AI Applications Lecture 5
Pipelines and Tokenization

SUZUKI, Atsushi
Jing WANG

2025-09-08

Contents

1 Introduction

1.1 Review of the Previous Lecture

In the previous lecture, we learned about the issue of **licenses** when using neural network models. We confirmed the non-trivial fact that the separation of **architecture** and **checkpoints** is crucial for understanding the scope of license application, and that even models touted as ”open” are not necessarily permitted for free use. In the exercise, we experienced the process of downloading a model from an online repository like the Hugging Face Hub, checking its license, and then using it.

1.2 Learning Outcomes of This Lecture

By the end of this lecture, students will be able to:

- Explain the significance and benefits of using a **pipeline**, which combines a neural network with other functions, rather than the neural network itself, in the practical application of generative AI, using natural language processing as an example.
- Given a pipeline provided by Hugging Face, roughly understand its structure as a function (algorithm) from its source code.
- Convert natural language strings into sequences of **tokens** for input into a neural network, and conversely, convert token sequences generated by the neural network back into natural language strings.

2 Neural Networks and Pipelines

In the practical application of generative AI, the single neural network f_θ we have discussed so far is rarely used as is. In most cases, **one or more neural networks** are combined with **other non-neural network algorithms (functions)** to build a larger single function (system). In this lecture, we will refer to this entire composite function as a **pipeline**.

Remark 2.1 (About the Terminology)**.** The term ”pipeline” is used as a central concept, particularly in libraries developed by Hugging Face such as `transformers` [a] and `diffusers` [b]. While this term is less frequently used with the same meaning in theoretical academic literature, it is a very useful concept for understanding practical AI systems.

[a] `https://huggingface.co/docs/transformers/en/pipeline_tutorial`
[b] `https://huggingface.co/docs/diffusers/using-diffusers/write_own_pipeline`

The purposes of constructing a pipeline are diverse, but the main motivations are as follows:

- **To reduce the amount of data handled by the core neural network.**

  - **Example 1: Tokenizer in Chat AI**. It converts long sentences into shorter sequences of tokens, reducing the sequence length that the neural network has to process. (The main topic of this lecture)
  - **Example 2: Variational Autoencoder (VAE) [2] in Image Generation**. It compresses high-resolution images into low-dimensional latent vectors, and computationally expensive models like diffusion models operate only in this low-dimensional space.

- **To interconvert between discrete objects like strings and continuous objects (real-valued vectors) that neural networks excel at.**
  - **Example 1 (overlapping with above): Tokenizer in Chat AI**. It converts strings into sequences of numbers (token IDs).
  - **Example 2: Sampling in Chat AI**. It selects a specific, discrete token ID from a continuous object, the "probability distribution of the next token," output by the neural network.

2.1 Updating the Framework of Training and Inference

By introducing the concept of a pipeline, we can update the framework of training and inference we have seen so far to a more realistic form.

- **Training:** Training data and one or more neural network architectures are given to a training algorithm to obtain checkpoints corresponding to each architecture.
- **Inference:** A **pipeline** $\phi_{f^{(1)}_{\boldsymbol{\theta}^{(1)*}}, f^{(2)}_{\boldsymbol{\theta}^{(1)*}}, \dots, g^{(1)}, g^{(2)}, \dots} : \mathcal{X} \to \mathcal{Y}$, constructed by combining one or more trained models (pairs of architecture and checkpoints) $f^{(1)}_{\boldsymbol{\theta}^{(1)*}}, f^{(2)}_{\boldsymbol{\theta}^{(1)*}}, \dots$, and non-neural network functions $g^{(1)}, g^{(2)}, \dots$, is given a new input to obtain the final output.

This updated relationship is shown in Figure 1. It is important to note that the inference phase consists of a pipeline composed of multiple components, not just a single trained model.

2.2 How to Understand the Structure of a Pipeline

Academic papers tend to focus on explaining the core neural network architecture that asserts novelty, and rarely provide a detailed explanation of the entire system, the pipeline, as a function or algorithm. Therefore, reading the implementation code of libraries like Hugging Face is an effective way to grasp the overall picture of a pipeline. However, these libraries are huge and complex, so you may not know where to start.

