length Penalty**. A length penalty ensures that the generated sequence adheres to a desired length. For example, in text summarization tasks, the penalty function could penalize outputs that are too short or too long.
- **Diversity Penalty**. A diversity penalty promotes variation in the generated text. For example, in beam search, we can measure the similarity between generated hypotheses, and encourage the model to explore different hypotheses.
- **Constraint-based Penalty**. A constraint-based penalty enforces specific constraints related to the content or style of the generated text. For example, in machine translation, the penalty function could penalize outputs that deviate from a desired tone or terminology.

In general, we can consider $\text{Penalty}(\mathbf{x}, \mathbf{y})$ as a function that defines the cost of generating the surface form of the output sequence $\mathbf{y}$ given the input sequence $\mathbf{x}$. Alternatively, this function can be defined to assess the hidden states of an LLM when generating $\mathbf{y}$. For example, Su et al. [2022] develop a penalty term that calculates the maximum distance between the representation of the predicted token and the representations of the previously generated tokens. Therefore, the search objective will penalize degenerated outputs, such as texts with many repetitions.

The method described in Eq. (11.33) is general and can be easily adapted to different search algorithms. For example, in greedy search, we can keep the single sequence that maximizes $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ at each decoding step; in sampling-based search, we can rank and

select the top-ranked sequences based on $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ to form the candidate pool.

5. Speculative Decoding

Speculative decoding stems from the concept of **speculative execution**, where a system makes educated guesses about future actions and performs them in advance. If the guess is correct, the results are immediately available, which speeds up processing. In the case of LLM inference, suppose we have two models. One is a smaller, faster model (called draft model), and the other is the full, more accurate model (called verification model). These two models represent two baselines in LLM inference: the draft model is efficient but not very accurate; the verification model is usually the one we want to run, but it is very slow. Given a prefix, we first use the draft model to speculatively predict a sequence of likely future tokens. This is a standard autoregressive decoding process, but it is still fast in practice due to the high efficiency of the draft model. Then, the verification model evaluates the speculated tokens in parallel. It checks whether the predicted tokens are correct or need to be adjusted. Note that, since we can deal with these tokens all at once, the verification can be done in a single step for all the tokens simultaneously, rather than in a token-by-token manner. If the speculated tokens are correct, they are accepted, and the process continues with the next set of tokens. If they are incorrect, the incorrect speculations are discarded, and the verification model is used to generate the correct tokens.

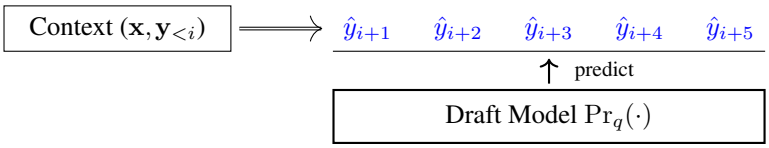
To be more specific, let us see the speculative decoding method presented in Leviathan et al. [2023]'s work. In this method, the draft model is a small language model, denoted by $\Pr_q(y_i|\mathbf{x}, \mathbf{y}_{<i})$, while the verification model is a normal LLM, denoted by $\Pr_p(y_i|\mathbf{x}, \mathbf{y}_{<i})$. The goal is that, given a prefix, we use the draft model to autoregressively predict up to τ tokens. The verification model is then employed to generate the last token at the point where errors begin to occur in the speculative predictions. Figure 11.7 illustrates one step in this decoding process.

The speculative decoding algorithm can be summarized as follows.

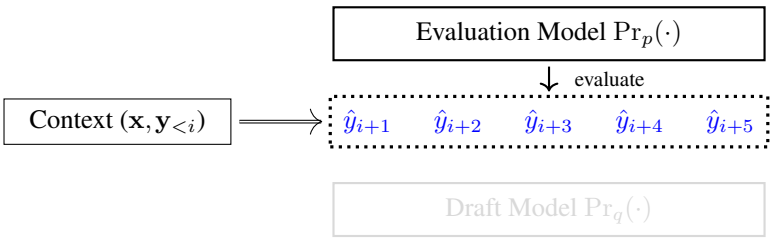
- Given the prefix $[\mathbf{x}, \mathbf{y}_{\leq i}]$, we use the draft model to predict the next τ consecutive tokens, denoted by $\{\hat{y}_{i+1}, ..., \hat{y}_{i+\tau}\}$. This is a token-by-token generation process, given by

$$\hat{y}_{i+t} = \arg\max_{y_{i+t}} \Pr_q(y_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1}) \tag{11.34}$$

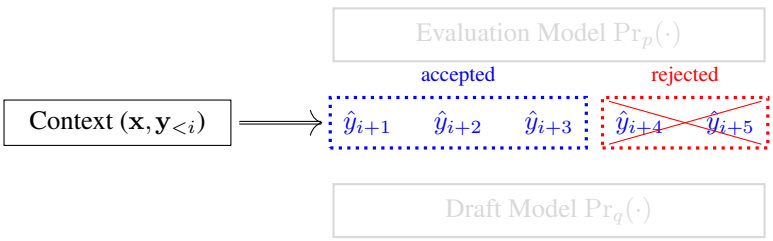
- We evaluate $\{\hat{y}_{i+1}, ..., \hat{y}_{i+\tau}\}$ using the verification model, that is, we compute $\{\Pr_p(\hat{y}_{i+1}|\mathbf{x}, \mathbf{y}_{\leq i})$ $, ..., \Pr_p(\hat{y}_{i+\tau}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+\tau-1})\}$. Note that we can compute these probabilities in parallel, and so this verification step is efficient.
- We determine the maximum number of accepted speculated tokens. In order to keep the notation uncluttered, we denote $\Pr_q(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1})$ and $\Pr_p(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1})$ simply by $q(\hat{y}_{i+t})$ and $p(\hat{y}_{i+t})$, respectively. We then define that, if $q(\hat{y}_{i+t}) \leq p(\hat{y}_{i+t})$, then we accept this speculation. By contrast, if $q(\hat{y}_{i+t}) > p(\hat{y}_{i+t})$, we reject this speculation with probability $1 - \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}$. Starting from $\hat{y}_{i+1}$, the maximum number of accepted



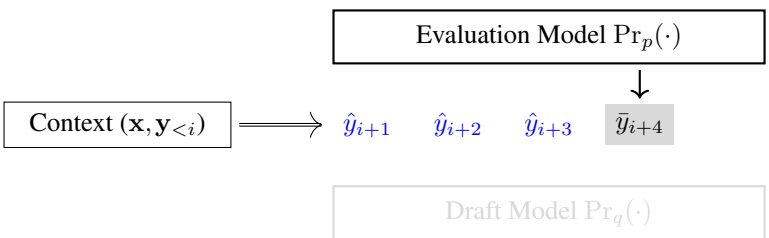
(a) Predict the next τ tokens given the context using the draft model ($\tau = 5$)



(b) Evaluate the predicted tokens using the evaluation model



(c) Determine the number of accepted tokens



(d) Predict a new token following the accepted tokens using the evaluation model

Figure 11.7: Illustration of one step of speculative decoding. The goal is to predict as many next tokens as possible using the draft model. There are four sub-steps. Given the context, we first use the draft model to predict the next τ tokens. Then, we evaluate these predictions in parallel using the evaluation model. Next, we determine the maximum number of predicted tokens that can be accepted. Finally, we use the evaluation model to predict a new token following these accepted tokens.

consecutive speculated tokens is defined as

$$n_a = \min\left\{t-1 | 1 \leq t \leq \tau, r_t > \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}\right\} \tag{11.35}$$

where r_t is a variable drawn from the uniform distribution $U(0,1)$.

- Given n_a, we keep the speculated tokens $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}\}$. We then use the verification model to make a new prediction at $i+n_a+1$

$$\bar{y}_{i+n_a+1} = \underset{y_{i+n_s+1}}{\arg\max} \Pr_p(y_{i+n_s+1} | \mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1} ... \hat{y}_{i+n_s}) \tag{11.36}$$

- Above, we have described one step of speculative decoding. The result sequence (including both the context and predicted tokens) is illustrated as follows

$$\underbrace{[\mathbf{x},\ \mathbf{y}_{<i}]}_{\text{Context}}\ \underbrace{\hat{y}_{i+1} ... \hat{y}_{i+n_a}}_{\substack{n_a \text{ tokens} \\ \text{predicted using} \\ \text{the draft model}}}\ \underbrace{\bar{y}_{i+n_a+1}}_{\substack{\text{One token} \\ \text{predicted using} \\ \text{the verification model}}}$$

Once we have finished this step, we add the predicted tokens $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}, \bar{y}_{i+n_a+1}\}$ to the context, and repeat the above process.

In practice, we usually wish to use a smaller draft model so that predicting $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}\}$ would be computationally cheaper. But a very small draft model is less accurate and can result in smaller n_a. We therefore need to carefully select the draft model to make the trade-off between the computational efficiency and accuracy.

6. Stopping Criteria

Stopping criteria are a critical component of LLM inference. They typically involve rules or conditions that specify when the model should stop generating text during decoding. Most LLMs are trained to generate an end-of-sequence token (e.g., $\langle \text{EOS} \rangle$ or $\langle /s \rangle$) to signal the end of the generated text. So one of the simplest strategies is that the generation process stops when this token is produced. For beam search, which explores multiple hypotheses simultaneously, the process can continue until a given number of complete sequences have been generated.

In practical applications, it will generally be undesirable to generate very long sequences, and so we need to reduce the decoding cost and unnecessary verbosity. One commonly-used stopping criterion is the maximum length of the output. The model stops generating text once it has produced a predetermined number of tokens. Alternatively, we can stop the decoding based on the real cost, such as the computational resources or time constraints. For example, in real-time applications like chatbots, decoding may need to stop after a certain time limit to ensure responsiveness.

Another approach is to design stopping criteria based on the behavior of LLMs. For example, decoding can be stopped if the probability of predicting the next token falls below a certain threshold. In addition to probability-based stopping, a repetition detection module can

be implemented to trigger the model to stop if it begins repeating tokens or phrases beyond a predefined limit. This helps prevent redundant or incoherent outputs.

11.1.4 Evaluation Metrics for LLM Inference

Evaluating the performance of LLMs during inference involves a variety of metrics to assess how well these models meet desired standards, such as accuracy, robustness, usability, and efficiency. As with most NLP systems, we can evaluate LLMs using accuracy-based metrics, such as perplexity and F1 score. We can also examine their robustness by testing how well they handle ambiguous or challenging inputs, including adversarial, perturbed, or out-of-distribution data. Additionally, usability can be assessed by measuring how well the generated outputs align with user expectations in terms of fluency, coherence, relevance, and diversity. Human evaluators can rate the naturalness of the text or assess whether the responses are contextually appropriate and logically consistent. Ethical and fairness metrics can also be included to ensure LLMs avoid perpetuating biases or generating harmful content.

