UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

CIS 6930 Special Topics in Large Language Models

LLM Tokenization

Outline

- Introduction of Tokenization

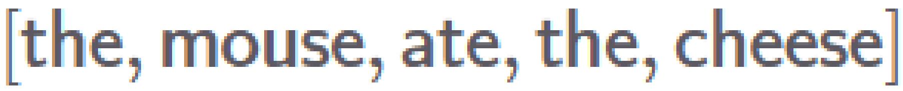
  (what is tokenization, why we need tokenization)

- Naïve ways of Tokenization

- Classical Algorithms for Tokenization used in LLMs

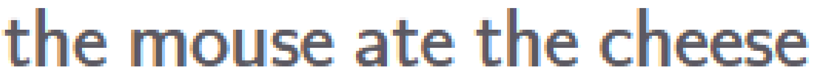
What is Tokenization

- Recall: a LM is a probability distribution over a sequence of tokens where each token comes from some vocabulary V



[the, mouse, ate, the, cheese]

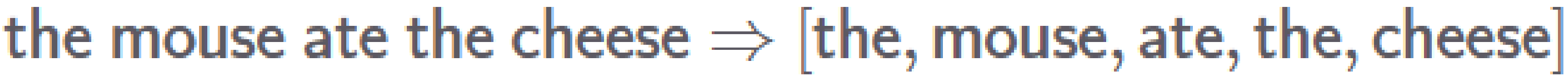
- However, natural language doesn’t come as a sequence of tokens, but just as a string



the mouse ate the cheese

What is Tokenization

- Tokenization is a process that converts any text string into a sequence of tokens, which can be words, sub-words, characters



the mouse ate the cheese $\Rightarrow$ [the, mouse, ate, the, cheese]

- These tokens are the smallest units of meaning in a text that can be processed by a LM

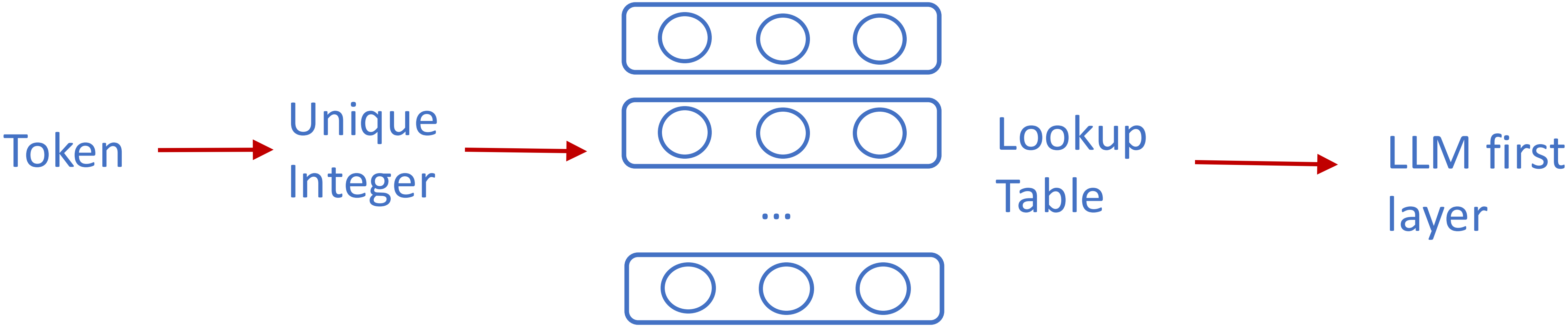
Why we need Tokenization

- The language data is in textual form, but language model (or any machine learning models) process numbers, and tokenization helps convert text into numbers that is readable by models

- LLM training and inference workflows need to process each token as unique integer, and map each unique integer into a vector embedding, and feed vector embedding as input into language model

Why we need Tokenization

- LLM training workflows: text -> tokens -> unique integer -> vector embeddings in the lookup table -> input into language model