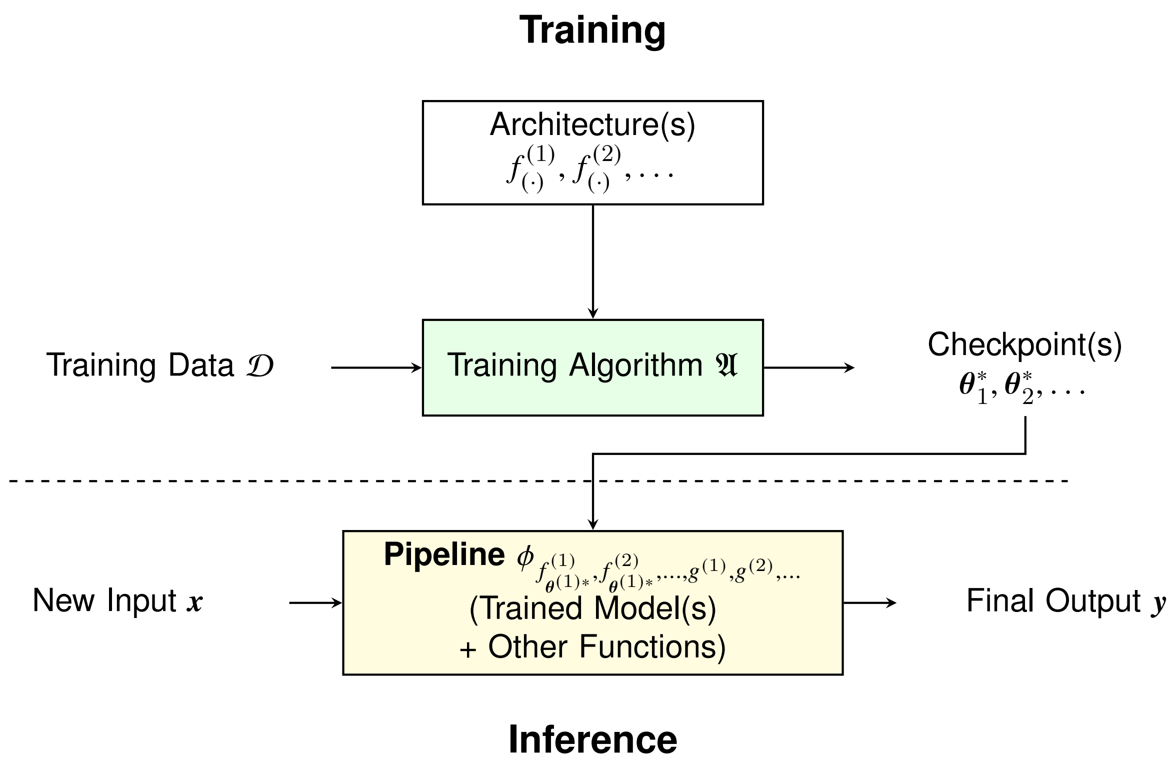
Training
Architecture(s) $f^{(1)}_{(\cdot)}, f^{(2)}_{(\cdot)}, \ldots$
Training Data $\mathcal{D}$
Training Algorithm $\mathfrak{A}$
Checkpoint(s) $\boldsymbol{\theta}^*_1, \boldsymbol{\theta}^*_2, \ldots$
New Input $\boldsymbol{x}$
Pipeline $\phi_{f^{(1)}_{\boldsymbol{\theta}^{(1)*}}, f^{(2)}_{\boldsymbol{\theta}^{(1)*}}, \ldots, g^{(1)}, g^{(2)}, \ldots}$ (Trained Model(s) + Other Functions)
Final Output $\boldsymbol{y}$
Inference

Figure 1: Scheme of Training and Inference Including a Pipeline

The procedure I recommend for grasping the overview of a pipeline is as follows[1].

1. Identify the pipeline's class: First, check the class name of the instance loaded with a method like `from_pretrained` (e.g., `print(type(pipeline_object))`). For example, for text generation with models like Qwen, it might be `TextGenerationPipeline` or similar[2].

2. Check the constructor (`__init__`): Look at the arguments of the `__init__` method of the identified class. This will give you a rough idea of what components (other models, processors, etc.) the pipeline is made of.

3. Check the execution method (`__call__`): Trace the processing of the class's `__call__` method (or a core method like `forward` or `generate`). This will show you how the components identified in `__init__` are combined and how the series of processes from input to output is executed.

[1]Once the relevant part of the source code is identified, asking a chat AI, "What does this code do?" is also a realistic and effective approach to understanding unfamiliar code. Since the source code is the very source of information, it is easy to verify whether the chat AI's answer is correct.

[2]`https://github.com/huggingface/transformers/blob/v4.56.1/src/transformers/pipelines/text_generation.py` However, since there are inheritance relationships between classes, it is often necessary to refer to parent classes in other files.

3 Structure of a Natural Language Generation Pipeline

In the remainder of this lecture, we will take the natural language generation task as a specific example. A typical pipeline used in auto-regressive language models like Qwen3 [7] [3] can be broadly seen as a composite function of the following four components.

1. **Tokenizer / Encoder:** Converts a natural language string (byte sequence) that humans read and write into a sequence of integer IDs (token sequence) corresponding to a fixed-size vocabulary.

2. **Neural Network and Sampler:** Takes a token sequence and returns another token sequence. The details will be covered next time, but internally it is a function defined by a recursive application of the neural network, determined by the neural network and a random seed.