All of the evaluation metrics mentioned above essentially focus on assessing the quality of the outputs. Given the high cost of deploying and applying LLMs, efficiency metrics are also very important for practitioners. Below are some commonly used efficiency metrics [Nvidia, 2025]:

- **Request Latency**. This metric measures the total time taken from when a request is sent to the LLM until the complete response is received. This includes the time taken for data transmission, processing by the model, and the return of the output to the user.
- **Throughput**. It refers to the number of tokens or requests the model can process per second.
- **Time to First Token** (TTFT). This metric measures the time it takes from the beginning of a request being sent to the generation of the first token of the response. If data transmission does not consume too much time, then TTFT is mainly the time for prefilling and predicting the first token.
- **Inter-token Latency** (ITL). This metric refers to the time taken to generate each subsequent token after the first one. It reflects the efficiency of the decoding process.
- **Tokens Per Second** (TPS). This metric quantifies the number of tokens that the model can generate per second.
- **Resource Utilization**. This involves measuring the computational resource usage (e.g., CPU and GPU utilization) and memory consumption of the model during inference.

In addition to these metrics, energy efficiency and cost efficiency are practical considerations for deploying LLMs at scale. Energy efficiency measures the amount of electrical power consumed by the model during inference. Cost efficiency, on the other hand, evaluates the total expenses related to deploying and maintaining the model.

In general, choosing the right evaluation metrics depends on the specific task and application. While quality-focused metrics are essential for assessing LLMs, efficiency metrics are equally crucial for their effective deployment in real-world applications. A comprehensive

evaluation framework should include both sets of metrics to accurately estimate an LLM's performance and practicality.

11.2 Efficient Inference Techniques

In practical applications, we often wish a system to be as efficient as possible. For LLM inference, this typically involves two types of improvements: reducing memory requirements and accelerating the system. For example, we can modify the Transformer architecture to avoid memory explosion when processing very long input sequences. Another example is that we can compress input sequences to reduce computational overhead while preserving their semantic information. In addition, techniques like quantization and pruning can be employed to further optimize memory usage and inference speed.

Efficient inference is a wide-ranging topic that overlaps with several sub-fields of LLMs, such as architecture design and model compression. Most of these topics have been covered in previous chapters. For example, in Chapter 6, we discussed efficient Transformer architectures; in Chapter 8, we discussed long-context LLMs; and in Chapter 9, we discussed prompt compression methods for reducing prompt length. In this section, we focus on techniques that are commonly used in LLM deployment and serving.

11.2.1 More Caching

In real-world applications, it is common practice to store frequent requests and their corresponding responses in a cache. When a new request hits the cache, the system can retrieve the response directly from the cache instead of recomputing the result. One straightforward implementation is a key-value datastore (e.g., a hash table) that maps input sequences to their LLM-generated output sequences. In the simplest case, we can collect frequent queries, generate their responses using the LLM, and store these query-response pairs in the datastore. This creates a basic sequence-level caching mechanism that allows the system to bypass LLM computation when the input sequence exactly matches a cached query.

A straightforward extension of the caching mechanism is to cache prefixes and their corresponding hidden states. Given an input sequence $\mathbf{x}$ in a dataset $\mathcal{D}$, we can process it as in the standard prefilling phase. Thus, we obtain a sequence of prefixes and their corresponding KV cache states:

$$\begin{aligned} x_0 \ (\mathbf{x}_{<1}) &\Rightarrow \text{cache}_{<1} \\ x_0 x_1 \ (\mathbf{x}_{<2}) &\Rightarrow \text{cache}_{<2} \\ &\dots \\ x_0 x_1 \dots x_{m-1} \ (\mathbf{x}_{<m}) &\Rightarrow \text{cache}_{<m} \end{aligned}$$

where $\text{cache}_{<i}$ denotes the KV cache for the prefix $\mathbf{x}_{<i}$ (see also Eq. (11.10)). All these mappings can be stored in the prefix cache for efficient reuse.

When processing a new sequence that shares a common prefix with a previously seen sequence in $\mathcal{D}$, we can load the corresponding cached hidden states instead of recomputing

them. Specifically, if a new input $\mathbf{x}'$ has $\mathbf{x}_{<k}$ (i.e., $\mathbf{x}'_{<k} = \mathbf{x}_{<k}$ for some $k \leq m$), we can initialize the KV cache with $\text{cache}_{<k}$ and only compute the hidden states for the remaining tokens $\mathbf{x}'_{\geq k}$.

As usual, we can maintain a key-value datastore that maps frequently encountered prefixes to their precomputed KV caches. The lookup can be performed using a hash of the prefix tokens, allowing constant-time access to the cached states. Care must be taken to manage memory usage, as storing all possible prefixes may be infeasible for large datasets. Practical systems often employ least recently used (LRU) caching methods or other strategies to balance between computational savings and memory constraints.

11.2.2 Batching

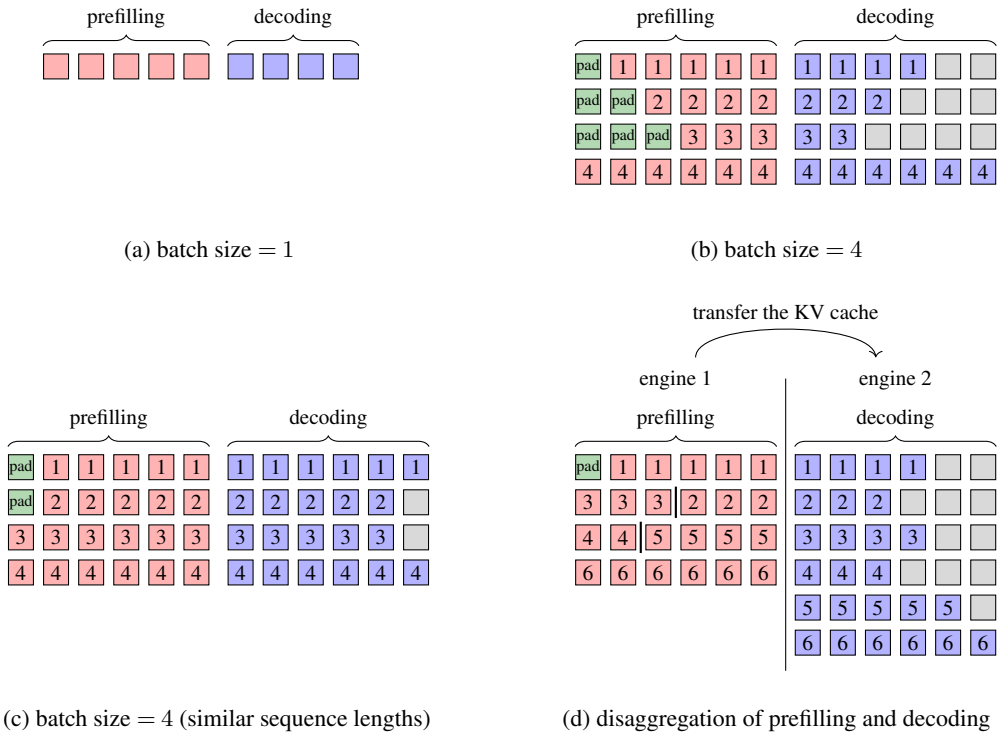
Batching in LLM inference refers to the process of processing multiple input sequences simultaneously as a group (called a batch) rather than one at a time. Because modern GPUs excel at parallel processing, batching allows them to compute multiple sequences in a single forward pass, keeping the hardware fully occupied. Therefore, when serving LLMs at scale, batching is important for improving computational efficiency and maximizing hardware utilization[2].

To illustrate the idea of batching, Figure 11.8 (a-b) show simple examples with batch sizes of 1 and 4, respectively. When using a batch size of 1 (i.e., without batching), the GPU processes one input sequence at a time. Thus, the processing is sequential: the next sequence must wait for the current computation to finish. By contrast, when using a batch size of 4, the GPU can process four sequences simultaneously in a single forward pass. As the input sequences vary in length, we need to standardize their length using padding techniques. Here we use left padding, which adds dummy tokens to the beginnings of short sequences, so all the sequences in the batch would have the same length for prefilling. For decoding, tokens are generated simultaneously for all these sequences, and the generation process continues until the longest sequence reaches completion.

The above examples imply a trade-off between throughput and latency, which is a very important consideration in designing and implementing LLM inference systems. If we choose a smaller batch size, the latency would be lower, as fewer tokens need to be processed in a single run of inference. Imagine that we have only one sequence. The result becomes available immediately after generation completes, with no additional computational overhead. However, this low-latency advantage comes at the cost of underutilizing parallel computing resources, as the parallelism of GPUs remains largely idle during sequential processing. On the other hand, if we use a larger batch, we can make better use of the parallelism, as GPUs can be occupied by large-scale matrix computations. As a result, we can process more tokens in the same period of time and the throughput is improved. However, since the result is obtained only when the last token in the batch is predicted, the latency would be higher.

In practice, we usually prefer to use a slightly larger batch, but try to fill the batch with sequences of similar lengths to reduce the number of padding tokens and improve device utilization. For example, we can group the incoming user requests in a short period of time into

[2] See https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf for a simple evaluation.



(a) batch size $= 1$

(b) batch size $= 4$

(c) batch size $= 4$ (similar sequence lengths)

(d) disaggregation of prefilling and decoding

Figure 11.8: Illustrations of basic batching methods. We use a 2D layout to illustrate the batch, where each square represents a token. Red squares indicate tokens in the prefilling stage, blue squares represent tokens in the decoding stage, green squares denote padding tokens, and gray squares correspond to meaningless tokens. Subfigures (a) and (b) compare the cases where the batch size is 1 and 4, respectively. Subfigure (c) shows the strategy of grouping sequences with similar lengths into the same batch. Subfigure (d) illustrates the disaggregation of prefilling and decoding. In this approach, we can make better use of the parallelism of GPUs by concatenating multiple short sequences into a single long sequence for joint processing. This allows us to maximize the number of tokens processed in a batch while minimizing the number of padding tokens. However, as a trade-off, we need to copy the KV cache to the decoding engine and reorganize it after the prefilling phase, which introduces additional data transfer overhead.

buckets, each of which contains sequences with similar lengths. Then, we can fill the batch with sequences in the same bucket, so that we can minimize wasted computational resources, as illustrated in Figure 11.8 (c).

Another approach to implementing batching in LLMs is to disaggregate the prefilling and decoding processes [Wu et al., 2023; Patel et al., 2024; Zhong et al., 2024]. For example, we can perform prefilling on one GPU, and perform decoding on another GPU. One advantage of disaggregation is that we can rearrange the input sequences in the batch to better fill it, because there is no interference between prefilling and decoding. For example, we can concatenate multiple short sequences into a longer one, thus ensuring that the lengths of sequences in the batch are as consistent as possible, as illustrated in Figure 11.8 (d). In this way, we can

maximize the throughput of the prefilling phase. However, as a trade-off, we need to transfer the KV cache to the devices performing decoding, which also incurs extra communication overhead. Typically, this method requires a high-bandwidth, low-latency network to achieve optimal performance.

In this section, we will discuss several improvements to the above basic batching strategies. Most of them are based on an aggregated architecture, that is, decoding and prefilling can be considered as different stages of a model executed on the same device.