Token

Unique Integer

…

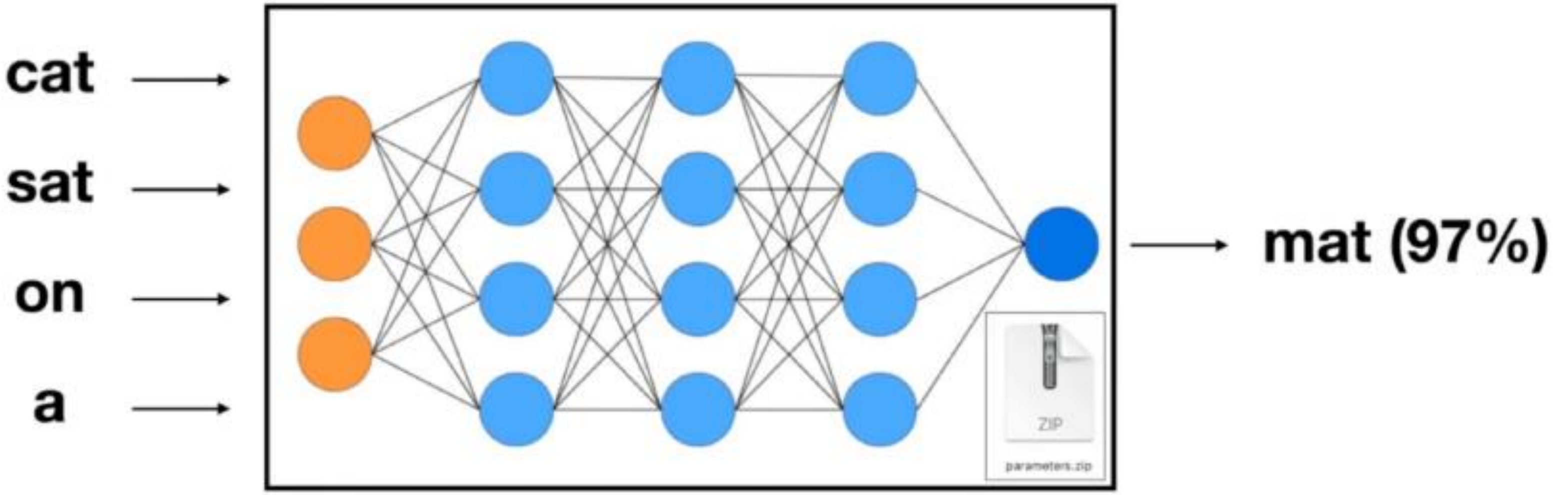
Lookup Table

LLM first layer

Unique integer corresponds to row id in the lookup table

Lookup table: each row represents vector embedding of each token

Why we need Tokenization

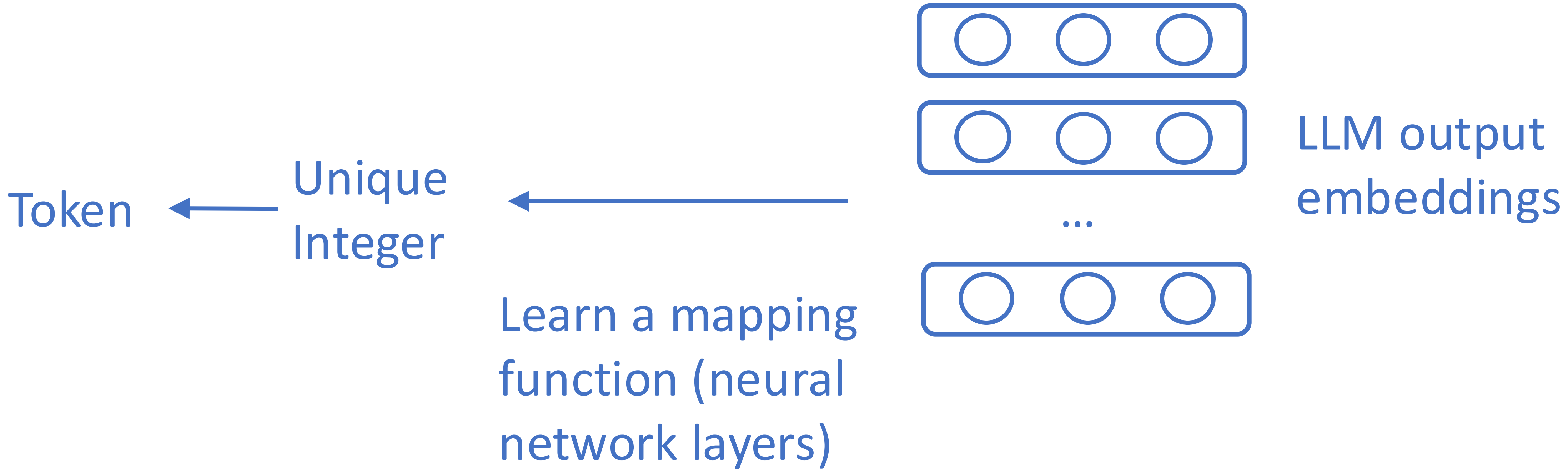


$$f(\text{“cat sit on a”};\theta) = \text{“mat”}$$

Model inside illustration

Why we need Tokenization

- LLM inference workflows: vector embeddings of each token (after layer of layer propagation in LLM) -> unique integer for each token -> text



