Tokenizers/Tokenization Notes

Tokenizing a text is splitting it into sub-words, which are then converted to ids through a look-up table

3 main types of tokenizers used in transformers

- Byte Pair Encoding
- Word Piece
- Sentence Piece

sentence: "Don't you love transformers? We sure do."

- a simple way to split this text is to split by spaces, which would give:

["you", "love", "transformers", "We", "sure", "do."]

- here you can notice that punctuation is attached to the word to the words "Transformer?" and "do.", this is suboptimal. We should take the punctuation into account so that a model does not have to learn a different representation of a word and every possible punctuation symbol that could follow it, which would explode the number of representations the model has to learn
- a pre-trained only performs properly if you feed it an input that was tokenized with the same rules that were used to tokenize its training data
- space and punctuation tokenization usually generates a very big vocabulary (the set of all unique words and tokens used)

Character-based tokenizers

- Character tokenization is very simple and would greatly reduce memory and time complexity it makes it much harder for the model to learn meaningful input representations
- Example: learning a meaningful context-independent representation for a singular letter "t" is much harder than learning a context-independent representation for the word "today"
- character-tokenization is often accompanied by A loss in performance

Subword Tokenization

- this method relies on the principal that frequently used words should not be split into smaller sub-words, but rare words should be decomposed into meaningful subwords
- this method allows a model to have A reasonable vocabulary size while being able to learn meaningful context-independent representations
- this method also allows the model to process words it has never seen before, by decomposing them into known sub-words

Byte-Pair Encoding

- Introduced in Neural Machine Translation of Rare Words with Subword Units
- a tokenization technique that helps break words into smaller words (subwords or symbols) to handle rare or unseen words more effectively
- BPE relies on a pre-tokenizer that splits the training data into words. This can be done by using spaces or more advanced methods, like splitting based on punctuation or special characters

1. Basic Vocabulary:

After pre-tokenization, BPE creates a base vocabulary by splitting each word into individual characters or symbols. For example, the words:

- "hug","pug","pun","bun","hugs"
  - these are split into their base symbols
- ["h","u","g"],["p","u","g"], ["p","u","n"], ["b","u","n"],["h","u","g","s"]
- the base vocabulary would be all the unique symbols used in these words:
  - ["b","g","h","n","p","s","u"]

2. Merge Symbols

BPE then looks for the most frequent pairs of symbols and merges them into a new subword. For example, in the above words:

- The pair "u" and "g" appears most frequently (in "hug","pug","hugs"), so it merges "u" and "g" to form "ug". The word hug now becomes:
  - ["h","ug"],["p","ug"],["p","u","n"],["b","u","n"],["h","ug","s"]
- the vocabulary now includes "ug"
  - ["b","g","h","n","p","s","u","ug"]

3. Continue Merging

- BPE keeps merging the most frequent pairs of symbols. The next common pair might be "u" and "n", appearing in both "pun" and "bun". So it merges them into "un"
  - ["h","ug"],["p","ug"],["p","un"],["b","un"],["h","ug","s"]
- After several merges, the vocabulary grows, adding new subwords like "hug" and "un"

4. Stopping at a target vocabulary size

- BPE continues merging pairs until it reaches the desired vocabulary size, a hyperparameter (a value chosen during model training)
- For example, GPT-2 uses a vocabulary size of 40,478, includes the base characters plus 40,000 merged sub-words

6. Using the learned rules:

- Once BPE learns the merge rules, it applies them to new words
- A new word like "mug" might be tokenized as ["<unk>", "ug"] if "m" is not in the vocabulary

Byte-Level Byte-Pair Encoding (BPE)

- A method that starts with bytes rather than characters or subwords as the base vocabulary

1. Base Vocabulary of bytes

- Instead of using characters as the base vocabulary, GPT-2 uses bytes, which means every possible byte (a number between 0 and 255) is part of the base vocabulary contains exactly 256 tokens that represent every possible byte.
- GPT-2 starts by creating a base vocabulary of 256 unique tokens, where each token corresponds to one of the 256 possible byte values (from 0 to 255)

2. Handling all characters

- Since every character in Unicode can be represented as a sequence of bytes, the tokenizer can break down any text (including emojis, foreign characters, etc.) into bytes, ensuring full coverage of all possible inputs. This also simplifies the base vocabulary, reducing the need for larger character sets

3. Merging bytes to form subwords

- After starting with the 256-byte vocabulary, BPE merges frequent byte sequences to form larger subwords. This is similar to how standard BPE works but operates on byte sequences instead of characters
  - Over time, frequently occurring byte pair (like sequences representing common words or subwords) are merged to form complex tokens.
  - GPT-2 performs 50,000 merges, resulting in a final vocabulary size of 50,257 (256 bytes, 1 special end of text token, and 50,000 merged subwords).