1. Scheduling

A practical LLM inference system typically consists of two components:

- **Scheduler**. Its primary role is to efficiently queue and dispatch tasks (i.e., input sequences) to the inference engine based on the current system load and task priorities. This often involves a variety of batching strategies that group certain requests together to maximize processing efficiency in some way.
- **Inference Engine**. It is responsible for the actual execution of the LLMs, processing the queued requests as they come in. As discussed previously, this engine involves both prefilling and decoding processes.

This architecture is illustrated in Figure 11.9. Incorporating scheduling into batch processing provides a flexible way to optimize both the system's throughput and latency, thereby achieving a better balance between them. For example, the batching methods shown in Figure 11.8 (a) and (b) can be considered one of the simplest scheduling strategies, called **request-level scheduling**. In this strategy, once a batch is filled and sent to the engine, the processing of the entire batch cannot be interrupted. The scheduler waits for this batch to be processed before handling the next batch [Timonin et al., 2022].

A more sophisticated scheduling strategy, called **iteration-based scheduling**, interacts with the inference engine at each token prediction step rather than at the sequence level. This approach allows dynamic batch adjustment during inference, as illustrated in Figure 11.10. Such fine-grained control lets the system prioritize critical tokens or sequences in real-time. For instance, if an urgent request arrives at some decoding step, the scheduler can add this request into the batch so that it can be processed as early as possible. In the following subsections, we will discuss batching methods based on iteration-based scheduling.

2. Continuous Batching

Continuous batching is an iteration-based scheduling method used in the Orca system [Yu et al., 2022]. In this method, an iteration refers to either the entire prefilling procedure or a single decoding step. For example, given an input sequence $\mathbf{x} = x_0 ... x_m$ and an output sequence $\mathbf{y} = y_1 ... y_n$, there are $n+1$ iterations in total: one for prefilling, and n for generating the output tokens (one per token). During scheduling, the batch can be adjusted between iterations. For example, we can either add a new input sequence to the batch, or remove a complete sequence from the batch at some iteration, even if the batch processing is not yet

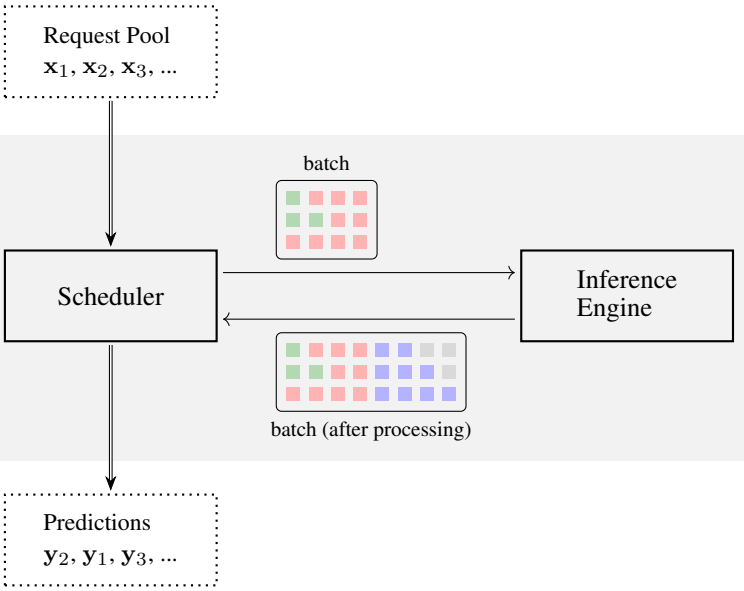


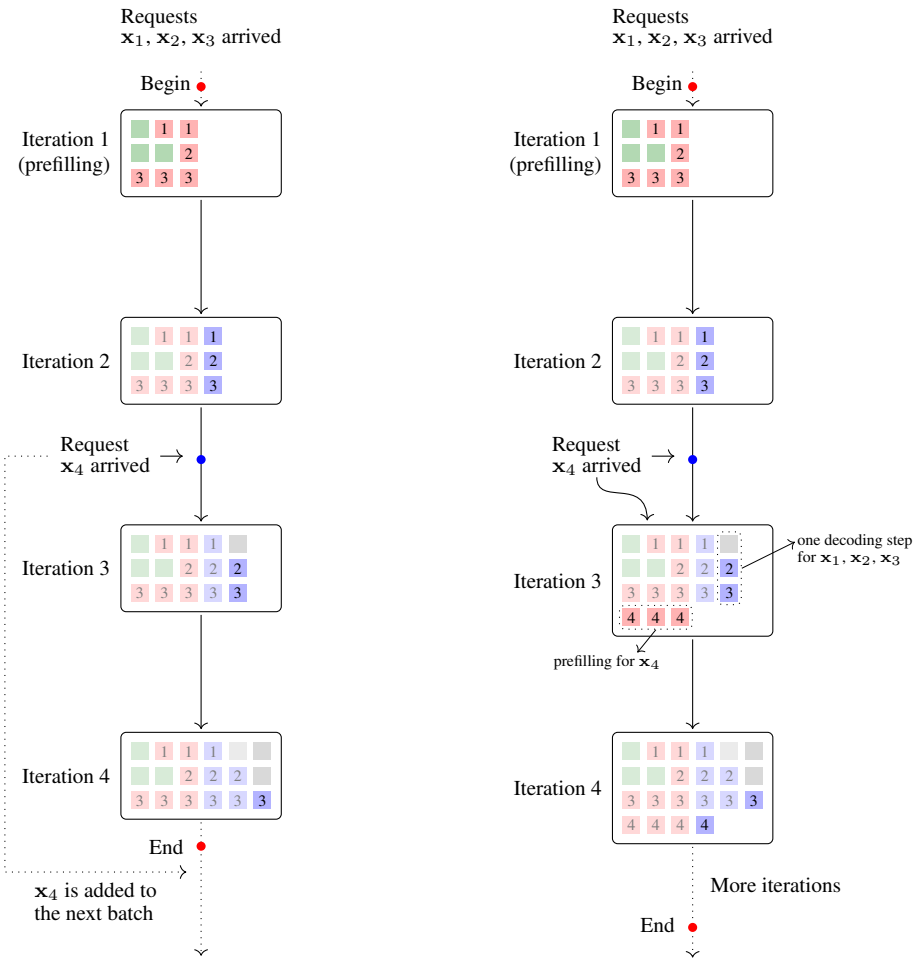
Figure 11.9: Illustration of the LLM inference architecture involving a scheduler and an inference engine. Each time, the scheduler selects a number of user requests to form a batch and sends it to the inference engine. The scheduler can interact with the inference engine and adjust the batch at certain points during inference, such as at the beginning of batch processing and at the start of each token prediction.

finished.

The general process of continuous batching includes the following steps:

- Initially, a batch is created with one or more input sequences, based on both the inference engine's processing capacity and the current user requests. The batch is then fed into the inference engine.
- The inference engine processes the batch iteration by iteration. After each iteration, the scheduler may adjust the batch in one of the following ways:
  - If a sequence in the batch completes generation (i.e., generates the end-of-sequence symbol), that sequence is removed from the batch.
  - If a new user request arrives and the inference engine has additional processing capacity, it is added to the batch.
  - If no sequences are added to or removed from the batch, the batch remains unchanged.
- The processing terminates only when all sequences have been completed and no new user requests arrive.

See Figure 11.11 for an example of continuous batching. In this example, we start with two user requests, $\mathbf{x}_1$ and $\mathbf{x}_2$. These two sequences are packed into a batch and sent to the inference engine for processing. After the engine completes two iterations, a new user request, $\mathbf{x}_3$, arrives. At this point, the scheduler adjusts the batch by adding $\mathbf{x}_3$ to it. The inference engine



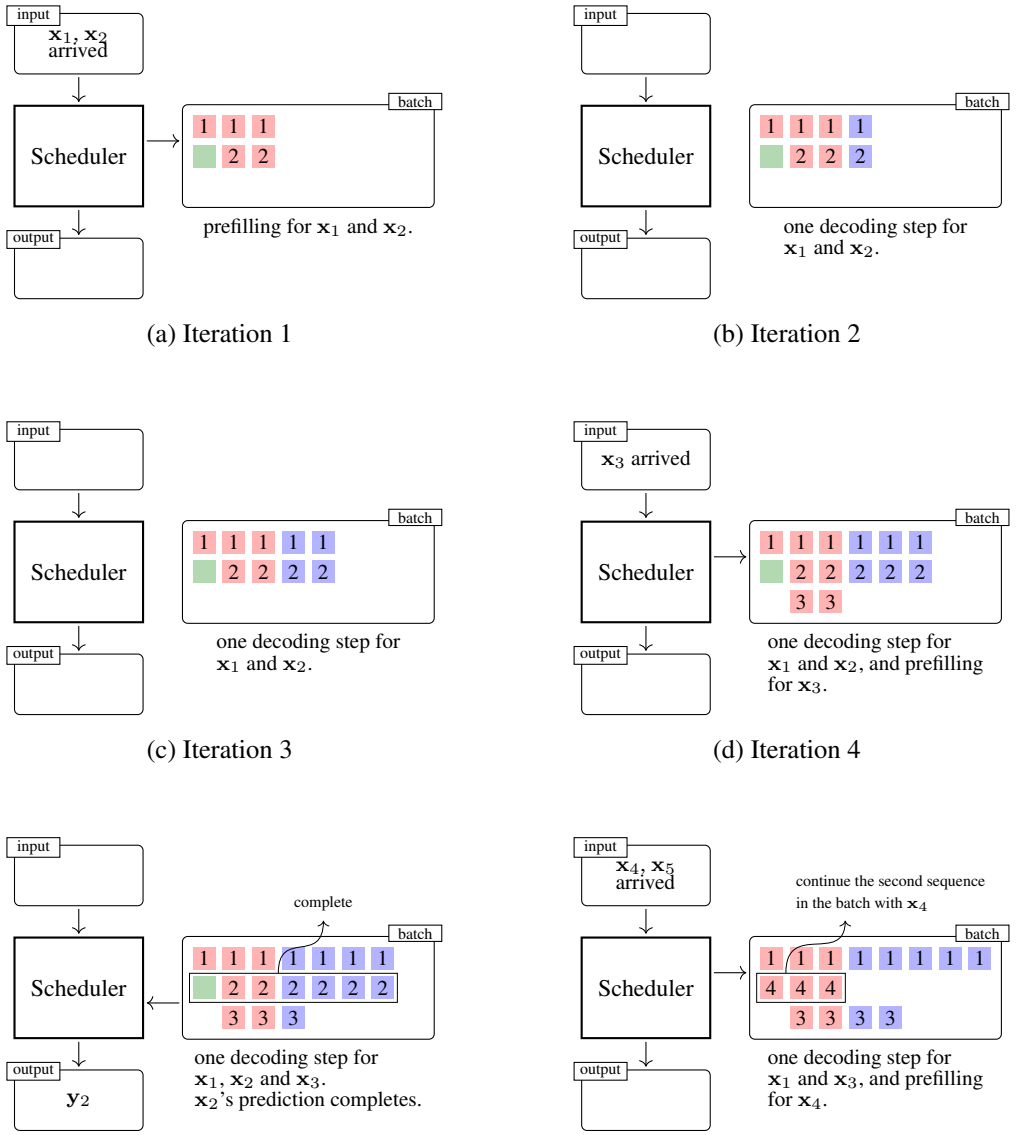
(a) Request-level Scheduling

(b) Iteration-level Scheduling

Figure 11.10: Illustrations of request-level scheduling and iteration-based scheduling. In request-level scheduling, once a batch is created and sent to the inference engine, we cannot adjust the batch. In other words, scheduling only occurs after the processing of a batch finishes. In iteration-level scheduling, we can perform scheduling during batch processing. For example, if a new request arrives at some point during inference, we can add it to the batch and continue processing.

then continues processing the updated batch. Note that the inference engine now processes different sequences in different ways: $\mathbf{x}_1$ and $\mathbf{x}_2$ proceed with the decoding process (i.e., predicting the next tokens), while $\mathbf{x}_3$ undergoes the prefilling process. After some time, the generation for $\mathbf{x}_2$ completes. As it happens, two more user requests, $\mathbf{x}_4$ and $\mathbf{x}_5$, arrive. The scheduler removes the completed sequence $\mathbf{x}_2$ from the batch and, considering the current load of the inference engine, adds $\mathbf{x}_4$ to the batch. However, $\mathbf{x}_5$ must wait until another sequence in the batch finishes before it can be added.