3. **Detokenizer / Decoder:** Converts the token sequence generated by the sampler back into a natural language string (byte sequence).

This time, we will focus on the first and last parts, the **tokenizer** and detokenizer, which are closest to the input and output. This is because it is difficult to correctly understand the behavior of the neural network and sampler inside without understanding tokenization.

4 Tokenization

4.1 The Necessity of Tokenization

Why don't we input natural language strings (byte sequences) directly into a neural network?

The larger the scale of a neural network's architecture, i.e., the more parameters it has, the more complex functions it can represent, and performance improvement can be expected. Since natural language has an extremely complex structure, recent high-performance models that handle it have a vast number of parameters (e.g., Llama 3 has 70 billion (70B) parameters[4], Qwen2 has 72 billion (72B) parameters [5]).

However, the giantization of architecture comes with serious disadvantages.

- **Merit:** If the checkpoint is chosen appropriately, a complex function can be obtained, and high performance can be expected.

- **Demerit:** The computational resources (memory, computing power) and time required for training and inference become enormous. This limits the environments where it can be practically applied, increases power consumption, and lengthens response times, thereby narrowing the range of applications.

[3] https://huggingface.co/docs/transformers/en/pipeline_tutorial
[4] https://huggingface.co/meta-llama/Meta-Llama-3-70B
[5] https://huggingface.co/Qwen/Qwen2-72B-Instruct

Therefore, there is a strong motivation to **realize a function with sufficient expressive power to solve a task while keeping the scale of the neural network within a realistic range**. Pipelining is an important strategy to solve this problem. By sandwiching relatively lightweight processes (such as tokenization) before and after the neural network, the burden on the neural network itself is reduced, and a balance between efficiency and performance of the entire system is achieved.

4.2 Why Not Input Byte Streams Directly?

On a computer, characters are represented as byte sequences, but using this directly as input to a neural network is generally not a good strategy. The reason is that **the proximity of byte values is completely unrelated to the proximity of the meanings they represent**.

For example, in UTF-8 encoding, the byte value of ‘A‘ is ‘0x41‘ and ‘B‘ is ‘0x42‘, which are numerically close. However, ‘a‘ is ‘0x61‘, which is numerically distant from ‘A‘. Neural networks essentially represent (locally Lipschitz) continuous functions, so they tend to return similar outputs for numerically close inputs. If byte values were input as is, the model might treat ‘A‘ and ‘B‘ as similar things and ‘A‘ and ‘a‘ as completely different things. This contradicts linguistic intuition.

Remark 4.1 (Comparison with Image Data)**.** This situation is in contrast to image data. An image is represented by a tensor of pixel color information. The fact that pixel values are close means that the colors are physically close, which often leads to semantic closeness (such as the surface of the same object). Therefore, it is common for the pixel data of an image to be normalized and then input directly into a neural network.

4.3 One-Hot Encoding

If using byte values directly is inappropriate, what should be input? It is difficult to decide in advance ”which characters are semantically close,” and it could introduce human bias. Therefore, the simplest solution is to **assume that all characters are equidistant**. The idea that embodies this is **One-Hot Encoding**.

Definition 4.1 (One-Hot Encoding)**.** Let a finite vocabulary set $\mathcal{V} = \{v_1, v_2, \dots, v_D\}$ be given, where $D = |\mathcal{V}|$ is the vocabulary size. The One-Hot encoding of an element $v_i \in \mathcal{V}$ is a D-dimensional vector $\boldsymbol{e}_i \in \{0,1\}^D$ in which only the i-th component is 1 and all other components are 0.

$$(\boldsymbol{e}_i)_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases} \tag{1}$$

Example 4.1 (One-Hot Encoding of Characters)**.** Let the vocabulary be $\mathcal{V} = \{\text{a, b, c}\}$.

- 'a' $\rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

- 'b' $\rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

- 'c' $\rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

In this representation, the Euclidean distance between any two vectors is $\sqrt{2}$, and equidistance is maintained.

4.4 Motivation for Shortening Sequence Length

When each character is converted into a One-Hot vector, a sentence becomes a sequence of vectors. For example, "abac" becomes $\left(\left[\begin{smallmatrix} 1 \\ 0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 0 \\ 1 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 \\ 0 \\ 0 \end{smallmatrix}\right], \left[\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}\right]\right)$.

Now, consider making the unit of conversion larger than a character, for example, a word. If we choose words as our vocabulary, the sequence length becomes dramatically shorter.

Example 4.2 (Character-level vs. Word-level)**.** Sentence: "Language models are powerful."

- **Character-level (28 characters):** 'L', 'a', 'n', 'g', 'u', 'a', 'g', 'e', ' ', 'm', ...

