

AI Applications Lecture 5

Pipelines and Tokenization

SUZUKI, Atsushi^a

^aI thank my collaborator, Jing WANG, for cooperating in creating the teaching materials.

Outline

Introduction

Neural Networks and Pipelines

Structure of a Natural Language Generation Pipeline

Tokenization

Subword Tokenization: Byte Pair Encoding (BPE)

Special Tokens and Chat AI

Conclusion and Future Outlook

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.

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.
- We also saw that even models touted as "open" are not necessarily permitted for free use.

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.
- We also saw 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, in the practical application of generative AI.

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, in the practical application of generative AI.
- Given a pipeline provided by Hugging Face, roughly understand its structure as a function (algorithm) from its source code.

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, in the practical application of generative AI.
- 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.

Neural Networks and Pipelines

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.

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).

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**.

2. Neural Networks and Pipelines

Remark

The term "pipeline" is used as a central concept, particularly in libraries developed by Hugging Face such as `transformers`¹ and `diffusers`². 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.

¹https://huggingface.co/docs/transformers/en/pipeline_tutorial

²https://huggingface.co/docs/diffusers/using-diffusers/write_own_pipeline

2. Neural Networks and Pipelines

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

1. 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: VAE in Image Generation.** It compresses high-resolution images into low-dimensional latent vectors, so expensive models only operate in this small space.

2. Neural Networks and Pipelines

The second reason is:

To interconvert between discrete objects (e.g., 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 from a continuous probability distribution output by the network.

2.1 Updating the Framework of Training and Inference

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

2.1 Updating the Framework of Training and Inference

By introducing the concept of a pipeline, we can update our framework of training and inference 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** ϕ , constructed by combining one or more trained models and non-neural network functions, is given a new input to obtain the final output.

$$\phi_{f_{\theta^{(1)*}}, \dots, g^{(1)}, \dots}^{(1)} : \mathcal{X} \rightarrow \mathcal{Y}$$

2.1 Updating the Framework of Training and Inference

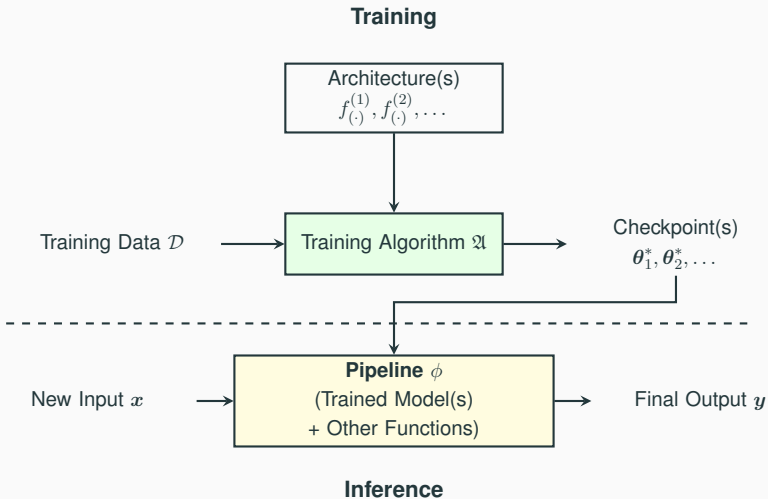


Figure 1: Scheme of Training and Inference Including a Pipeline

2.2 How to Understand the Structure of a Pipeline

Academic papers tend to focus on the core neural network architecture and rarely explain the entire pipeline.

2.2 How to Understand the Structure of a Pipeline

Academic papers tend to focus on the core neural network architecture and rarely explain the entire pipeline.

Therefore, reading the implementation code of libraries like Hugging Face is an effective way to grasp the overall picture. However, these libraries are huge, so where do you start?

2.2 How to Understand the Structure of a Pipeline

Here is a recommended procedure for grasping the overview of a pipeline:

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))`).

2.2 How to Understand the Structure of a Pipeline

Here is a recommended procedure for grasping the overview of a pipeline:

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

2.2 How to Understand the Structure of a Pipeline

Here is a recommended procedure for grasping the overview of a pipeline:

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))`).
2. **Check the constructor** (`__init__`): Look at the arguments of the `__init__` method of that 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. This will show you how the components are combined and how the series of processes from input to output is executed.

