**Tokenization in Large Language Models (LLMs)**

**Importance of Tokenization**

* Tokenization is the process of breaking streaming text into units (tokens), essential for determining word similarities and handling unknown words.
* Traditional tokenization methods used delimiters (space, tab, full stop, newline), but these methods proved inefficient for handling unknown or complex tokens.
* Tokenization plays a crucial role in neural machine translation (NMT) and LLMs.

**Types of Tokenization Strategies**

1. **Word-Level Tokenization:**
   * Uses whole words as tokens.
   * Issues: Exponential vocabulary size, inability to handle unknown words.
2. **Character-Level Tokenization:**
   * Uses individual characters as tokens.
   * Issues: Fixed vocabulary but lacks meaningful word representation.
3. **Subword-Level Tokenization:**
   * Breaks words into meaningful subunits.
   * Example: "catching" → ["catch", "ing"].
   * Used in **FastText**, **BPE**, **WordPiece**, **Unigram LM**.

**Role of Tokenization in Training vs. Testing**

* **Training Phase:**
  + Builds vocabulary by processing the entire corpus.
  + Identifies frequent tokens.
* **Testing Phase:**
  + Uses learned vocabulary to segment new text.
  + Uses a **Token Segmenter** to tokenize input text based on predefined vocabulary and rules.

**Modern Tokenization Techniques**

1. **Byte Pair Encoding (BPE)**
   * Originally for text compression.
   * Used in **GPT, GPT-2, RoBERTa, BERT**.
   * Steps:
     1. **Pre-tokenization:**
        + Cleans corpus (removes punctuations, normalizes text).
        + Splits text into words using spaces and punctuation.
     2. **Base Vocabulary Creation:**
        + Extracts unique characters from the corpus.
        + Forms the starting vocabulary (e.g., {a, b, c, d, ..., z}).
     3. **Merging Step:**
        + Iteratively combines the most frequent character pairs (bigrams, trigrams, etc.) to form subwords.
        + Stops when the vocabulary reaches a predefined size.
     4. **Example:**
        + Given words: [cat, bat, bag, tag, cats].
        + Frequency analysis of character pairs:
          - (C, A) = 15, (A, T) = 20, (B, A) = 17...
        + The most frequent pair (A, T) is merged into **"AT"**.
        + Repeat until a stable vocabulary is built.
2. **WordPiece Encoding (Used in BERT)**
   * Similar to BPE but uses likelihood estimation rather than frequency count for merging.
   * Applied in **BERT, T5, ALBERT**.
3. **Unigram Language Model Tokenization** (Used in SentencePiece)
   * Unlike BPE and WordPiece, starts with a large vocabulary and gradually removes less useful tokens based on probability models.
   * Used in **T5, XLM-R**.

**Key Takeaways**

* **Tokenization is an active research area** and critical for NLP models.
* **Subword tokenization** balances between word-level and character-level approaches.
* **BPE, WordPiece, and Unigram LM** are the dominant tokenization methods in modern LLMs.
* **Pre-training and inference tokenization processes differ**, ensuring consistency and efficiency in model deployment.

**WordPiece Tokenization - Summary for Revision**

**🔹 Overview**

* WordPiece Tokenization is similar to BPE (Byte-Pair Encoding) but with a **key difference in the merging criteria**.
* Instead of using **frequency** for merging subwords, WordPiece uses **likelihood (mutual information-like score)**.
* First introduced in **2012**, and used in many **Google models**: BERT, DistilBERT, MobileBERT, ALBERT, etc.

**🔹 Key Differences from BPE**

| **Feature** | **BPE** | **WordPiece** |
| --- | --- | --- |
| Merging Criterion | Based on **frequency** | Based on **likelihood/score** |
| Merging Rules Storage | Yes | **No** (Only vocabulary is stored) |
| Merge Symbol Format | Plain text | Uses ## (double hash) for subwords |
| Decoding during Inference | Uses merge rules | Uses **longest subword match** |

**🔹 Merging Criterion (Score Function)**

* Score used to decide which subwords to merge:

Score(A,B)=P(AB)P(A)⋅P(B)=freq(AB)freq(A)⋅freq(B)\text{Score}(A, B) = \frac{P(AB)}{P(A) \cdot P(B)} = \frac{freq(AB)}{freq(A) \cdot freq(B)}Score(A,B)=P(A)⋅P(B)P(AB)​=freq(A)⋅freq(B)freq(AB)​

* This is inspired by **PMI (Pointwise Mutual Information)**.
* It rewards **specific co-occurrence** and **penalizes** merges of frequently occurring but **non-cohesive** subword pairs.

**🔹 Token Format**

* WordPiece uses ## to indicate **non-initial** (i.e. internal) subwords.
* Example:
  + "hello" → "h", "##e", "##l", "##l", "##o"
  + "token" → "t", "##o", "##k", "##e", "##n"

**🔹 Training Time Steps**

1. **Pre-tokenization**:
   * Corpus is split on whitespace, tabs, punctuations, etc.
   * Characters are initially treated as tokens.
     + Starting characters → stored as-is.
     + Internal characters → prefixed with ##.
2. **Base Vocabulary Initialization**:
   * Includes all individual characters.
   * Starts with character-level tokens like: t, ##o, ##k, ##e, ##n, etc.
3. **Iterative Merging**:
   * Identify pairs (e.g., ##s and ##u) with highest **likelihood score**.
   * Merge to form new token (e.g., ##su).
   * Update the vocabulary and continue until a specified vocab size is reached.
4. **Vocabulary Growth**:
   * Vocabulary keeps adding new tokens formed from merging.
   * Merging continues using bigrams, trigrams, etc.
   * Ties in score can be broken arbitrarily or deterministically.