The idea behind continuous batching is to keep the inference engine fully utilized by



(a) Iteration 1

(b) Iteration 2

(c) Iteration 3

(d) Iteration 4

(e) Iteration 5

(f) Iteration 6

Figure 11.11: Illustration of batch adjustment in continuous batching. Instead of fixing a batch of input sequences and processing them to completion (as in request-level batching), continuous batching dynamically updates the batch during inference. The system continuously accepts and adds new requests (e.g., $\mathbf{x}_3$ and $\mathbf{x}_4$) into the current batch as long as there is available compute capacity.

processing as many sequences as possible, thereby maximizing computational resource usage. A key difference between continuous batching and standard batching (see Figure 11.8) lies in the fact that, in continuous batching, prefilling and decoding can occur simultaneously across different sequences, whereas in standard batching, these two phases are performed sequentially for the entire batch. As discussed in Section 11.1.2, prefilling is considered a

compute-bound process, while decoding is considered a memory-bound process. The intuition behind overlapping prefilling and decoding is to reduce idle times for both computation and data transfer. Consider two mini-batches: one for prefilling and one for decoding. While the prefilling mini-batch keeps the GPUs occupied, the decoding mini-batch can perform memory transfers concurrently.

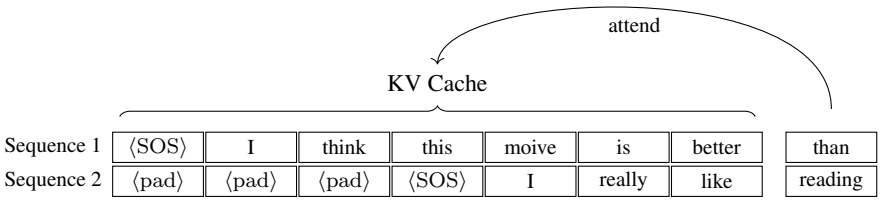
Another difference between continuous batching and standard batching is that continuous batching is prefilling-prioritized, while standard batching is decoding-prioritized [Agrawal et al., 2024]. In continuous batching, once the inference engine has spare computational resources, the scheduler will add new requests to the batch. In other words, these newly added requests will be processed for prefilling as early as possible. This approach improves system throughput, but at the cost of increased latency, as the newly added requests extend the processing time of earlier ones. In contrast, in standard batching, once the batch is created, we must wait for the last sequence in the batch to complete before processing new requests. This ensures relatively low latency, but results in lower device utilization and system throughput.

It is important to note that the cost of continuous batching is that we need to continuously reorganize the batches, which involves rearranging the data in memory. Each time a new request is added, the scheduler needs to reassess and optimize the current batch structure. This dynamic adjustment can incur additional memory and computational overhead, especially when the batches are frequently adjusted. Therefore, while this method can improve throughput, it may also lead to increased memory fragmentation and, in some cases, introduce additional latency.

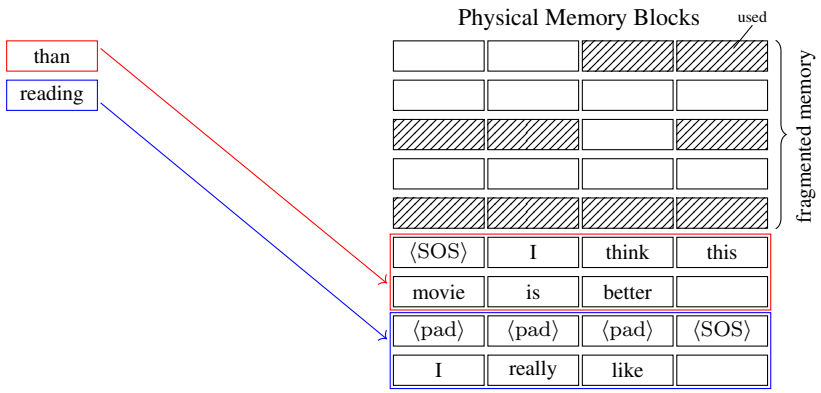
3. PagedAttention

PagedAttention (or paged KV caching) is a technique used in the vLLM system [Kwon et al., 2023]. Inspired by operating system paging, it optimizes memory usage during LLM inference — particularly for the KV cache — by addressing fragmented memory allocation in dynamic batching scenarios with variable-length sequences. The idea behind PagedAttention is to break down large memory requirements for KV caching into more manageable "pages" or chunks of memory. In this way, we do not need to store the KV cache of the full sequence in a continuous memory. Instead, the KV cache is divided into fixed-size blocks (analogous to memory pages in an operating system), which can be non-contiguously allocated in physical memory. One advantage of PagedAttention is that it enables flexible memory management, supporting dynamic sequence growth without requiring expensive reallocation or copying of large contiguous memory regions. Note that PagedAttention is not specifically designed for batching. But it indeed helps improve memory efficiency in batched inference scenarios, where memory management is more demanding and complicated.

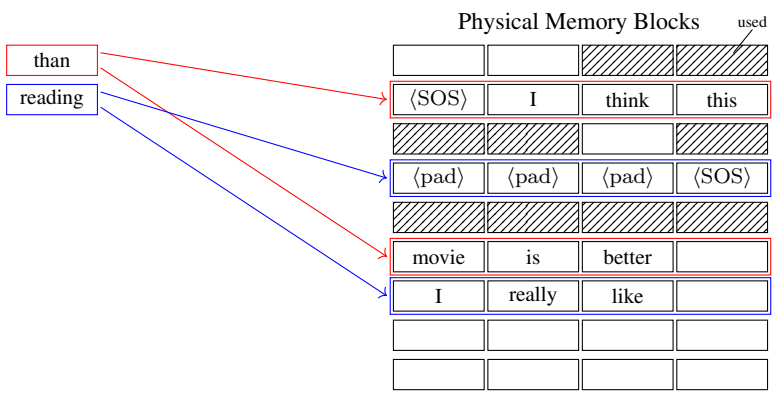
Consider a simple example of memory allocation in Figure 11.12 in which self-attention is performed for a batch consisting of two sequences. For each sequence, we need to attend the current token to the key-value pairs in the KV cache of this sequence, as required by self-attention. In the standard implementation of self-attention, the KV cache is stored in a contiguous block of memory, allowing us to efficiently access this continuous memory. However, in a paged KV caching system, the KV cache is divided into smaller, fixed-size



(a) Two sequences in a batch



(b) Memory allocation for KV caching in standard self-attention



(c) Memory allocation for KV caching in PagedAttention

Figure 11.12: Illustration of memory allocation in PagedAttention. There are two sequences in the batch, as illustrated in sub-figure (a). Since the memory is fragmented, the KV cache is stored in a large unused block of memory in standard self-attention (see sub-figure (b)), but the fragmented memory is not used. By contrast, in PagedAttention (see sub-figure (c)), the KV cache is divided into smaller blocks and thus fits into fragmented memory.

memory blocks which are not necessarily contiguous. These smaller KV cache blocks can be more effectively allocated to fragmented memory regions, thereby improving memory utilization. Another benefit of distributing chunks of the KV cache across different memory blocks is that it enables parallelization of the caching process. For example, if the input sequence is long and the memory bandwidth is sufficient, it would be beneficial to write and read the key and value vectors of different segments of the sequence in parallel across multiple memory blocks.

In general, storing contiguous data in non-contiguous regions can cause issues, for example, accessing fragmented data requires additional seek time, which reduces I/O efficiency. However, when handling large-scale data (e.g., performing multiplication on extremely large matrices), we typically do not process all the data at once but instead divide it into smaller blocks for block-level computation. From this perspective, it is also reasonable to partition the attention computation. If the paging strategy is well designed, the additional overhead in memory access can be minimal, while the improvement in memory utilization can be significant.

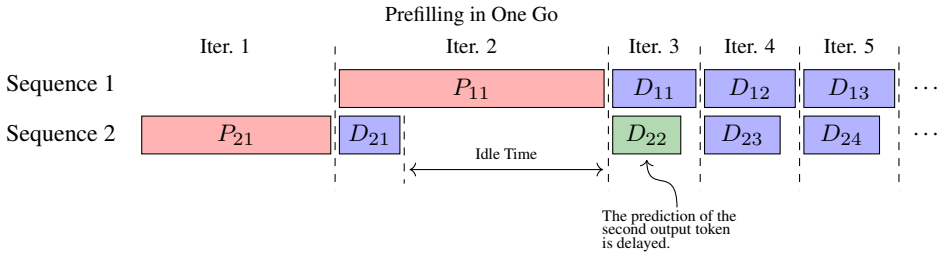
4. Chunked Prefilling

We have seen that, in iteration-level scheduling, prefilling and decoding for different sequences can occur simultaneously. This can be seen as a prefilling-prioritized strategy which can maximize the throughput. However, one such iteration can take a long time if the input sequence is very long and the prefilling process dominates the computation. In this case, decoding for other sequences has to wait until the prefilling completes, leading to increased latency for generating output tokens. Therefore, while prefilling-prioritized strategies are effective for maximizing hardware utilization, they may introduce significant variability in token generation latency, particularly when the system is handling a mix of long and short input sequences.

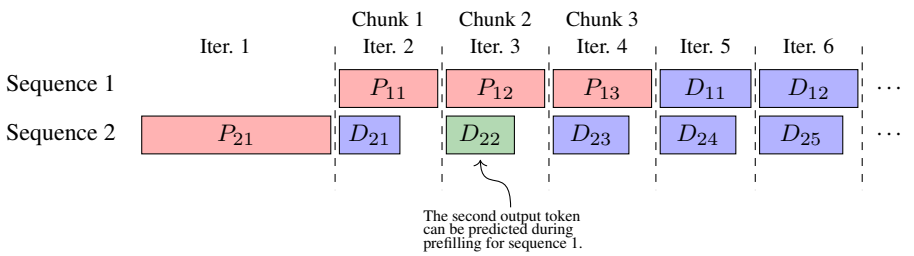
A simple way to reduce decoding latency is to make computations for different sequences in the batch comparable. One such method is to divide sequences into chunks and perform prefilling chunk by chunk. This approach, often referred to as chunked prefilling, processes smaller portions of each sequence at a time, allowing the system to better balance the computational load across sequences [Agrawal et al., 2023]. By choosing an appropriate chunk size, we can ensure that when prefilling and decoding overlap for two sequences, their processing within the same iteration tends to take a similar amount of time. As a result, decoding idle time is reduced and overall throughput is improved.