Structure of a Natural Language Generation Pipeline

3. Structure of a Natural Language Generation Pipeline

Let's take the natural language generation task as a specific example. A typical pipeline used in auto-regressive language models can be broadly seen as a composite function of the following components.

3. Structure of a Natural Language Generation Pipeline

Let's take the natural language generation task as a specific example. A typical pipeline used in auto-regressive language models can be broadly seen as a composite function of the following components.

1. **Tokenizer / Encoder:** Converts a natural language string into a sequence of integer IDs (tokens).
2. **Neural Network and Sampler:** Takes a token sequence and returns another token sequence.

3. Structure of a Natural Language Generation Pipeline

Let's take the natural language generation task as a specific example. A typical pipeline used in auto-regressive language models can be broadly seen as a composite function of the following components.

1. **Tokenizer / Encoder:** Converts a natural language string into a sequence of integer IDs (tokens).
2. **Neural Network and Sampler:** Takes a token sequence and returns another token sequence.
3. **Detokenizer / Decoder:** Converts the generated token sequence back into a natural language string.

3. Structure of a Natural Language Generation Pipeline

This time, we will focus on the first and last parts, the **tokenizer** and **detokenizer**, which are closest to the input and output.

3. Structure of a Natural Language Generation Pipeline

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.

Tokenization

4.1 The Necessity of Tokenization

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

4.1 The Necessity of Tokenization

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

The larger a neural network's architecture, the more complex functions it can represent, and performance improvement can be expected.

4.1 The Necessity of Tokenization

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

The larger a neural network's architecture, the more complex functions it can represent, and performance improvement can be expected.

Recent high-performance models have a vast number of parameters (e.g., Llama 3 70B, Qwen2 72B).

4.1 The Necessity of Tokenization

However, the giantization of architecture comes with serious disadvantages.

4.1 The Necessity of Tokenization

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.

4.1 The Necessity of Tokenization

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 where they can be used and increases costs.

4.1 The Necessity of Tokenization

There is a strong motivation to **realize a function with sufficient expressive power while keeping the scale of the neural network within a realistic range.**

4.1 The Necessity of Tokenization

There is a strong motivation to **realize a function with sufficient expressive power while keeping the scale of the neural network within a realistic range.**

Pipelining is an important strategy to solve this problem.

4.1 The Necessity of Tokenization

There is a strong motivation to **realize a function with sufficient expressive power 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 (like tokenization) before and after the neural network, the burden on the network itself is reduced, achieving a balance between efficiency and performance.

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.

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.**

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, 'A' ('0x41') and 'B' ('0x42') are numerically close. But 'a' ('0x61') is numerically distant from 'A'. Neural networks, being continuous functions, would incorrectly infer that 'A' and 'B' are similar, while 'A' and 'a' are very different.

4.2 Why Not Input Byte Streams Directly?

Remark

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?

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.

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**.

4.3 One-Hot Encoding

Definition (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 $e_i \in \{0, 1\}^D$ in which only the i -th component is 1 and all other components are 0.

$$(e_i)_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases} \quad (1)$$

4.3 One-Hot Encoding

Example (One-Hot Encoding of Characters)

Let the vocabulary be $\mathcal{V} = \{a, b, c\}$.

- '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 a sequence of four 3D vectors.

$$\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right)$$

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 a sequence of four 3D vectors.

$$\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right)$$

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.

4.4 Motivation for Shortening Sequence Length

Example (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"

4.4 Motivation for Shortening Sequence Length

Example (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)

4.4 Motivation for Shortening Sequence Length

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).

4.4 Motivation for Shortening Sequence Length

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.

4.4 Motivation for Shortening Sequence Length

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.

Subword Tokenization: Byte Pair Encoding (BPE)

5. Subword Tokenization: Byte Pair Encoding (BPE)

To solve these problems, an intermediate approach between "character" and "word," **subword** tokenization, has become mainstream.

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, 2].

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, 2].