Token

Unique Integer

Learn a mapping function (neural network layers)

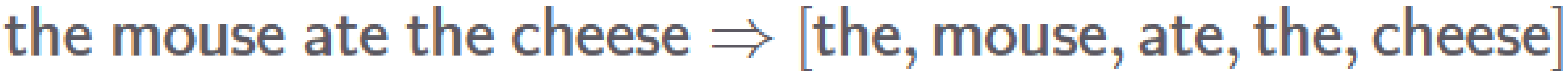
…

LLM output embeddings

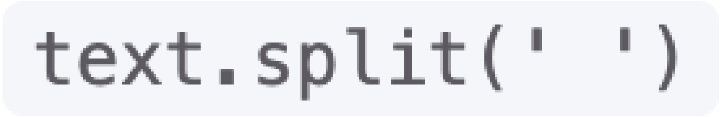
Naïve Ways of Tokenization

Split by spaces

The simplest solution is to split by spaces



the mouse ate the cheese $\Rightarrow$ [the, mouse, ate, the, cheese]

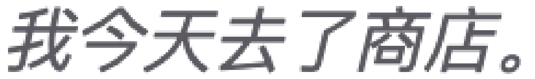


```
text.split(' ')
```

Split by spaces

There exists several problems:

- That doesn’t work for some languages, such as Chinese, where sentences are written without spaces between words



我今天去了商店。

- There are languages, such as German, that have long compound words (e.g., Abwasserbehandlungsanlange)

- Even in English, there are hyphenated words(e.g., “father-in-law”) or contraction (e.g., “don’t”), which should get split up

Character-level Tokenization

What about character-level tokenization? (token = character)

- Step 1: Find all unique characters in your corpus
- Step 2: Decompose each string into characters
- Step 3: Map each character into unique integer
- Step 4: Create Lookup Table (convert integer into vector embedding)
- Step 5: LM is then trained on these vector embeddings of text string

Character-level Tokenization

There exists several problems:

- Difficult when dealing with diverse languages:

This method does not account for languages other than the language for which you created vocabulary lookup tables

- Difficult when dealing with long sequences:

The vocabulary size is small (number of characters), but the sequence length of tokenized sentences will be much larger. Because LLM usually has fixed window size, LLM will process much less data at a time.

Algorithm 1 – Byte Pair Encoding Tokenization

Byte-Pair Encoding (BPE) Tokenization

- Used by OpenAI for tokenization when training the GPT models. It’s used by LLMs like GPT, GPT-2, RoBERTa, BART, and DeBERTa.

- Core Idea: start with each character as the token, and then iteratively **combine** the **token pair** that **co-occur** together **most often** into new token, and gradually form a full vocabulary

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 1: Initialize the vocabulary as a set of characters

Our corpus ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Initial Vocabulary: `["b", "g", "h", "n", "p", "s", "u"]`

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 2: Count how often each **pair** of tokens co-occur

Our corpus

("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)

Vocabulary `["b", "g", "h", "n", "p", "s", "u"]`

Count frequency of token pairs: `("h", "u")` 15, `("u", "g")` 20 …

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 3: Merge the most frequent token pair into new token

Our corpus ("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)

Vocabulary `["b", "g", "h", "n", "p", "s", "u"]`

Merge Rule learned by tokenizer `("u", "g") -> "ug"`

Add newly combined token into the vocabulary add `"ug"` into V

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 4: Repeat Iteratively (iteratively merge the token pair that co-occur together most often into new token, and add new token into Vocabulary)

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]

Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)

Most frequent pair is（"u", "n"） 16 times

Add the second merge rule（"u", "n"） -> "un"

Add new token "un" into the vocabulary

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 4: Repeat Iteratively (iteratively merge the token pair that co-occur together most often into new token, and add new token into Vocabulary)

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]

Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)

Most frequent pair is ("h", "ug") 15 times

Add the third merge rule ("h", "ug") -> "hug"

Add new token "hug" into the vocabulary

Byte-Pair Encoding (BPE) Tokenization – Training

- Step 5: Stop when we reach the desired vocabulary size

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]

Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)

Byte-Pair Encoding (BPE) Tokenization – Training

- Training Outcome

(1) Vocabulary ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]