Figure 11.13 shows an illustration of chunked prefilling in a few iterations. In this example, the batch contains two sequences. The whole prefilling process of the first sequence is divided into three prefilling steps, giving rise to the chunks denoted P_{11}, P_{12} and P_{13}. Each chunk corresponds to one iteration and can thus overlap with one decoding step. In this way, during the prefilling of the first sequence, we can perform three decoding steps, rather than only a single decoding step, as is the case in standard iteration-level scheduling. As a result, the idle time of the decoding process is reduced, and the output tokens can be generated earlier.

Chunked Prefilling improves decoding efficiency by overlapping prefilling and decoding,



(a) Simple Iteration-level Scheduling



(b) Chunked Prefilling

Figure 11.13: Comparison of simple iteration-based scheduling and chunked prefilling. P_{xy} denotes the y-th prefilling step for sequence x, and D_{xy} denotes the y-th decoding step for sequence x. In simple iteration-based scheduling (or prefilling-prioritized scheduling), since prefilling is treated as a single iteration, D_{22} has to wait for the completion of the prefilling of sequence 1. In chunked prefilling, the prefilling process can be divided into multiple steps. Thus, D_{22} can execute during prefilling for sequence 1 (i.e., during P_{12}).

but at the cost of additional memory overhead and scheduling complexity. In standard prefilling, we process the whole input sequence once, building the KV cache in one go. By contrast, in chunked prefilling, each chunk needs a separate forward pass to compute its attention outputs and update the KV cache. As a result, we need to maintain the KV cache of early chunks while processing later chunks. This also compromises the parallelism of completing the prefilling for the entire sequence in a single pass. In practice, it is usually possible to balance throughput and latency by choosing an appropriate chunk size.

It is worth noting that the methods discussed in this subsection can broadly be categorized as priority-based scheduling methods. In these methods, we can give priority to certain requests, or to certain prefilling or decoding steps, so that system resources are allocated in a way that better aligns with specific performance goals. As presented above, for example, we may prioritize decoding over prefilling to minimize token generation latency, or prioritize prefilling over decoding to maximize overall throughput in batch-processing scenarios. Practitioners can design custom priority policies for specific needs and operational constraints in real-world applications, such as request deadlines and importance levels defined by users.

11.2.3 Parallelization

Parallelization is a widely used approach to scale up LLM inference, especially for large-scale deployments. In Chapter 7, we have discussed several common parallelization strategies to parallelize LLM pre-training, such as model parallelism, tensor parallelism, and pipeline parallelism. We have also discussed efficient architectures that are easy to deploy in distributed computing systems. For example, in MoE models, we assigns different experts to different devices[3]. Only the active experts for a given input are executed, which significantly improves computational efficiency while maintaining model quality. Many of these methods can be directly applied to LLM inference with minimal modifications.

However, applying these parallelization techniques to inference poses new challenges compared to pre-training. These issues become especially pronounced in real-time or low-latency inference scenarios, where load imbalance across devices and communication overhead can significantly impact performance. For example, unlike pre-training, where batches can be prepared in advance, inference must handle variable-length sequences in real time. This makes it harder to maintain optimal device utilization and complicates scheduling across heterogeneous computational resources. A related issue is load balancing. When a large number of requests arrive in a short period of time, the system must efficiently distribute workloads across available devices. For example, real-world requests typically exhibit highly variable computational demands due to differences in task types and prompt lengths. Such variability renders simple static load balancing approaches ineffective, and so we need to use finer-grained strategies that can adapt to runtime conditions. The problem becomes even more complicated when we deploy the system on heterogeneous hardware and there are strict latency constraints.

In the development of LLMs, parallelization is closely related to LLM serving. Generally, building a high-quality LLM serving system is not a simple task — it typically requires the combination of multiple techniques, such as architectural design, workload distribution, and LLM-specific hardware/software optimizations. As such, LLM serving constitutes an exceptionally broad subject that often demands substantial engineering expertise. Here, we will not go into the details of LLM serving. For related concepts and techniques, readers may refer to relevant open-source systems (such as vLLM[4], TensorRT-LLM[5] and TGI[6]) and papers [Pope et al., 2023; Li et al., 2024].

11.2.4 Remarks

We have considered many methods for improving the efficiency of LLMs in this and previous chapters. Although these approaches address different issues, most of them essentially explore trade-offs between various performance factors. One important trade-off is between inference speed and accuracy. For example, techniques like quantization, pruning, and knowledge

[3] In LLMs, the experts are typically modular FFNs. So each expert is a part of the FFN component in the Transformer architecture.

[4] https://github.com/vllm-project/vllm

[5] https://github.com/NVIDIA/TensorRT-LLM

[6] https://github.com/huggingface/text-generation-inference

distillation can significantly reduce computational overhead and latency but may introduce minor degradations in model performance. Conversely, preserving full precision or using larger models enhances accuracy but at the cost of slower inference and higher resource demands.

Another important consideration in LLM inference is the memory-compute trade-off. As in computer system design, we need to consider the balance between memory usage and computation required to generate the output. In particular, storing intermediate results such as KV caches during inference can significantly reduce redundant computation, but at the cost of increased memory usage. In KV caching, storing past attention states avoids recomputation of self-attention over previous tokens, thereby reducing compute time per token. However, as the number of tokens grows, so does the memory footprint of the KV cache, especially when processing very long sequences or multiple sequences in parallel. In response, various techniques have been developed to reduce memory consumption by partially recomputing intermediate states. For instance, chunked or windowed attention limits the attention span to a recent subset of tokens, reducing KV cache size at the cost of reduced context or additional compute if past information must be reprocessed.

Note that considering the memory-compute trade-off is a very general principle. It can be extended beyond attention mechanisms and Transformers to other components in system design. An example is the choice of data precision. Using lower-precision formats such as FP16 or INT8 can reduce both memory usage and memory bandwidth requirements, effectively alleviating pressure on the memory subsystem. However, lower precision may lead to numerical instability or slight accuracy degradation, requiring careful calibration or retraining. Thus, this trade-off can also be seen as a memory-compute-accuracy triangle, where improvements in one dimension may come at the expense of another.

Beyond speed, accuracy, and memory, several other dimensions also influence LLM inference efficiency. Some of these dimensions have been discussed in this chapter, while others have not. Here we outline them as follows.

- **Throughput vs. Latency**: In large-scale multi-user LLM serving scenarios, we often aim to maximize system throughput. For example, as discussed in this section, we can batch multiple requests together to increase the number of tokens processed at the same time. However, batching increases waiting time and may lead to higher per-request latency, especially for short or interactive requests. By contrast, optimizing for low latency often requires serving requests individually or in smaller batches, which underutilizes hardware resources and reduces throughput. Achieving a good balance depends on the quality-of-service requirements and user interaction patterns.

- **Generalization vs. Specialization**: General-purpose LLMs are trained to perform a wide range of tasks with a single set of parameters. While flexible, they may be less efficient or accurate for specific tasks. Specialized models can yield better performance and lower inference costs for targeted applications. However, maintaining multiple specialized models increases system complexity and storage requirements. The trade-off between maintaining a single general model versus multiple specialized models is an important system-level design choice.

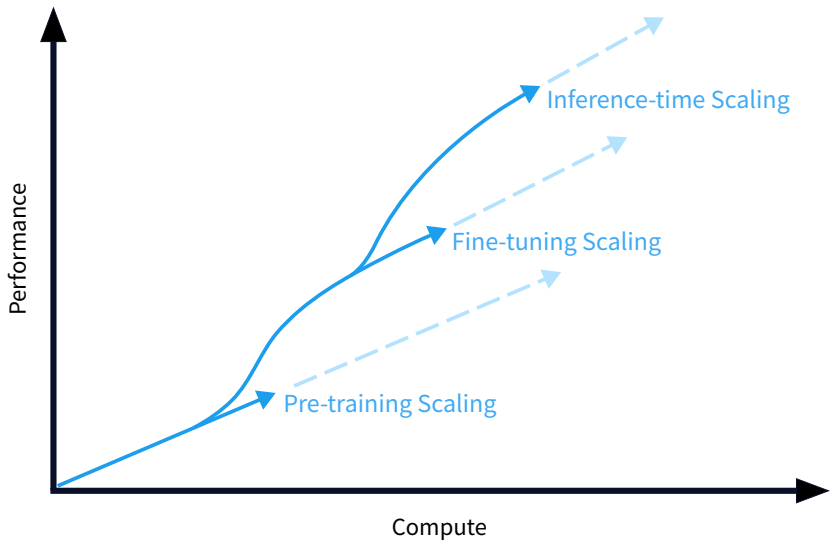


Figure 11.14: Scaling for pre-training, fine-tuning and inference stages [Briski, 2025].

- **Energy Efficiency vs. Performance**: High-performance inference often requires running large models at high throughput on powerful accelerators, which consumes considerable energy. This may be problematic for edge deployments or energy-sensitive environments. Techniques like model compression can improve energy efficiency, but usually with some degradation in output quality or increase in latency. Energy constraints thus introduce another important dimension in optimizing LLM inference.

11.3 Inference-time Scaling

Scaling laws can be considered one of the fundamental principles guiding the development of LLMs. In previous chapters, we discussed several times that scaling up training data, model size, and compute can effectively improve the performance of pretraining. In fact, scaling laws also apply to downstream stages such as fine-tuning and inference (see Figure 11.14). Here we consider **inference-time scaling**, which has been widely employed by recent LLMs to solve complex problems, such as complex math problems [Snell et al., 2025]. Unlike pre-training and fine-tuning scaling, which focuses on improving LLMs via parameter updates, inference-time scaling improves these models during inference without further training. This includes a large variety of methods which scale LLMs in different dimensions, such as ensembling multiple model outputs, increasing context length, adopting more aggressive decoding algorithms, and using external tools to extend model capabilities.

While inference-time scaling is wide-ranging, in this section we consider those methods that incorporate more compute into inference (called inference-time compute scaling). Here is

a list of inference-time (test-time) compute scaling methods, organized by category:

- **Context Scaling**. It involves scaling the input or context to improve generation (or potentially scale the output).
- **Search Scaling**. It involves increasing computational effort during decoding.
- **Output Ensembling**. It involves combining multiple model outputs.
- **Generating and Verifying Thinking Paths**. It involves guiding LLMs to generate and verify thinking paths for solving complex reasoning problems.

We will describe these methods in the following subsections.

11.3.1 Context Scaling

Context scaling improves LLM performance by extending the input to the model. A straightforward approach is to incorporate more helpful context during inference, allowing the model to condition its predictions on more prior information. One example is few-shot prompting. It augments the context with multiple input-output examples, and so the model can learn task behavior implicitly from these examples without parameter updates. On top of few-shot prompting, we can use chain-of-thought prompting to encourage the model to produce intermediate reasoning steps before final answers. Note that chain-of-thought prompting is one of the most important methods in addressing reasoning problems. By explicitly providing intermediate steps in problem-solving, we can prompt the model to break down complex tasks into simpler sub-tasks, which is found to be very beneficial for generating accurate and interpretable outputs.