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)**.

5.1 The BPE Algorithm

BPE is divided into two phases:

1. A **dictionary creation phase** that builds the vocabulary and merge rules.

5.1 The BPE Algorithm

BPE is divided into two phases:

1. A **dictionary creation phase** that builds the vocabulary and merge rules.
2. An **encoding/decoding phase** that uses them to process text.

5.1 The BPE Algorithm

Definition (BPE Dictionary Creation Algorithm)

The inputs are a training corpus \mathcal{C} and the desired number of merges N .

1. Initialization:

- Initialize the vocabulary \mathcal{V} with all unique characters in \mathcal{C} .
- Split all text in the corpus into sequences of these characters.
- Initialize an empty, ordered list of merge rules m .

5.1 The BPE Algorithm

Definition (BPE Dictionary Creation Algorithm (cont.))

2. **Iteration:** For $k = 1$ to N , repeat the following:
 - 2.1 Count the frequency of all adjacent token pairs in the corpus.
 - 2.2 Find the most frequent pair (t_a, t_b) .
 - 2.3 Generate a new token t_{new} by concatenating t_a and t_b .
 - 2.4 Add the new token t_{new} to the vocabulary \mathcal{V} .
 - 2.5 Add the rule $(t_a, t_b) \rightarrow t_{\text{new}}$ to our list of merge rules m .
 - 2.6 Replace all occurrences of the pair (t_a, t_b) in the corpus with t_{new} .

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

5.1 The BPE Algorithm

Definition (BPE Encode/Decode Algorithms)

- **Encoding:** Given a string and the ordered merge rules from dictionary creation.
 1. Split the string into a sequence of characters.
 2. For each merge rule, in the order they were created, replace all corresponding adjacent pairs in the sequence with the new token.
 3. Output the final token sequence.

5.1 The BPE Algorithm

Definition (BPE Encode/Decode Algorithms)

- **Encoding:** Given a string and the ordered merge rules from dictionary creation.
 1. Split the string into a sequence of characters.
 2. For each merge rule, in the order they were created, replace all corresponding adjacent pairs in the sequence with the new token.
 3. Output the final token sequence.
- **Decoding:** Given a token sequence.
 1. Concatenate all tokens in order to form a single string.
 2. Output the string.

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation)

Suppose the training corpus is just: "fast faster fastest slow slower slowest". We'll treat words as being separated.

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'

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

2. 1st Merge:

- We count all adjacent pairs. The pair '(s, t)' appears 6 times, which is the most frequent.
- We create a new token 'st' and add it to the vocabulary.
- Our data becomes: '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'

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

3. 2nd Merge:

- The pairs '(f, a)' and '(s, l)' are now the most frequent (3 times each). We'll pick '(f, a)' based on alphabetical order.
- New token: 'fa'. Data: 'fa st', 'fa st e r', 'fa st e st', ...

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

3. 2nd Merge:

- The pairs '(f, a)' and '(s, l)' are now the most frequent (3 times each). We'll pick '(f, a)' based on alphabetical order.
- New token: 'fa'. Data: 'fa st', 'fa st e r', 'fa st e st', ...

4. 3rd Merge:

- The most frequent pair is now '(fa, st)' (3 times).
- New token: 'fast'. Data: 'fast', 'fast e r', 'fast e st', ...

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

By repeating this process, frequent words like 'slow' and frequent suffixes like 'er' and 'est' are learned as single tokens.

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

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.

5.1 The BPE Algorithm

Example (Detailed Example of BPE Operation (cont.))

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.

5.1 The BPE Algorithm

Exercise

Suppose the corpus is “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?

5.1 The BPE Algorithm

Solution:

1. **Initial State:** Data is 'u n z i p', 'u n z i p p e d', 'z i p'.

5.1 The BPE Algorithm

Solution:

1. **Initial State:** Data is 'u n z i p', 'u n z i p p e d', 'z i p'.
2. **1st Merge:**
 - Most frequent pairs: '(z, i)' and '(i, p)' (3 times each). Pick '(i, p)' alphabetically.
 - Rule 1: 'i p' \rightarrow 'ip'.
 - Data: 'u n z ip', 'u n z ip p e d', 'z ip'.