4. No <unk> symbol

- By using bytes as the base, every possible character can be broken into byte sequences, so GPT-2's tokenizer never encounters an unknown token. Even if the input contains rare or non-standard characters, the model can represent them with byte-level tokens

Word Piece

- Like BPE, wordpiece is a subword tokenization algorithm that helps break down words into smaller parts to better handle unknown or rare words
- Starts with a vocabulary of characters and progressively merges pairs of symbols to form new subwords
- However wordpiece uses a different merging strategy compared to BPE: instead of choosing the most frequent pair of symbols, wordpiece selects the pair that maximizes the likelihood of training data

1. Initial vocabulary of characters

- Wordpiece by including every character from the training data in the initial vocabulary. This means that, at the start, each word is split into individual characters
- For example, for the words:
  - "hug", "pug", "pun", "bun", "hugs"
  - ["h", "u", "g"], ["p", "u", "g"], ["p", "u", "n"], ["b", "u", "n"], ["h", "u", "g", "s"]

2. Merging based on Maximizing likelihood:

- Instead of merging the most frequent symbol pairs (as bpe does), wordpiece evaluates which pair of symbols, if merged, will maximize the likelihood of the training data
- The likelihood here refers to how well the subwords represent the original training data after each merge. In other words, wordpiece looks at which merges are most useful in modeling the data
- Maximizing likelihood formula: Word piece chooses the pair of symbols (a,b) that maximizes:

$$\frac{P(A \text{ and } B)}{P(A)P(B)}$$

- The joint probability of the merged symbol divided by the product of the individual probabilities of the two symbols a and b
- If merging two symbols results in a large improvement in representing the data (the new subword is much more useful than keeping the two symbols seperate), the model will choose to merge them

3. What does maximum likelihood mean?

- Intuition: wordpiece looks at the tradeoff of merging two symbols. It checks if merging them makes sense in terms of improving the representation of the text

4. Evaluating merges carefully

- Word piece is slightly more cautious than BPE. It evaluates the cost of merging two symbols and merges them only if the gain outweighs the loss

5. Progressively learning Merges

- Wordpiece continues merging symbol pairs until it reaches the desired vocabulary size. Like BPE the vocabulary size is a hyperparameter defined during training

Unigram

- Unigram tokenization is a subword tokenization method where the algorithm starts with a large base vocabulary and progressively trims it down to the desired size.
- Instead of building up the vocabulary by merging symbols (like BPE and word piece), Unigram works by removing symbols that contribute the least to modeling the data.

1. Initialize a large vocabulary

- The unigram algorithm starts with a very large vocabulary that includes:
  - All pretokenized words
  - Common sub-strings or subwords that frequently appear in the training data

- the base vocabulary is typically large enough to include different possible segmentations of words, providing many options for tokenizing text

2. Define a loss function

- Unigram is based on a language model where it calculates the likelihood of the training data given the current vocabulary. The goal is to minimize the loss, defined as the negative log-likelihood of the training data. The formula:

$$L = -\sum_{i=1}^{N} \log\left(\sum_{x \in S(x_i)} p(x)\right)$$

where:

- x_i is a word in the training data
- $S(x_i)$ is the set of all possible ways to tokenize x_i
- $p(x)$ is a probability of a given tokenization x

3. Evaluate the impact of each symbol

- For each symbol in the vocabulary, the algorithm computes how much the overall loss would increase if the symbol were removed. The key idea is to measure the importance of each symbol in modeling the training data
- Symbols that don't contribute much to reducing the loss, are candidates for removal

4. Remove least important symbols

- The algorithm removes a certain percentage (typically 10 or 20%) of the least important symbols - those that increase the loss the least when removed

5. Ensure base characters are kept

- Unigram always keeps the base characters, so even if most of the vocabulary is trimmed down, the tokenizer can split any word into individual characters. This ensures that the tokenizer can handle any word even if it was not seen it during training

Sentence Piece

- was introduced to solve the problem where spaces are assumed to seperate words, which does not hold true for all languages
- It treats the text as a continuous stream of characters, including spaces as part of the character set
- It then applies either byte-pair encoding or unigram to create subwords and build a vocabulary

1. Input treated as raw character stream
   - It reads the input text as a raw character sequence. This means even spaces are treated as characters and included in the base vocabulary
   - The sentence "I love Pizza" is treated as a sequence of characters:
     ["I", " ", "l", "o", "v", "e", " ", "p", "i", "z", "z", "a"]
2. Spaces are part of the vocabulary
   - Spaces are assigned their own symbol in the vocabulary. The space character is often represented as an underscore (_)
3. Use of bpe or unigram algorithm
   - Sentence piece can use either bpe or unigram to build its subword vocabulary
     - If bpe is used, the algorithm starts with characters and merges frequent pairs to form subwords just as BPE
     - If unigram is used, sentence piece starts with a large vocabulary and progressively removes the least important symbols