Beyond extending the prompt with examples or reasoning steps, another approach to context scaling involves dynamically incorporating external knowledge. This is often achieved through RAG. RAG systems first retrieve relevant document snippets from a large collection of documents or a database based on the current input. These retrieved pieces of information are then added to the context provided to the LLM. This essentially expands the context to include timely or specialized external knowledge. By doing so, the model grounds its responses in specific knowledge found in the external source. The LLM thus can generate responses that are not only relevant to the input but also factually accurate and up-to-date.

However, as the context grows, these methods often suffer from the constraints of finite context window length. While model architectures and techniques (like efficient attention models) are continually evolving to support longer contexts, processing extremely long inputs still poses challenges. Increased computational cost is one factor. More critically, when the context window becomes very large, the model might struggle to attend effectively to the most relevant information (e.g., the "lost in the middle" phenomenon). Therefore, effective context scaling is not just about adding more information, but also about strategically selecting, structuring, and presenting the most pertinent information within the model's processing capabilities.

Here we omit the detailed discussion of these methods, as they have already been covered in previous chapters. See Chapters 8 and 9 for more details, including prompting, RAG, and

long-sequence modeling methods.

11.3.2 Search Scaling

In LLMs, decoding is a search process that aims to efficiently find the best output sequence given the input sequence. Search scaling (or decoding scaling) typically involves two aspects: scaling the output length and scaling the search space.

Scaling the output length refers to increasing the number of tokens generated during inference. This is especially important in tasks that require long-form generation, such as story writing. More recently, generating outputs with long thinking paths has shown strong performance in math problem solving and code generation. For example, encouraging the model to generate long thinking paths before producing the final answers has been found to be very beneficial in performing complex reasoning. This idea has been widely used in developing recent LLMs for reasoning, such as OpenAI [2024]'s o1 and Deepseek [2025]'s R1. We will discuss more about output length scaling in Section 11.3.4.

Scaling the search space, on the other hand, refers to expanding the set of candidate output sequences considered during search, so that higher-quality outputs can be found. As discussed in Section 11.1.3, a simple example is that in beam search we increase the beam width to allow more candidate sequences to be explored in parallel at each decoding step. This increases the chance of discovering better outputs, especially in tasks where the optimal solution is not immediately apparent from local decisions.

In addition to decoding algorithm adjustments, it is also possible to explore compact structures to encode a large number of outputs. For example, we can construct and navigate a tree or graph of reasoning steps [Yao et al., 2024]. In this paradigm, each node represents a partial solution or intermediate step, and edges represent transitions between reasoning states. Such structured search enables the model to consider multiple paths simultaneously. Another related direction is Monte Carlo tree search-inspired decoding, where the model stochastically explores and scores different paths based on learned heuristics or external reward models.

Search scaling is a very general idea, and it is often implicitly involved in the design of search procedures that exploit search structure, heuristics, and model uncertainty. Many of the above methods have been discussed previously, though they were not originally developed with scaling as their primary goal. However, search scaling inherently comes with computational costs. Increasing beam width, for instance, directly translates to higher memory usage and longer inference times. In practice, there is often a point of diminishing returns, where further expansion of the search space yields marginal improvements in output quality at a significant computational expense. Therefore, an effective strategy often involves finding an optimal balance between scaling and computational feasibility.

11.3.3 Output Ensembling

If we have multiple model outputs, it is often beneficial to combine them to mitigate the impact of individual model errors and synthesize a superior final output. Each model might capture different aspects of the underlying data distribution or possess unique strengths and weaknesses. By ensembling, we can average out the noise or random errors present in individual predictions,

leading to a more stable and reliable outcome. In LLM ensembling, one of the simplest approaches is to average the probability distributions over the next token from each model, and select the best token using this averaged distribution. Or, if we regard the problem as a discrete decision-making task, majority voting can be employed. More sophisticated methods might involve re-ranking candidate outputs generated by different models based on a separate scoring function or even using a meta-learner to intelligently combine the predictions.

The "scaling" from output ensembling comes at the cost of running multiple models or sampling multiple outputs. This not only increases the latency of inference but also leads to the additional complexity of managing multiple models. But the quality of outputs does not continue to improve indefinitely as more models are added. In some cases, the benefits of output ensembling may diminish as the number of component models in the ensemble exceeds a certain threshold. Instead, the benefits of ensembling are generally greater when the individual models are diverse (i.e., they make different errors), even if there are a relatively small number of component models. Therefore, it is common practice to use a set of diverse LLMs which differ in their training data, model architectures, or fine-tuning objectives.

In LLMs, "scaling" often implies making things "bigger" for quality with more resources. However, in addition to scaling up the quality, scaling can mean more. It can also signify scaling up the robustness (making the system less prone to errors and more reliable) and exploration (covering a wider range of potential solutions). In output ensembling, these dimensions are naturally integrated. For instance, the very act of averaging or voting across different model outputs is a direct strategy to scale up robustness against individual model failures. Furthermore, by intentionally including varied models, ensembling increases the chances of discovering novel or superior solutions. In this sense, scaling is not limited to making models larger or running them longer — it also means strategies for making inference more robust, exploratory, and adaptive.

11.3.4 Generating and Verifying Thinking Paths

So far, we have viewed inference-time scaling as a general class of methods for scaling various aspects of inference, such as sequence length, model size, and/or search strategies. In fact, one successful application is the use of inference-time scaling to enhance the reasoning capabilities of LLMs. As we have seen, the reasoning performance of LLMs can be improved by using chain-of-thought methods. We can therefore make use of the chain-of-thought prompts to generate intermediate reasoning steps and reach a correct answer. However, reasoning problems are often so complicated that we cannot obtain high-quality solutions by providing simple chain-of-thought prompts. For example, when solving a math problem, we typically need to reason over a sequence of steps. At each step, we need to work out some intermediate result, verify it, and then determine what to do next. The reasoning path is not a fixed pattern but a dynamically generated thinking process that often involves trial-and-error, backtracking, and self-correction. This requires more sophisticated prompting strategies or search algorithms to navigate such complex reasoning. In this subsection, we focus on inference-scaling methods that go beyond simple chain-of-thought to address complex reasoning problems more effectively.

At a high level, methods for scaling the reasoning of LLMs can be categorized into two

classes:

- **Training-free Methods**. These methods aim to improve reasoning capabilities without requiring any modification or retraining of the pre-trained parameters. Instead, they focus on techniques applied during inference, such as sophisticated prompting strategies (e.g., chain-of-thought) and algorithmic control over the reasoning process (e.g., search).
- **Training-based Methods**. These methods involve further training or fine-tuning the model parameters to explicitly improve reasoning abilities, such as supervised fine-tuning on datasets with reasoning examples (e.g., math problems with step-by-step solutions).

In the following, we first discuss training-free methods, and then training-based methods.

1. Solution-level Search with Verifiers

Given an input sequence (e.g., a math problem), there are many possible output sequences (e.g., solutions to the problem). If we have a model to evaluate or verify each solution, we can select the best one. This is the fundamental principle behind methods like best-of-N sampling, where multiple outputs are generated, and the optimal result is picked based on some selection mechanism. Such a selection process can be viewed as a search problem, which involves two components:

- **Search Algorithm**. This defines the strategy used to explore the space of possible output sequences (solutions) and generate a set of candidates. It can range from simple independent sampling to more sophisticated search techniques as discussed in Section 11.1.3.
- **Verifier**. This is a model or function responsible for evaluating the quality, correctness, or utility of each candidate solution generated by the search algorithm. It provides a score, a probability, or a judgment that allows the system to select the best among the candidates. The verifier can be another LLM, or even a set of predefined rules or heuristics.

Given an input problem $\mathbf{x}$, we define that an output solution $\mathbf{y}$ can be represented as a sequence of reasoning steps:

$$\mathbf{y} = (a_1, a_2, ..., a_{n_r}) \tag{11.37}$$

where a_i is the i-th reasoning step, and a_{n_r} is the last step which should contain the answer to the problem. See Figure 11.15 for an example of a multi-step reasoning path.

The search algorithm can efficiently generate a set of candidate solutions

$$\mathcal{D}_c = \{\mathbf{y}_1, ..., \mathbf{y}_K\} \tag{11.38}$$

Then, we can use a verifier, which evaluates each solution by the function $V(\mathbf{y})$, to score

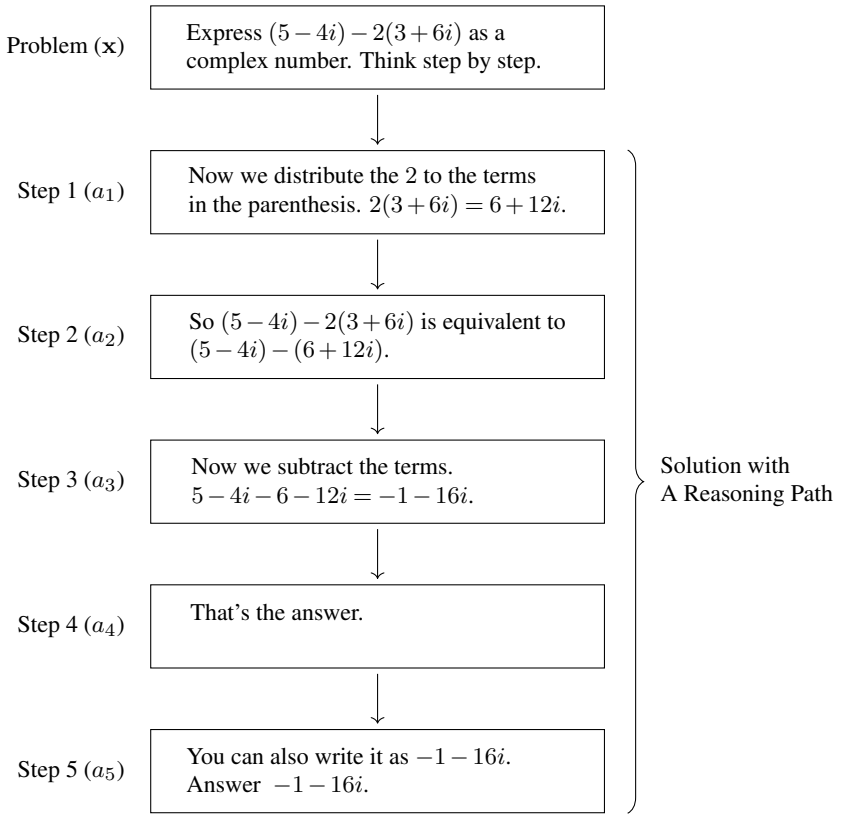


Figure 11.15: Illustration of multi-step reasoning. This example is from the PRM800K dataset [Lightman et al., 2024]. Given a math problem, the LLM is prompted to generate a thinking path (or reasoning path) consisting of several reasoning steps. Each step addresses a sub-problem based on the results of the previous steps. The answer to the original problem is contained in the last step.

the candidates in $\mathcal{D}_c$. The final output is the best candidate selected by the verifier

$$\hat{\mathbf{y}} = \underset{\mathbf{y} \in \mathcal{D}_c}{\arg\max}\, V(\mathbf{y}) \tag{11.39}$$

Although verifying the entire reasoning path is possible, a simpler alternative is to verify only the final reasoning step. In this way the verifier function $V(\mathbf{y})$ is simplified to depend solely on the final answer contained within a_{n_r}. This can be achieved in various ways, depending on the nature of the problem and the expected answer format.