5.1 The BPE Algorithm

Solution:

1. **Initial State:** Data is 'u n z i p', 'u n z i p p e d', 'z i p'.
2. **1st Merge:**
 - Most frequent pairs: '(z, i)' and '(i, p)' (3 times each). Pick '(i, p)' alphabetically.
 - Rule 1: 'i p' \rightarrow 'ip'.
 - Data: 'u n z ip', 'u n z ip p e d', 'z ip'.
3. **2nd Merge:**
 - Most frequent pair: '(z, ip)' (3 times).
 - Rule 2: 'z ip' \rightarrow 'zip'.
 - Data: 'u n zip', 'u n zip p e d', 'zip'.

5.1 The BPE Algorithm

Solution (cont.):

3. 3rd Merge:

- Most frequent pairs: '(u, n)' and '(n, zip)' (2 times each). Pick '(n, zip)'.
- Rule 3: 'n zip' \rightarrow 'nzip'. Data: 'u nzip', 'u nzip p e d', 'zip'.

5.1 The BPE Algorithm

Solution (cont.):

3. 3rd Merge:

- Most frequent pairs: '(u, n)' and '(n, zip)' (2 times each). Pick '(n, zip)'.
- Rule 3: 'n zip' \rightarrow 'nzip'. Data: 'u nzip', 'u nzip p e d', 'zip'.

4. 4th Merge:

- Most frequent pair: '(u, nzip)' (2 times).
- Rule 4: 'u nzip' \rightarrow 'unzip'. Data: 'unzip', 'unzip p e d', 'zip'.

5.1 The BPE Algorithm

Solution (cont.):

3. 3rd Merge:

- Most frequent pairs: '(u, n)' and '(n, zip)' (2 times each). Pick '(n, zip)'.
- Rule 3: 'n zip' \rightarrow 'nzip'. Data: 'u nzip', 'u nzip p e d', 'zip'.

4. 4th Merge:

- Most frequent pair: '(u, nzip)' (2 times).
- Rule 4: 'u nzip' \rightarrow 'unzip'. Data: 'unzip', 'unzip p e d', 'zip'.

5. 5th Merge:

- Many pairs occur once. Pick '(e, d)' alphabetically.
- Rule 5: 'e d' \rightarrow 'ed'. Data: 'unzip', 'unzip p p ed', 'zip'.

5.1 The BPE Algorithm

Exercise

Use the rules you just created to encode the string "unzipping".

Final Result and Encoding:

- **Merge rules (ordered):** '(i, p)', '(z, ip)', '(n, zip)', '(u, nzip)', '(e, d)'
- **Encoding "unzipping":**
 1. Initial: ['u', 'n', 'z', 'i', 'p', 'p', 'i', 'n', 'g']
 2. After rule 1 ('ip'): ['u', 'n', 'z', 'ip', 'p', 'i', 'n', 'g']
 3. After rule 2 ('zip'): ['u', 'n', 'zip', 'p', 'i', 'n', 'g']
 4. After rule 3 ('nzip'): ['u', 'nzip', 'p', 'i', 'n', 'g']
 5. After rule 4 ('unzip'): ['unzip', 'p', 'i', 'n', 'g']
 6. Rule 5 is not applicable.
- **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.

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.
- The neural network receives this token ID sequence and outputs a sequence of probability distributions for the next token.

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.
- 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.

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.
- 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.

5.2 The Role of BPE in the Pipeline

Remark

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.

Special Tokens and Chat AI

6. Special Tokens and Chat AI

By wrapping the neural network with encoding and decoding processes like BPE, we can realize a process that takes a string and returns a string.

6. Special Tokens and Chat AI

By wrapping the neural network with encoding and decoding processes like BPE, we can realize a process that takes a string and returns a 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.

6. Special Tokens and Chat AI

To handle this structure, we use **Special Tokens** and **Chat Templates**.

6. Special Tokens and Chat AI

To handle this structure, we use **Special Tokens** and **Chat Templates**.