(2) Merge Rules

```
("u", "g") -> "ug”
("u", "n") -> "un”
("h", "ug") -> "hug"
```

Byte-Pair Encoding (BPE) Tokenization – Tokenize

- Step 1: Splitting each words into individual tokens
- Step 2: Applying the Merge rules learned in order

Merge Rules

```
("u", "g") -> "ug”, ("u", "n") -> "un”, ("h", "ug") -> "hug”
```

New Word: `"bug" -> ["b", "u", "g"] -> ["b", "ug"]`

Byte-Pair Encoding (BPE) Tokenization – Tokenize

- Step 1: Splitting each words into individual tokens
- Step 2: Applying the Merge rules learned in order

Merge Rules

```
("u", "g") -> "ug”, ("u", "n") -> "un”, ("h", "ug") -> "hug”
```

New Word:
```
"thug" -> ["[UNK]","h", "u", "g"]
       -> ["[UNK]","h", "ug"]
       -> ["[UNK]", "hug"]
```

Algorithm 2 – WordPiece Tokenization

WordPiece Tokenization

- Used by Google for tokenization when training BERT model. It’s used by language models like BERT, DistilBERT, MobileBERT, MPNET etc.

- Core Idea: similar to Byte-Pair Encoding (BPE) tokenization (iteratively merge token pair that co-occur most often into new token), but different in **vocabulary initialization** and **frequency score calculation**

WordPiece Tokenization – Training

- Step 1: Initialize the vocabulary as a set of characters

Difference: add a prefix ## to all the character inside the word

Our corpus ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Initial Vocabulary:

```
["b", "h", "p", "##g", "##n", "##s", "##u"]
```

(All the beginning character and inside character preceded by the prefix)

WordPiece Tokenization – Training

- Step 2: Count how often each **pair** of tokens co-occur

Difference: $\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$

Our corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)

Vocabulary ["b", "h", "p", "##g", "##n", "##s", "##u"]

Calculate score of token pairs: ("##u", "##g") 20/(36*20)

("##g", "##s") 5/(20*5) …

WordPiece Tokenization – Training

- Step 3: Merge the token pair with the highest score into new token

Our corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)

Vocabulary ["b", "h", "p", "##g", "##n", "##s", "##u"]

Merge Rule learned by tokenizer ("##g", "##s") -> ("##gs")

Add newly combined token into the vocabulary add ″##gs" into V

WordPiece Tokenization – Training

- Step 4: Repeat Iteratively (iteratively merge the token pair with the highest score, and add new token into Vocabulary)

Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs"]

Corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##gs", 5)

The pair with highest score is ("h", "##u")

Merge rule ("h", "##u") -> "hu"

Add new token "hu" into the vocabulary

WordPiece Tokenization – Training

- Step 4: Repeat Iteratively (iteratively merge the token pair with the highest score, and add new token into Vocabulary)

Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu"]

Corpus: ("hu" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs", 5)

The pair with highest score is（"hu"， "##g"）

Merge rule（"hu"， "##g"） -> "hug"

Add new token "hug" into the vocabulary

WordPiece Tokenization – Training

- Step 5: Stop when we reach the desired vocabulary size

Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]

Corpus: ("hug", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("hu" "##gs", 5)

WordPiece Tokenization – Training

- Training Outcome

(1) Vocabulary ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]

~~(2) Merge Rules~~

Difference: only saves the final vocabulary, not the merge rule learned

WordPiece Tokenization – Tokenize

- Step 1: Finding the longest substring that is in the vocabulary
- Step 2: Splitting It

Vocabulary ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]

New Word: "hugs" -> "hug" is the longest substring starting from the beginning that is inside the Vocabulary V

-> split it ["hug", "##s"]

-> continue with "##s" which is already in V

WordPiece Tokenization – Tokenize

- Step 1: Finding the longest substring that is in the vocabulary
- Step 2: Splitting It

Vocabulary ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs", "hu", "hug"]

New Word: ″bugs" -> ″b" is the longest substring

-> split it [″b", "##ugs"]

-> continue with "##ugs″, "##u″ is the longest substring

-> split it ["b", "##u, "##gs"]

-> continue with "##gs" which is already in V

Algorithm 3 – Unigram Tokenization

Unigram Tokenization

- Used by language models like T5, XLNet, ALBERT, Big Bird

- Core Idea: Different from BPE and WordPiece that start from a small vocabulary and then iteratively expand, it starts from a big vocabulary and removes tokens from it until reaching the desired vocabulary size