- **Word-level (4 words):** "Language", "models", "are", "powerful"

For simplicity, converting each unit to an integer ID would look like this:

- **Character-level:** (44, 65, 78, ..., 76) (length 28)

- **Word-level:** (1345, 11646, 322, 2) (length 4)

The shorter the sequence length, the more the computational complexity of the neural network (especially models like the Transformer) is reduced, leading to faster training and inference and a reduction in necessary computational resources.

However, using words as the basic unit creates new problems.

- **Out-of-Vocabulary (OOV) Problem:** Cannot handle words not in the vocabulary (new words, proper nouns, typos).

- **Vocabulary Explosion:** Trying to cover many words results in a huge vocabulary size, increasing the dimensionality of the One-Hot vectors.

- **Difficulty in Multilingual Support:** It is necessary to have a huge vocabulary for each language, which is inefficient.

5 Subword Tokenization: Byte Pair Encoding (BPE)

To solve these problems, an intermediate approach between ”character” and ”word,” **subword** tokenization, has become mainstream. Its representative algorithm is **Byte Pair Encoding (BPE)** [1,5]. The basic idea of BPE is to build a data-adapted vocabulary by **merging frequently occurring pairs of characters into a new single unit (token)**.

Many modern models such as GPT-2 [4] and Qwen [7] adopt BPE or its derivative algorithms as their tokenizer. For example, the merge rules for the Qwen3-0.6B-Instruct model are publicly available for anyone to see[6].

5.1 The BPE Algorithm

BPE is divided into (1) a dictionary creation phase that builds the vocabulary and merge rules, and (2) an encoding/decoding phase that uses them to process text.

Definition 5.1 (BPE Dictionary Creation Algorithm (Formal Definition))**.** The inputs are a training corpus C (a multiset of strings) and the number of merges N.

1. **Initialization:**
   - Initialize the vocabulary $\mathcal{V}$ with the set of all characters (basic units) in C.
   - Split each string in the corpus into a sequence of basic unit tokens, and let $\mathcal{D}$ be the multiset of these sequences.
   - Initialize the ordered list of merge rules as $\boldsymbol{m} = ()$.
2. **Iteration:** For $k = 1$ to N, repeat the following:
   (a) Count the frequency of all adjacent token pairs (t_1, t_2) in $\mathcal{D}$.
   (b) Find the most frequent pair $(t_a, t_b) \leftarrow \arg\max_{(t_1,t_2)} \mathrm{count}((t_1, t_2))$.[a]
   (c) Generate a new token $t_{\text{new}} \leftarrow t_a \cdot t_b$ (concatenated as a string).
   (d) Update the vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$.
   (e) Add the merge rule to the list: $\boldsymbol{m} \leftarrow \boldsymbol{m} \oplus ((t_a, t_b) \to t_{\text{new}})$.
   (f) Replace all adjacent pairs (t_a, t_b) in $\mathcal{D}$ with t_{new}.

The outputs are the final vocabulary $\mathcal{V}$ and the ordered merge rules $\boldsymbol{m}$.

[a]If there are multiple most frequent pairs, select one based on some criterion (e.g., lexicographical order).

Remark 5.1 (Idea)**.** The basic idea of this algorithm is to repeat the operation of ”merging the most frequently adjacent pair into one and considering it a new character (token).” This uses the concept of data compression to automatically learn statistically important substrings

[6]`https://huggingface.co/Qwen/Qwen3-0.6B/blob/main/merges.txt`

(such as frequent words and affixes) as vocabulary.

Definition 5.2 (BPE Encode/Decode Algorithms (Formal Definition)). • **Encoding:** The inputs are a string S_{in} and the ordered merge rules $\boldsymbol{m} = (M_1, M_2, \dots, M_N)$ obtained during dictionary creation.

1. Split S_{in} into a token sequence T of basic units (characters).
2. Apply each merge rule $M_k = ((t_a, t_b) \to t_{\text{new}})$ from $\boldsymbol{m}$ in order from $k = 1, \dots, N$. Applying a rule means replacing all adjacent pairs (t_a, t_b) in T with t_{new}.
3. Output the final token sequence T_{out}.

- **Decoding:** The input is a token sequence $T_{\text{in}} = (t_1, t_2, \dots, t_m)$.
  - Concatenate all tokens in order as strings: $S_{\text{out}} = t_1 \cdot t_2 \cdot \dots \cdot t_m$.

  Output S_{out}.

Remark 5.2 (Idea). Encoding is a process of applying the learned merge rules as a ”recipe” to the input string in the same order as during training. By processing from the highest priority (= earliest merged) pairs, consistent tokenization is possible. Decoding is a simple process of just converting tokens back to their original strings and joining them.

Example 5.1 (Detailed Example of BPE Operation). Suppose the training corpus is only ‘”fast faster fastest slow slower slowest”‘. For simplicity, assume words are separated by spaces.