- **Special Tokens** are tokens used to represent the structure of the conversation (e.g., who is speaking, start/end of a message).

6. Special Tokens and Chat AI

To handle this structure, we use **Special Tokens** and **Chat Templates**.

- **Special Tokens** are tokens used to represent the structure of the conversation (e.g., who is speaking, start/end of a message).
- A **Chat Template** is a set of rules for converting a structured conversation history into a single string containing these special tokens, which can then be fed to the tokenizer.

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."

```
1 messages = [  
2     {"role": "system", "content": "This is a test. Think as short as possible."  
3     },  
4     {"role": "user", "content": "Find 7+8."},  
5 ]
```

6.1 Chat Templates and Special Tokens

This structured data needs to be converted into a single string. The Hugging Face tokenizer's `apply_chat_template` method does this. For the Qwen model, the previous 'messages' are converted into a string like this:

```
<|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>
```

6.1 Chat Templates and Special Tokens

This string contains special tokens in addition to the normal BPE vocabulary:

- `<|im_start|>` and `<|im_end|>`: Mark the start and end of an utterance.

6.1 Chat Templates and Special Tokens

This string contains special tokens in addition to the normal BPE vocabulary:

- `<|im_start|>` and `<|im_end|>`: Mark the start and end of an utterance.
- `system`, `user`, `assistant`: Explicitly state the role of the utterance.

6.1 Chat Templates and Special Tokens

This string contains special tokens in addition to the normal BPE vocabulary:

- `<|im_start|>` and `<|im_end|>`: Mark the start and end of an utterance.
- `system`, `user`, `assistant`: Explicitly state the role of the utterance.
- `<think>` and `</think>`: Mark the start and end of the model's internal "thinking" process.

6.1 Chat Templates and Special Tokens

Particularly important is the part at the end: `<|im_start|>assistant<think>`.

6.1 Chat Templates and Special Tokens

Particularly important is the part at the end: `<|im_start|>assistant<think>`.

The language model's task is to complete a given token sequence. Models that use "Thinking" are trained on complete sequences of the form:

"Question <think> Reasoning </think> Answer"

6.1 Chat Templates and Special Tokens

Particularly important is the part at the end: `<|im_start|>assistant<think>`.

The language model's task is to complete a given token sequence. Models that use "Thinking" are trained on complete sequences of the form:

"Question <think> Reasoning </think> Answer"

Therefore, by giving *"Question <think>"* as input, the model is prompted to generate the remaining part: *"Reasoning </think> Answer"*.

6.1 Chat Templates and Special Tokens

Remark

We have stated that the goal of AI is to acquire a function that represents an appropriate input-output relationship. For a Chat AI, it seems sufficient to take a question as input and output a response.

6.1 Chat Templates and Special Tokens

Remark

We have stated that the goal of AI is to acquire a function that represents an appropriate input-output relationship. For a Chat AI, it seems sufficient to take a question as input and output a response.

However, large language models (LLMs) used in Chat AI have ended up learning a **token sequence completion function**.

6.1 Chat Templates and Special Tokens

Remark

The difference between a "question-to-answer" function and a "token sequence completion" function lies in their domains. The completion function has an unnecessarily larger domain.

6.1 Chat Templates and Special Tokens

Remark

The difference between a "question-to-answer" function and a "token sequence completion" function lies in their domains. The completion function has an unnecessarily larger domain.

It accepts not just a "*Question*", but also inputs like "*Question Beginning of the answer*". This allows one to control the direction of the response.

6.1 Chat Templates and Special Tokens

Remark

The difference between a "question-to-answer" function and a "token sequence completion" function lies in their domains. The completion function has an unnecessarily larger domain.

It accepts not just a "*Question*", but also inputs like "*Question Beginning of the answer*". This allows one to control the direction of the response.

This property is both convenient and can be used for attacks, such as Prefix Injection or Prefilling Attacks [3, 4].

Conclusion and Future Outlook

7.1 The Overall Picture of the Pipeline

Integrating the discussions so far, let's reconfirm the overall picture of the pipeline. It is a composition of a series of functions:

1. **Template Renderer (Renderer):** Converts structured 'messages' into a single prompt string.