- For some math and coding problems, we can use off-the-shelf tools as verifiers. Examples include proof checkers for mathematical theorems, interpreters or compilers for code execution, and unit test systems for verifying program correctness against predefined test cases.
- If there is labeled data for evaluating the answer, such as human preference data, we can train a reward model on such data. The learned reward model is then used as the verifier

which assigns a scalar score to each candidate answer.

- If there are no existing systems or suitable reward models, we can use another LLM to act as the verifier. This LLM is prompted to assess the quality of the candidate answer. It could potentially be a more capable model, or the same LLM used with a specific "evaluator" prompt.
- Alternatively, simpler heuristic-based verifiers can be designed. A commonly used approach is to employ majority voting, where the most frequently occurring answer among a set of candidates is selected.

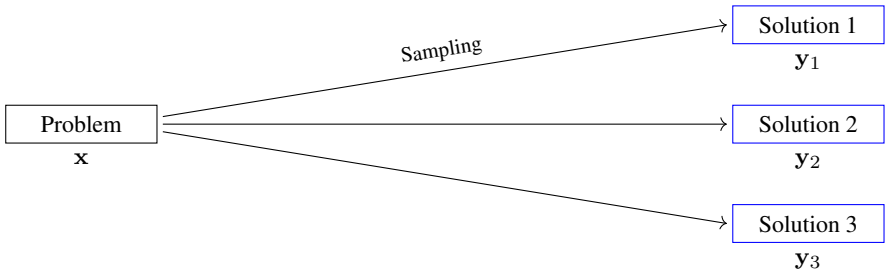
Based on these verifiers, we can search to obtain a set of candidate solutions for selection. One simple strategy, which is often referred to as **parallel scaling** [Brown et al., 2024; Snell et al., 2024], involves generating K candidate solutions by running the base LLM K times independently. In this process, we can adjust the temperature in sampling to control the diversity in the outputs. The verifier then assesses each of these K complete solutions, and the one with the highest score is selected as the final output. This is conceptually very similar to best-of-N sampling, which in previous chapters we primarily described as a method of selecting the best one from a set of sampled outputs using a reward model.

Another approach is **sequential scaling**, which builds a sequence of solutions incrementally [Gou et al., 2024; Zhang et al., 2024]. It starts with an initial solution generated by the LLM with prompting. Then, we use a verifier (often the same LLM) to evaluate the solution. This can be seen as a critique stage. The output of this stage is some form of feedback, such as textual critiques pinpointing errors or suggesting improvements, numerical scores reflecting solution quality, or even a revised plan or intermediate step to guide the next generation. This feedback, along with the original problem and the current solution, is then used to prompt the LLM to generate a potentially improved solution. This can be seen as a refine stage. This critique-refine cycle can be repeated, forming an iterative loop:

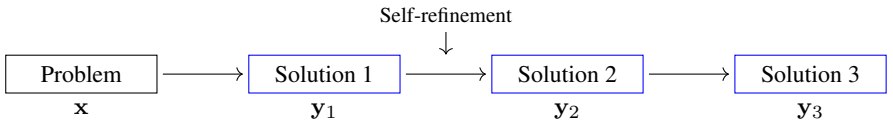
$$\mathbf{y}_{k+1} = \text{Refine}(\mathbf{x}, \mathbf{y}_k, \text{Feedback}(\mathbf{y}_k)) \tag{11.40}$$

where $\text{Feedback}(\mathbf{y}_k)$ represents the feedback from the verifier. The $\text{Refine}(\cdot)$ function generates the improved solution $\mathbf{y}_{k+1}$ by prompting the LLM with the original problem $\mathbf{x}$, the previous solution $\mathbf{y}_k$, and this feedback. The process can be iterated for K times, or until the solution quality, as assessed by the verifier, converges to a satisfactory level. This iterative framework, where a solution is progressively improved through cycles of generation, evaluation (critique), and revision, is precisely what constitutes self-refinement [Shinn et al., 2023; Madaan et al., 2024]. In such scenarios, the role of the verifieris not just to pick the best complete solution from a static set, but to actively guide the generation process itself.

See Figure 11.16 for illustrations of parallel scaling and sequential scaling. Note that there are other ways to perform search and obtain different sets of candidate solutions. One alternative method is to organize search as a tree structure. This approach, often referred to as tree search, provides a more structured way to explore the space of possible reasoning paths. In solution-level search, each node of the tree represents a complete solution. During search, we need to expand a node to a set of child nodes, representing new solutions that can be considered



(a) Parallel Scaling



(b) Sequential Scaling

Figure 11.16: Illustrations of parallel scaling and sequential scaling. In parallel scaling, we obtain multiple solutions by running the LLM several times independently. In sequential scaling, the LLM generates an initial solution. Then, we use the LLM to refine it iteratively, with each refinement yielding a new, possibly better solution.

in verification. The expansion process typically involves taking an existing solution (the parent node) and using the LLM to generate variations or alternative solutions.

2. Step-level Search with Verifiers

While the methods discussed above primarily focus on generating complete solutions before final selection, the search process can also be integrated more deeply into the step-by-step generation of the reasoning path itself. This leads to approaches that perform step-level search with verifiers, where guidance or pruning occurs at intermediate reasoning steps $\{a_1, ..., a_{n_k}\}$ rather than only after a full solution $\mathbf{y}$ is formed.

Such fine-grained control is particularly beneficial for complex reasoning problems where a single incorrect intermediate step can render the entire subsequent reasoning chain invalid. By evaluating or guiding the generation at each intermediate step, the LLM can explore the reasoning space more effectively, potentially pruning unpromising paths early or allocating more resources to explore more plausible ones.

Step-level search with verifiers can also be modeled as a tree search problem. In this paradigm, each node (or state) corresponds to a partial reasoning path, $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$, representing the sequence of i reasoning steps taken so far (i.e., a path from the root node to the current node). The objective of the search process is to explore the underlying state space, starting from an initial empty path, to find a complete path that constitutes a correct solution. Note that we use $\mathbf{a}_{\leq i}$ here to represent a partial reasoning path instead of $\mathbf{y}_{\leq i}$. While this makes notation a bit inconsistent with that used for representing complete solutions ($\mathbf{y}$) or full

paths in solution-level search, it serves to highlight the focus on individual actions or steps.

The core components of step-level search with verifiers are:

- **Node Representation**. A node is a partial reasoning path $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$. The root node is an empty path, and terminal nodes are complete reasoning paths.
- **Node Expansion**. Given a current partial path $\mathbf{a}_{\leq i}$, the LLM is used to generate one or more candidate next reasoning steps $\{a_{i+1}^{(1)}, ..., a_{i+1}^{(M)}\}$. Each candidate step, when appended to $\mathbf{a}_{\leq i}$, forms a new potential partial path $\mathbf{a}_{\leq i+1} = (a_1, ..., a_i, a_{i+1}^{(j)})$.
- **Verification**. The verifier $V(\cdot)$ evaluates the quality of a newly generated step in the context of the current partial path $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$ and the original problem $\mathbf{x}$. As with solution-level verification, step-level verifiers might output a numerical score, a categorical label, and textual feedback.
- **Search**. This governs how the search space is explored. Based on the evaluations from the verifier, the search strategy decides which partial paths to extend further, which to prune, and the order of exploration.

This step-by-step verification allows for dynamic adjustments to the reasoning process. If a step a_{i+1} is deemed incorrect or unpromising by $V(\cdot)$, the search algorithm can backtrack and explore alternative steps from $\mathbf{a}_{\leq i}$, or even from an earlier node $\mathbf{a}_{\leq i'}$ (where $i' < i$). Conversely, if a step is highly rated, resources can be focused on extending that path. See Figure 11.17 for an illustration of step-level search with verifiers.

Clearly, this search framework is very similar to that used in decoding methods for LLMs, as discussed in Section 11.1.3. For example, beam search maintains a set of K most promising partial sequences at each generation step. This is a form of step-level search where the "verifier" is implicitly the LLM's own probability model, and the "search" is the pruning mechanism to maintain the beam size.

However, step-level search with explicit verifiers, as described here, presents differences from standard decoding. One of them is that the verifier can be a much more sophisticated component than just the raw output probabilities of the generative LLM. The design of step-level verifiers basically follows that of solution-level verification. A step-level verifier might be a language model that assesses the quality of an individual reasoning step within the context of the preceding path. This LLM can even be fine-tuned to enhance its verification capability. Alternatively, for domains with well-defined rules, it could be a symbolic engine or a set of programmatic checks. Furthermore, verifiers can be designed to predict the future utility or likelihood of success given the current partial path, drawing inspiration from value functions in reinforcement learning. Human expertise can also be incorporated to provide judgments on critical steps, especially in high-stakes scenarios.

One example of such a step-level verifier, particularly when using human feedback to assess intermediate progress, is the **process reward model** (**PRM**). A PRM is typically a separate language model trained to output a scalar reward for each reasoning step $a_{i'}$ within a partial path $\mathbf{a}_{\leq i}$. It provides a more direct and fine-grained supervisory signal compared to **outcome reward models** (**ORMs**) which only evaluate the final solution. However, the development

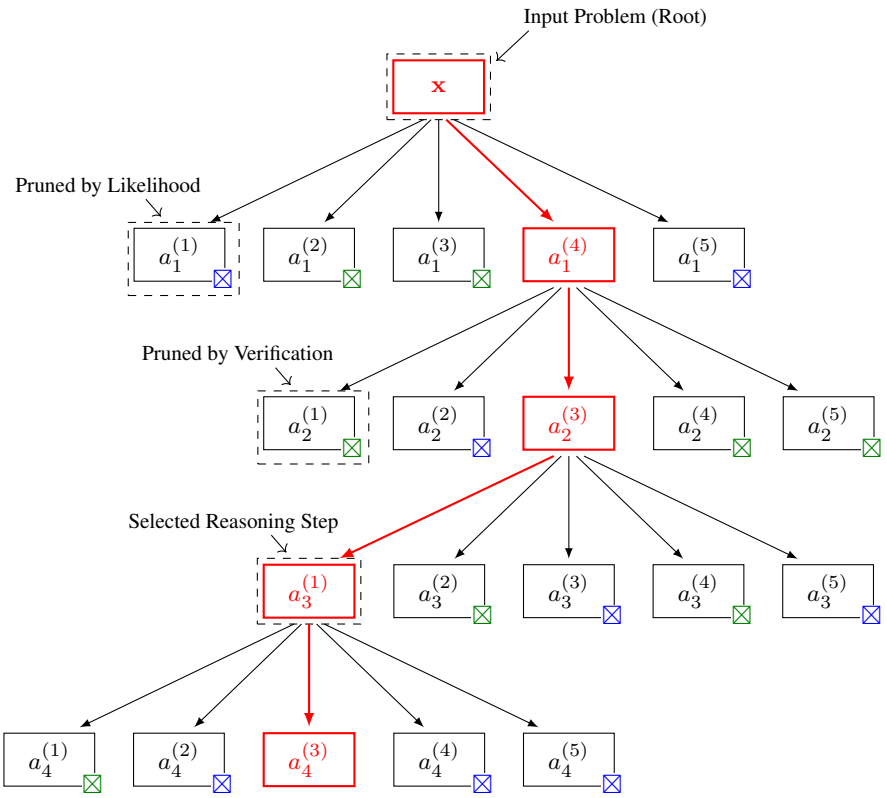


Figure 11.17: Illustration of step-level search with verifiers. $a_i^{(j)}$ = the j-th candidate for the i-th reasoning step, ☒ = candidate pruned by the LLM's output probability, and ☒ = candidate pruned by the verifier. Given the input problem as the root node, we expand the tree by generating multiple reasoning steps at each expansion. Each candidate can be pruned by either likelihood (as in standard decoding) or step-level verification. The unpruned candidates are then expanded to generate further reasoning steps. The process is iterated until a complete reasoning chain leading to a final answer is generated, or until a predefined search limit is reached.

of PRMs relies on step-level human annotations, such as preferences on different next steps. Collecting supervision for each intermediate step is considerably more labor-intensive and requires greater cognitive effort from human annotators than simply labeling final outcomes.