1. **Initial State:**
   - Vocabulary $\mathcal{V}$: ‘f, a, s, t, e, r, l, o, w‘
   - Data $\mathcal{D}$: ‘f a s t‘, ‘f a s t e r‘, ‘f a s t e s t‘, ‘s l o w‘, ‘s l o w e r‘, ‘s l o w e s t‘
2. **1st Merge:**
   - Pair frequencies: ‘(s, t)‘: 6 times, ‘(f, a)‘: 3 times, ‘(s, l)‘: 3 times, and many others.
   - Most frequent pair: ‘(s, t)‘. Add new token ‘st‘ to vocabulary.
   - Data: ‘f a st‘, ‘f a st e r‘, ‘f a st e st‘, ‘s l o w‘, ‘s l o w e r‘, ‘s l o w e st‘
3. **2nd Merge:**
   - Pair frequencies: ‘(f, a)‘: 3 times, ‘(s, l)‘: 3 times, ‘(e, r)‘: 2 times, ‘(e, st)‘: 2 times, etc.
   - Most frequent pairs: ‘(f, a)‘ and ‘(s, l)‘ are tied. Choose ‘(f, a)‘ by alphabetical order. Add new token ‘fa‘ to vocabulary.
   - Data: ‘fa st‘, ‘fa st e r‘, ‘fa st e st‘, ‘s l o w‘, ‘s l o w e r‘, ‘s l o w e st‘

4. **3rd Merge:**
   - Most frequent pair: ‘(fa, st)‘ (3 times). Add new token ‘fast‘ to vocabulary.
   - Data: ‘fast‘, ‘fast e r‘, ‘fast e st‘, ‘s l o w‘, ‘s l o w e r‘, ‘s l o w e st‘
5. **4th Merge:**
   - Most frequent pair: ‘(s, l)‘ (3 times). Add new token ‘sl‘ to vocabulary.
   - Data: ‘fast‘, ‘fast e r‘, ‘fast e st‘, ‘sl o w‘, ‘sl o w e r‘, ‘sl o w e st‘

By repeating this process, frequent words like ‘slow‘ and frequent suffixes like ‘er‘ and ‘est‘ are learned as single tokens. This allows an unknown word like ‘slowing‘ to be handled by splitting it into the learned ‘slow‘ and the unknown ‘ing‘, partially preserving information. Also, by starting BPE from **bytes** instead of characters, the complexity of Unicode can be avoided, and all languages and emojis can be handled uniformly.

Exercise 5.1. Suppose the corpus is only ‘”unzip”, ”unzipped”, ”zip”‘ and the basic units are ‘’u’, ’n’, ’z’, ’i’, ’p’, ’e’, ’d’‘. If you perform 5 merges, what will the final merge rules and vocabulary be? Also, use those rules to encode the string ‘”unzipping”‘.

Answer. 1. **Initial State:** The data is split into ‘u n z i p‘, ‘u n z i p p e d‘, ‘z i p‘.

2. **1st Merge:**
   - Pair frequencies: ‘(z, i)‘: 3 times, ‘(i, p)‘: 3 times, ‘(u, n)‘: 2 times, ‘(n, z)‘: 2 times, others once.
   - Most frequent pairs are ‘(z, i)‘ and ‘(i, p)‘. Choose ‘(i, p)‘ by alphabetical order.
   - Merge rule 1: ‘i p‘ $\rightarrow$ ‘ip‘. Add ‘ip‘ to vocabulary.
   - Data: ‘u n z ip‘, ‘u n z ip p e d‘, ‘z ip‘
3. **2nd Merge:**
   - Pair frequencies: ‘(z, ip)‘: 3 times, ‘(u, n)‘: 2 times, ‘(n, z)‘: 2 times, etc.
   - Most frequent pair is ‘(z, ip)‘.
   - Merge rule 2: ‘z ip‘ $\rightarrow$ ‘zip‘. Add ‘zip‘ to vocabulary.
   - Data: ‘u n zip‘, ‘u n zip p e d‘, ‘zip‘
4. **3rd Merge:**
   - Pair frequencies: ‘(u, n)‘: 2 times, ‘(n, zip)‘: 2 times.
   - Choose most frequent pair ‘(n, zip)‘ (by alphabetical order).
   - Merge rule 3: ‘n zip‘ $\rightarrow$ ‘nzip‘. Add ‘nzip‘ to vocabulary.