7.1 The Overall Picture of the Pipeline

Integrating the discussions so far, let's reconfirm the overall picture of the pipeline. It is a composition of a series of functions:

1. **Template Renderer (Renderer)**: Converts structured 'messages' into a single prompt string.
2. **Tokenizer (Encoder)** (Tokenizer): Converts the prompt string into a sequence of token IDs.

7.1 The Overall Picture of the Pipeline

Integrating the discussions so far, let's reconfirm the overall picture of the pipeline. It is a composition of a series of functions:

1. **Template Renderer (Renderer)**: Converts structured 'messages' into a single prompt string.
2. **Tokenizer (Encoder)** (Tokenizer): Converts the prompt string into a sequence of token IDs.
3. **Neural Network and Sampler** ($\text{Sampler}_{f_{\theta^*}}$): Takes a token ID sequence and returns another. This is the core parametric function.

7.1 The Overall Picture of the Pipeline

Integrating the discussions so far, let's reconfirm the overall picture of the pipeline. It is a composition of a series of functions:

1. **Template Renderer (Renderer)**: Converts structured 'messages' into a single prompt string.
2. **Tokenizer (Encoder) (Tokenizer)**: Converts the prompt string into a sequence of token IDs.
3. **Neural Network and Sampler (Sampler _{f_{θ^*}})**: Takes a token ID sequence and returns another. This is the core parametric function.
4. **Detokenizer (Decoder) (Detokenizer)**: Converts the generated token IDs back into a human-readable string.

7.1 The Overall Picture of the Pipeline

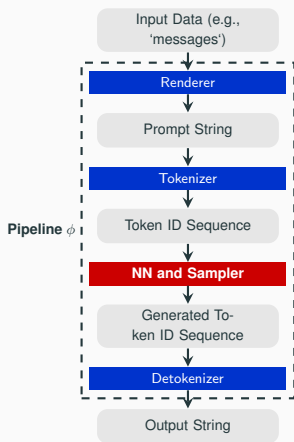


Figure 2: Detailed Inference pipeline.

Mathematically, the entire pipeline ϕ is 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^*} , the entire pipeline is also a parametric function dependent on the checkpoint θ^* .

7.2 Summary of Today's Lecture

- In practical AI, a **pipeline** that combines multiple models and algorithms is used. This is a key strategy for balancing efficiency and performance.

7.2 Summary of Today's Lecture

- In practical AI, a **pipeline** that combines multiple models and algorithms is used. This is a key strategy for balancing efficiency and performance.
- An important component in NLP pipelines is **tokenization**, which handles the conversion between human-readable strings and numerical data for the network.

7.2 Summary of Today's Lecture

- In practical AI, a **pipeline** that combines multiple models and algorithms is used. This is a key strategy for balancing efficiency and performance.
- An important component in NLP pipelines is **tokenization**, which handles the conversion between human-readable strings and numerical data for the network.
- **Byte Pair Encoding (BPE)** is an effective subword tokenization algorithm that can handle unknown words while keeping sequence lengths manageable.

7.2 Summary of Today's Lecture

- In practical AI, a **pipeline** that combines multiple models and algorithms is used. This is a key strategy for balancing efficiency and performance.
- An important component in NLP pipelines is **tokenization**, which handles the conversion between human-readable strings and numerical data for the network.
- **Byte Pair Encoding (BPE)** is an effective subword tokenization algorithm that can handle unknown words while keeping sequence lengths manageable.
- In Chat AI, **special tokens** and **chat templates** are used to represent conversational structure and control the model's behavior.




7.3 Preview of the Next Lecture


This time, we focused on tokenization, which handles the input to the neural network.

7.3 Preview of the Next Lecture

This time, we focused on tokenization, which handles the input to the neural network.

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."

-  Philip Gage.
A new algorithm for data compression.
C Users Journal, 12(2):23–38, 1994.
-  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.
-  Alexander Wei, Nika Haghtalab, and Jacob Steinhardt.
Jailbroken: How does llm safety training fail?
Advances in Neural Information Processing Systems, 36:80079–80110, 2023.

 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.