Unigram Tokenization – Tokenize

- Step 1: List out all the possible segmentations of a new word
- Step 2: Compute the probability of each possible segmentation based on Unigram LM
- Step 3: Select the one segmentation with the highest probability as the final tokenization

Unigram Tokenization – Tokenize

- Step 1: List out all the possible segmentations of a new word

Example: Final Vocabulary and its corresponding frequency in corpus

("h", 15) ("u", 36) ("g", 20) ("hu", 15) ("ug", 20) ("p", 17) ("pu", 17) ("n", 16) ("un”, 16) ("b", 4) ("bu", 4) ("s", 5) ("hug", 15) ("gs", 5) ("ugs", 5)

New Word: "pug"

All the possible segmentations: ["p", "u", "g"], ["p", "ug"], ["pu", "g"]

Unigram Tokenization – Tokenize

- Step 2: Compute the prob of each segmentation based on Unigram LM

Example: Final Vocabulary and its corresponding frequency in corpus

("h", 15) ("u", 36) ("g", 20) ("hu", 15) ("ug", 20) ("p", 17) ("pu", 17) ("n", 16) ("un”, 16) ("b", 4) ("bu", 4) ("s", 5) ("hug", 15) ("gs", 5) ("ugs", 5)

$$P([``pu", ``g"]) = P(``pu") \times P(``g") = \frac{5}{210} \times \frac{20}{210} = 0.0022676$$

New Word: "pug"

["p", "u", "g"] : 0.000389 ["p", "ug"] : 0.0022676 ["pu", "g"] : 0.0022676

Unigram Tokenization – Tokenize

- Step 3: Select the one segmentation with the highest probability as the final tokenization

New Word: "pug"

["p", "u", "g"] : 0.000389 ["p", "ug"] : 0.0022676 ["pu", "g"] : 0.0022676

Choose ["p", "ug"] or ["pu", "g"] depending on which is encountered first

Unigram Tokenization – Training

- Core Idea:

Calculate a **loss of corpus** based on current vocabulary V

Remove each token from V and re-calculate loss of corpus

Select the token that results in the least increase in loss of corpus

(least change, less effect, less needed, should be removed)

Unigram Tokenization – Training

- Step 1: Starting from a big vocabulary, e.g., listing out all the possible substrings in corpus, or applying BPE on the initial corpus with a large vocabulary size

Our corpus ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Initial Vocabulary: ["h", "u", "g", "hu", "ug", "p", "pu", "n", "un", "b", "bu", "s", "hug", "gs", "ugs"]

Unigram Tokenization – Training

- Step 2: Calculate a Loss of corpus by tokenizing every word in the corpus
- **Loss of each word = – log (P(word)), Loss of corpus = sum(Loss of all the word in the corpus)**

Our corpus ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

Prob of each word: "hug": ["hug"] *(score 0.071428)* "pug": ["pu", "g"] *(score 0.007710)* "pun": ["pu", "n"] *(score 0.006168)* "bun": ["bu", "n"] *(score 0.001451)* "hugs": ["hug", "s"] *(score 0.001701)*

Loss of corpus = 10 * (-log(0.071428)) + 5 * (-log(0.007710)) + 12 * (-log(0.006168)) + 4 * (-log(0.001451)) + 5 * (-log(0.001701)) = 169.8

Unigram Tokenization – Training

- Step 3: For each token in the current vocabulary V, we remove it from V, and recalculate the Loss of corpus based on the updated vocabulary
- **Loss of each word = - log (P(word)), Loss of corpus = sum(Loss of all the word in the corpus)**

- Step 4: Select the token w that results in the **least increase** in Loss

(Loss of corpus based on vocabulary V' after removing token w – Loss of corpus based on vocabulary V before removing w)

Remove this token w from the current vocabulary V and update V

Summary of Tokenization Algorithms

- Byte-Pair Encoding (BPE) & WordPiece Tokenization

Core Idea: start with each character as the token, and then iteratively merge the token pair that co-occur together most often into new token

Difference: vocabulary initialization and frequency score calculation

- Unigram Tokenization

Core Idea: start from a big vocabulary and iteratively remove the token that results in the least increase in loss of corpus

Thank you!

UF Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

LEADING THE CHARGE, CHARGING AHEAD

References

https://huggingface.co/learn/llm-course/en/chapter6/7?fw=pt

https://medium.com/thedeephub/all-you-need-to-know-about-tokenization-in-llms-7a801302cf54