- Data: ‘u nzip‘, ‘u nzip p e d‘, ‘zip‘

5. **4th Merge:**
   - Pair frequencies: ‘(u, nzip)‘: 2 times, others once.
   - Most frequent pair is ‘(u, nzip)‘.
   - Merge rule 4: ‘u nzip‘ → ‘unzip‘. Add ‘unzip‘ to vocabulary.
   - Data: ‘unzip‘, ‘unzip p e d‘, ‘zip‘
6. **5th Merge:**
   - Pair frequencies: ‘(p, p)‘: 1 time, ‘(p, e)‘: 1 time, ‘(e, d)‘: 1 time, etc.
   - Choose most frequent pair ‘(e, d)‘ (by alphabetical order).
   - Merge rule 5: ‘e d‘ → ‘ed‘. Add ‘ed‘ to vocabulary.
   - Data: ‘unzip‘, ‘unzip p p ed‘, ‘zip‘
7. **Final Result:**
   - Merge rules (ordered): ‘(i, p)‘, ‘(z, ip)‘, ‘(n, zip)‘, ‘(u, nzip)‘, ‘(e, d)‘
   - Vocabulary: ‘’u’, ’n’, ’z’, ’i’, ’p’, ’e’, ’d’, ’ip’, ’zip’, ’nzip’, ’unzip’, ’ed’‘
8. **Encoding:** String ‘”unzipping”‘
   (a) Initial split: ‘[’u’, ’n’, ’z’, ’i’, ’p’, ’p’, ’i’, ’n’, ’g’]‘
   (b) Merge rule 1 (‘i p‘ → ‘ip‘): ‘[’u’, ’n’, ’z’, ’ip’, ’p’, ’i’, ’n’, ’g’]‘
   (c) Merge rule 2 (‘z ip‘ → ‘zip‘): ‘[’u’, ’n’, ’zip’, ’p’, ’i’, ’n’, ’g’]‘
   (d) Merge rule 3 (‘n zip‘ → ‘nzip‘): ‘[’u’, ’nzip’, ’p’, ’i’, ’n’, ’g’]‘
   (e) Merge rule 4 (‘u nzip‘ → ‘unzip‘): ‘[’unzip’, ’p’, ’i’, ’n’, ’g’]‘
   (f) Merge rule 5 (‘e d‘ → ‘ed‘): Not applicable.
   (g) Final token sequence: ‘[’unzip’, ’p’, ’i’, ’n’, ’g’]‘

5.2 The Role of BPE in the Pipeline

Let’s reconfirm the role of BPE in the context of a natural language generation pipeline.

- BPE **encoding** converts a natural language byte sequence into a sequence of token IDs (an integer sequence in implementation).
- The neural network receives this token ID sequence and outputs a sequence of probability distributions for the next token.
- The sampler selects the next token ID from the probability distribution.

- BPE **decoding** converts the generated token ID sequence into a byte sequence corresponding to natural language.

Remark 5.3. In the case of byte-level BPE, there is no guarantee that the decoded byte sequence will constitute a valid UTF-8 string, etc. It is possible for invalid byte sequences to be generated.

6 Special Tokens and Chat AI

As we have learned so far, by wrapping the processing of a neural network (and sampler) with encoding and decoding processes like BPE, we can realize a process that takes a natural language string (byte sequence) and returns a natural language string. However, the implementation of widely used Chat AIs today is insufficient with just this. This is because the input to a Chat AI is not a single string, but includes structural information such as the conversation history up to that point and fixed inputs by non-users called system prompts. Furthermore, it is empirically known that the quality of the final answer improves if the process leading up to it is also output along with the direct answer (depending on the context, this is called Chain of Thought, Reasoning, Thinking, etc.). We want to handle these structures as uniformly as possible. Since we already have the technology to input and output token sequences, using it is a natural idea. That is, if we can interconvert multiple strings with structure and token sequences, it would be sufficient. For this purpose, we not only convert strings into token sequences but also use **Special Tokens** to represent the structure, and according to rules called chat templates, convert the input into a single token sequence, and convert the output from a token sequence into a structured output. For details on these implementations, the official Hugging Face page is comprehensive[7].

6.1 Chat Templates and Special Tokens

The input to a Chat AI is often given as a list of dictionaries recording ”who said what.” Below is an example from `Qwen/Qwen3-4B-Thinking-2507`.

```
messages = [
    {"role": "system", "content": "This is a test. Think as short as possible."},
    {"role": "user", "content": "Find 7+8."},
]
```

Here, ‘role‘ indicates the role of the utterance, where ‘system‘ is a general instruction to the AI, ‘user‘ is input from a human, and ‘assistant‘ means the AI’s response. Since neural networks cannot handle such structured data directly, it needs to be converted into a single

[7]`https://huggingface.co/docs/transformers/v4.48.0/en/chat_templating`. This is an explanation in an older version of the documentation, but I could not find a corresponding description in the latest version, possibly due to updates in the document structure.

string (and ultimately a single token sequence) containing special tokens. This conversion rule is called a **Chat Template**.