One alternative approach to developing training data for step-level verification is to use LLMs to generate such annotations automatically. For example, we can take a strong LLM, referred to as a teacher model, and prompt it to first generate a complete reasoning path for a given problem. Then, at each intermediate step within this path, we can prompt the same teacher LLM (or another capable LLM) to generate several alternative candidate next steps in addition to the one it originally chose. The teacher LLM can then be prompted again to evaluate these alternatives. These evaluation results (e.g., correct vs. incorrect) can then serve as data annotations. Alternatively, the generalization capabilities of PRMs can be leveraged. We can train a PRM on tasks where step-level verification is easier and then generalize this PRM to other tasks with little or no additional training.

Note that step-level verification also comes with its own problems. Frequent verification, especially if using an LLM as the verifier, can substantially increase computational costs and latency. The design of effective step-level verifiers is non-trivial itself. An inaccurate verifier might prematurely discard good reasoning paths or fail to identify flawed ones, thereby misleading the search. This makes the development of such systems more complex and difficult.

3. Encouraging Long Thinking

So far in this subsection, most of the methods are implicitly based on a simple idea: generating longer reasoning paths can help. In addition to CoT and search with verifications, we can consider alternative methods to achieve this. For example, we can prompt the LLM by explicitly asking for extended deliberation. Beyond direct prompting, we can also make modifications to the decoding process itself, such as adjusting token limits or applying penalties for short outputs. Another approach is to employ multi-stage generation schemes where the model incrementally builds upon its reasoning.

4. Training-based Scaling

As well as considering inference-time scaling methods without training, we also wish to consider methods that can improve intrinsic reasoning capabilities of LLMs by modifying their parameters through further training. While such training-based scaling methods typically require additional training cost and computational resources, they instill stronger reasoning skills directly into the model parameters, which in turn can lead to more effective and efficient reasoning performance. We can even combine them with training-free methods for better inference-time scaling results.

Although our discussion here is restricted to reasoning problems, methods for training-based scaling are common. Most of them have been discussed in Chapter 10. Here, we will briefly describe how these methods can be applied to improving inference-time scaling for reasoning problems.

- **Fine-tuning on Reasoning Data**. One of the most direct ways to enhance reasoning is by fine-tuning pre-trained LLMs on datasets specifically curated for reasoning tasks. These datasets can range from simple input-output pairs to more structured formats that include step-by-step reasoning processes. Typical examples include datasets of math word problems, logical deduction exercises, or code generation with explanations. By training on such data, the model learns from common reasoning patterns, and thus can generate detailed and coherent reasoning paths at test time.
- **Reinforcement Learning for Reasoning**. If we regard a verifier as a reward model, we can see that the methods discussed in the previous subsection are a direct application of the reward model to reasoning problems, though they are training-free. Of course, we can apply this reward model to LLM fine-tuning. This follows a standard paradigm of reinforcement learning. Given a reward model, the LLM, acting as a policy, is fine-tuned using reinforcement learning algorithms. The LLM generates reasoning steps

or full solutions, receives feedback (rewards) from the reward model, and updates its parameters to produce outputs that maximize these rewards. This process aligns the LLM output with notions of high-quality reasoning, thereby encouraging the LLM to generate more reliable reasoning paths. Another key issue is the training of the reward model. Generally, this reward model could be an outcome reward model that evaluates the correctness or quality of the final answer, or a process reward model that assesses the quality of each intermediate reasoning step, as discussed in the context of step-level verifiers. In some cases, we can even develop a reward model based on simple rules, such as giving bonuses to longer outputs.

- **Knowledge Distillation for Reasoning**. In this approach, a smaller, more efficient student LLM is trained to mimic the reasoning outputs or internal representations of a larger, more capable teacher LLM. The teacher model might generate detailed reasoning steps for a variety of problems. The student model then learns to reproduce these high-quality reasoning demonstrations. This strategy makes stronger reasoning capabilities more accessible by deploying them in smaller models that are less computationally expensive at inference time.
- **Iterative Refinement**. Training-based scaling can also involve iterative refinement. For example, an LLM can generate solutions to a set of problems. These solutions and their reasoning paths are then verified, either by humans or automatic verifiers. The correct reasoning paths are subsequently added to the training data, and the LLM is further fine-tuned on this augmented dataset. This creates a cycle where the LLM progressively improves its reasoning capabilities through repeated generation, critique, and learning.

The primary advantage of these training-based scaling methods is that they endow the LLM with stronger inherent reasoning skills. This directly contributes to improved inference-time scaling in several ways: it can lead to more efficient inference, as the LLM might require less extensive search or fewer generation samples to arrive at a correct solution. Moreover, the base quality of generated steps or solutions is higher. Therefore, a well-trained LLM might generalize its learned reasoning abilities to novel problems more effectively than an LLM relying solely on in-context learning or training-free inference schemes.

On the other hand, training-based approaches also present challenges, compared to the training-free counterparts. The creation of high-quality, large-scale training datasets for reasoning can be expensive and labor-intensive. The fine-tuning process itself, particularly for the largest LLMs or when using RL, can be computationally intensive and require substantial engineering effort. There is also the risk of the model overfitting to the specific types of problems or reasoning styles present in the training data, potentially limiting its performance on out-of-distribution tasks.

11.4 Summary

In this chapter, we have discussed the inference issue for LLMs. We have presented the prefilling-decoding framework and related decoding algorithms for LLM inference. Then, we

have described several techniques for efficient inference. We have also discussed inference-time scaling, which has been considered one of the most important methods for improving LLM reasoning.

Inference over sequential data has long been a concern in AI [Wozengraft and Reiffen, 1961; Viterbi, 1967; Forney, 1972]. In the context of NLP, this line of work dates back to the very early days of speech recognition and statistical machine translation [Koehn, 2010], where researchers faced the challenge of efficiently searching vast hypothesis spaces to find the most probable output sequence. Techniques like beam search and various pruning strategies were developed then to make this computationally tractable. At that time, models were relatively weak, and much of the research focused on developing powerful search algorithms to reduce search errors. These foundational ideas continue to influence modern approaches.

As we enter the era dominated by deep learning methods, models based on deep neural networks have become extremely powerful. Even with very simple search algorithms, these models can achieve excellent results. In this context, inference no longer seems as "important" as it once was, and research attention has gradually shifted toward model architectures, training methods, and scaling up models.

However, history tends to repeat itself. With the rise of LLMs, inference has once again attracted significant attention. This renewed focus is primarily manifested in two aspects:

- The inference cost for LLMs is very high. For example, efficiently deploying LLMs in high-concurrency, low-latency scenarios remains a challenging problem, making inference efficiency critically important. In this context, efficient architecture designs, optimized search algorithms, and various inference optimization strategies hold substantial practical significance.
- Input and output sequence lengths have significantly increased in complex tasks. Especially in tasks like mathematical reasoning, the growth of sequence lengths further highlights the importance of inference efficiency. Moreover, scaling the inference process has recently proven to be an effective way to improve the reasoning capabilities of models. Therefore, achieving efficient inference scaling is emerging as a particularly promising research direction.

Inference is now a wide-ranging topic that encompasses many techniques. It involves not only the development of model architectures and decoding algorithms, but is increasingly shaped by the intricate engineering and sophisticated systems-level optimizations required to deploy LLMs effectively and efficiently. Many of these techniques are beyond the scope of NLP or a specific AI area. Instead, the frontier of LLM inference optimization now extends deeply into domains traditionally considered core computer science and engineering. This systemic perspective has brought many new ideas to the study of inference problems. Unfortunately, this chapter cannot cover all relevant techniques — indeed, that would be an almost impossible task in itself. Ultimately, the best way to better understand and master these techniques may still lie in hands-on practice.

Bibliography

[Agrawal et al., 2023] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[Agrawal et al., 2024] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[Briski, 2025] Kari Briski. How scaling laws drive smarter, more powerful ai, 2025.

[Brown et al., 2024] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[Deepseek, 2025] Deepseek. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[Fan et al., 2018] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, 2018.

[Forney, 1972] GDJR Forney. Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. *IEEE Transactions on Information theory*, 18(3):363–378, 1972.

[Gou et al., 2024] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024.

[Holtzman et al., 2020] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

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

[Kumar and Byrne, 2004] Shankar Kumar and William Byrne. Minimum bayes-risk decoding for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 169–176, 2004.

[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.

[Leviathan et al., 2023] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from

transformers via speculative decoding. In *Proceedings of International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[Li et al., 2024] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. *arXiv preprint arXiv:2407.12391*, 2024.

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

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

[Nvidia, 2025] Nvidia. Nvidia nim llms benchmarking. https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html, 2025. Retrieved 2025-03-17.

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

[Patel et al., 2024] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[Pope et al., 2023] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, 2023.

[Shinn et al., 2023] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[Snell et al., 2024] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[Snell et al., 2025] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

[Su et al., 2022] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. *Advances in Neural Information Processing Systems*, 35:21548–21561, 2022.

[Timonin et al., 2022] Denis Timonin, BoYang Hsueh, and Vinh Nguyen. Accelerated inference for large transformer models using nvidia triton inference server. https://developer.nvidia.com/blog/accelerated-inference-for-large-transformer-models-using-nvidia-fastertransf 2022.

[Viterbi, 1967] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 1967.

[Wozengraft and Reiffen, 1961] John M. Wozengraft and Barney Reiffen. *Sequential Decoding*. The MIT Press, 1961.

[Wu et al., 2023] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

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

[Yu et al., 2022] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[Zhang et al., 2024] Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. In *ACL (Findings)*, 2024.

[Zhong et al., 2024] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.