The tokenizer from Hugging Face provides a method called `apply_chat_template`, which performs the conversion according to a model-specific template (note that a neural network is not involved here). For example, in the case of `Qwen/Qwen3-4B-Thinking-2507`, the 'messages' above are converted into a string like the following.

```
<|im_start|>system
This is a test. Think as short as possible.<|im_end|>
<|im_start|>user
Find 7+8.<|im_end|>
<|im_start|>assistant
<think>
```

This string contains, in addition to the normal tokens included in the BPE vocabulary, the following special tokens.

- `<|im_start|>` and `<|im_end|>`: Markers indicating the start and end of each utterance.
- `system`, `user`, `assistant`: Tokens that explicitly state the role of the utterance.
- `<think>` and `</think>`: Markers indicating the start and end of the model's "thinking" part.

Particularly important is the part at the end, `<|im_start|>assistant\n<think>`. Basically, the neural network of a large language model is designed to be a function that, when appropriately combined with a sampler, takes the input as a part of a complete token sequence from the beginning to some point and outputs the subsequent continuation of the token sequence (as will be explained later, this is not obvious considering the original purpose!). Among them, neural network models that use "Thinking," such as `Qwen/Qwen3-4B-Thinking-2507`, handle complete token sequences in the form of "*Question* `<think>` *Reasoning for the answer* `</think>` *Answer*". Therefore, by giving "*Question* `<think>`" as input, the remaining part of the complete string, "*Reasoning for the answer* `</think>` *Answer*", is obtained as output.

Remark 6.1 (The Non-obvious Domain of Chat AI Models and Unexpected "Features")**.** We have stated that the goal of AI or machine learning is to acquire a function that represents an appropriate input-output relationship. But what is an appropriate input-output relationship? In the case of Chat AI, considering its purpose, it should be sufficient to take a question as input and output an appropriate response. However, large language models (LLMs) used in Chat AI have, as mentioned above, ended up learning a token sequence completion function. To be more specific, and simplifying by ignoring special tokens, they have ended up learning a function that takes a partial token sequence from the beginning of a proper

token sequence of the form ”*Question Answer*” and outputs the rest.
The difference between a function that returns a response to a question and a function that completes a token sequence consisting of a question and an answer is hard to see at first glance. You might think that being able to choose the most suitable response token sequence for a question token sequence is the same. In fact, the difference between the two lies in their domains. The token sequence completion function has an unnecessarily larger domain. That is, if it only receives inputs in the form of a ”*Question*”, that’s fine, but the token sequence completion function also accepts inputs in the form of ”*Question Beginning of the answer*” and returns the completed string. This allows one to control the direction of the response. This property is both convenient and can be used for attacks. The control of responses by this method, when considered as an attack method, is called Prefix injection (e.g., [6]) or Prefilling Attacks (e.g., [3]).

7 Conclusion and Future Outlook

7.1 The Overall Picture of the Natural Language Generation Pipeline

Integrating the discussions so far, let’s reconfirm the overall picture of the natural language generation pipeline we have covered in the exercises. This pipeline is realized as a composition of the following series of functions.

1. **Template renderer:** Converts a list of dictionaries with roles and content (`messages`) into a single prompt string containing special tokens. Hereafter, we will denote this function as Renderer.

2. **Tokenizer (Encoder):** Converts the prompt string into an integer sequence of token IDs. Hereafter, we will denote this function as Tokenizer.

3. **Neural Network and Sampler:** Takes a token sequence and returns another token sequence. The details will be covered next time, but internally it is a function defined by a recursive application of the neural network, determined by the neural network and a random seed. To state it more formally (omitting the dependency on the random seed for brevity), it takes a token ID sequence as input and outputs a token ID sequence. The function Sampler$_{f_{\boldsymbol{\theta}^*}}$, determined by the trained neural network $f_{\boldsymbol{\theta}^*}$, performs the input and output of token ID sequences. Here, Sampler$_{(\cdot)}$ is, so to speak, a function of a function, which, given a function f that takes a token ID sequence and returns scores over the token set, returns a function Sampler$_f$ that takes a token ID sequence and returns a token ID sequence. As will be detailed next time, what is important is that since $f_{(\cdot)}$ is a parametric function, Sampler$_{f_{(\cdot)}}$ is also a parametric function.

4. **Detokenizer (Decoder):** Converts the generated token ID sequence into a human-readable string. Hereafter, we will denote this function as Detokenizer.

The structure of this pipeline is shown in Figure 2. Mathematically, the entire pipeline ϕ can be written as a composition of these functions: $\phi = \text{Detokenizer} \circ \text{Sampler}_{f_{\theta^*}} \circ \text{Tokenizer} \circ \text{Renderer}$. Since it contains the parametric function f_{θ^*} internally, the entire pipeline can also be considered a parametric function dependent on the checkpoint $\boldsymbol{\theta}^*$.

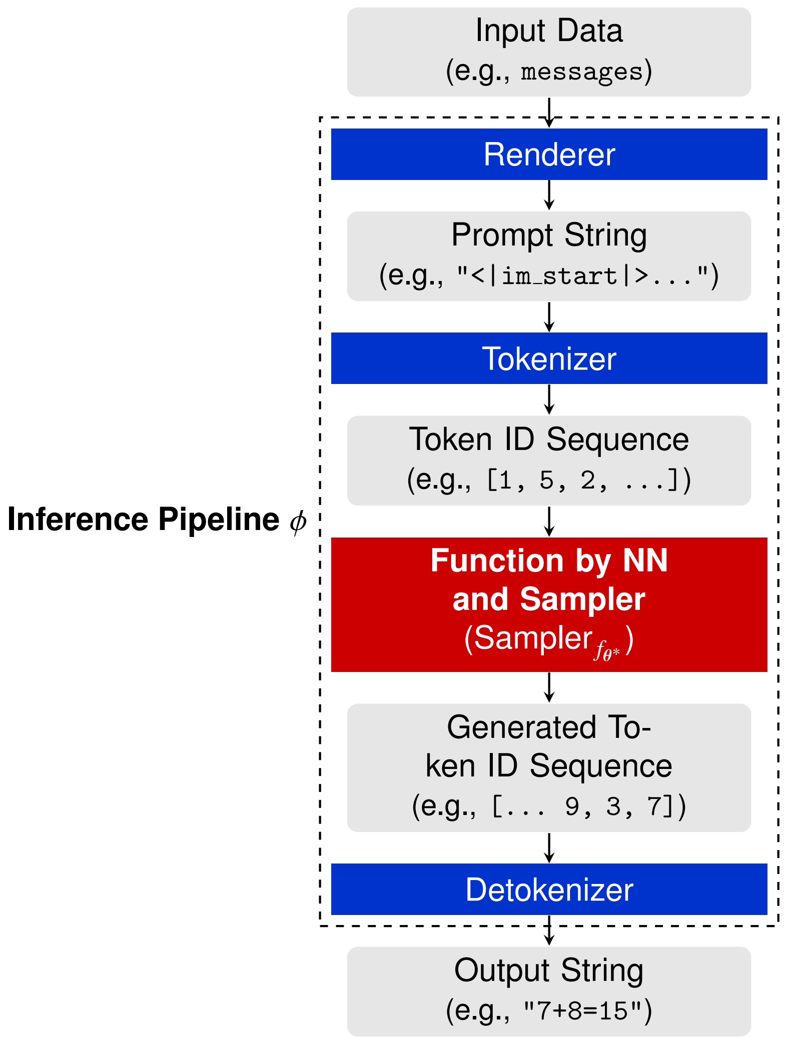

Figure 2: Inference pipeline for natural language generation. In the process called training, typically only the checkpoint $\boldsymbol{\theta}^*$ is changed, but the entire inference pipeline is a composition of many functions, and of course, as a whole, it depends on the checkpoint $\boldsymbol{\theta}^*$. Note that, in general, Sampler also depends on a random seed, but it is omitted in the figure.

7.2 Summary of Today's Lecture

- In the practical application of generative AI, a **pipeline** that combines multiple models and algorithms is used, rather than a single neural network. This is an important strategy for balancing computational efficiency and performance.
- The structure of a pipeline can be roughly grasped by reading the source code of libraries like Hugging Face (especially `__init__` and `__call__`).

- An important component of the pipeline in natural language processing is **tokenization**. It is responsible for interconverting between the strings that humans handle and the numerical data (token ID sequences, embedding vectors) that neural networks handle.
- **Byte Pair Encoding (BPE)** is an effective subword tokenization algorithm that can handle unknown words while shortening the sequence length by learning frequently occurring strings as single tokens.
- In Chat AI, **special tokens** are used to represent the structure of a conversation, and a prompt string is constructed according to a **chat template**. By understanding this mechanism, advanced control such as injecting the model's thoughts becomes possible.

7.3 Preview of the Next Lecture

This time, we focused on tokenization, the input part of the pipeline. Next time, we will look in detail at the output side of the neural network, namely the technique of **sampling**, which is "how the next token is determined from a probability distribution."

References

[1] Philip Gage. A new algorithm for data compression. C Users Journal, 12(2):23–38, 1994.

[2] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114, 2013.

[3] Xiangyu Qi, Ashwinee Panda, Kaifeng Lyu, Xiao Ma, Subhrajit Roy, Ahmad Beirami, Prateek Mittal, and Peter Henderson. Safety alignment should be made more than just a few tokens deep. In The Thirteenth International Conference on Learning Representations, 2025.

[4] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9, 2019.

[5] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, 2016.

[6] Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? Advances in Neural Information Processing Systems, 36:80079–80110, 2023.

[7] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025.