**🔹 Training Example Summary**

* Sample corpus: "sunflower", "flowers", etc.
* All initial characters go in as-is; internals have ##.
* Compute frequency and score for each candidate merge.
* Choose the highest scoring pair (e.g., s, ##u).
* Merge them → su, update vocabulary.
* Repeat till vocab size limit is reached.

**🔹 Inference (Test Time) Steps**

* **No merging rules stored**—only vocabulary is available.
* Tokenization is done by **greedy longest-match-first** algorithm.

**Steps:**

1. Start from the beginning of the word.
2. Find the **longest substring** present in vocabulary.
3. Split it and repeat for the remaining part.
4. Continue until full word is tokenized.

**🔹 Inference Example**

Given word: "fused"  
Vocabulary: ["f", "##u", "##s", "##e", "##d", "##fu", ...]

* Step-by-step:
  1. Longest match from start: f → keep.
  2. Remaining: "used" → longest match: ##u
  3. Remaining: "sed" → ##s, ##e, ##d
* Final tokens: ["f", "##u", "##s", "##e", "##d"]

**🔹 Advantages of WordPiece**

* Better handling of **rare or unknown words**.
* Prevents merging of high-frequency but **unrelated** subwords.
* Efficient vocabulary utilization.
* Easy decoding using **just vocabulary** without merge rules.

**🔹 Key Takeaways**

* Merging is based on **score, not frequency**.
* Uses **##** to indicate subwords (except first token).
* Stores **only vocabulary**, not merging rules.
* Decoding uses **longest-match-first** strategy.
* Inspired by **PMI-like** approach for better semantic splits.

**Unigram Tokenization (Used in SentencePiece)**

**🔹 Where is it used?**

* **SentencePiece** tokenizer.
* Models like **T5, mBERT, XLM-R, XLNet**, etc.

**⚙️ Core Concept**

Unigram tokenization is inspired by the **unigram language model**, where:

The probability of a sequence of tokens is assumed to be the **product of the individual probabilities** of each token, assuming independence.

Given a sequence W1 W2 ... Wn:

P(W1,W2,...,Wn)=P(W1)×P(W2)×...×P(Wn)P(W1, W2, ..., Wn) = P(W1) \times P(W2) \times ... \times P(Wn)P(W1,W2,...,Wn)=P(W1)×P(W2)×...×P(Wn)

**❗ Motivation**

* In **BPE** and **WordPiece**, for each word, there is **only one segmentation**.
* However, in reality, **multiple valid segmentations** may exist.
* **Unigram Tokenizer** allows **multiple segmentations** per word and picks the most probable one using a probabilistic model.

**🔄 Main Strategy**

1. **Start with a large base vocabulary** (can include all possible substrings, characters, or even generated via BPE).
2. **Iteratively reduce** the vocabulary to a target size using the \*\*EM

**🔹 Unigram Tokenization – Deep Notes**

**📌 What is Unigram Tokenization?**

* **Used in:** SentencePiece (a tokenization library used in T5, mBERT, XLNet, etc.)
* **Concept:** Based on **Unigram Language Models**.
* Unlike BPE and WordPiece, which produce **only one segmentation** per word, unigram allows for **multiple possible segmentations**.

**📌 Unigram Language Model Refresher**

* Assumes **tokens are independent**.
* Probability of sequence:

P(w1,w2,...,wn)=P(w1)⋅P(w2)⋅...⋅P(wn)P(w\_1, w\_2, ..., w\_n) = P(w\_1) \cdot P(w\_2) \cdot ... \cdot P(w\_n)P(w1​,w2​,...,wn​)=P(w1​)⋅P(w2​)⋅...⋅P(wn​)

**📌 Motivation Behind Unigram Tokenization**

* **BPE and WordPiece** give only a **single way** to segment a word.
* But in **real-world**, a word can be segmented in **multiple valid ways**.
* Unigram tokenization considers **multiple subword candidates** for each word.

**📌 General Steps in Unigram Tokenization**

**🔸 Step 1: Build a Large Initial Vocabulary**

* **Opposite** of BPE and WordPiece:
  + BPE/WP: Start small and **grow** vocab.
  + Unigram: Start big and **shrink** vocab.
* **Vocabulary Initialization Methods:**
  + Include all characters.
  + Add frequent substrings.
  + Can even run BPE exhaustively to generate candidate subwords.

**🔸 Step 2: Compute Unigram Probabilities**

* Calculate **frequency of each vocabulary token**.
* Convert frequency to **probability**:

P(t)=freq(t)∑freq(ti)P(t) = \frac{\text{freq}(t)}{\sum \text{freq}(t\_i)}P(t)=∑freq(ti​)freq(t)​

* Example:  
  For total frequency = 155,
  + If freq(R) = 3, → P(R) = 3/155
  + If freq(U) = 31 → P(U) = 31/155
  + And so on...

**🔸 Step 3: Find All Possible Splits for Each Word**

* Each word in the training data is split into **all valid segmentations** using current vocabulary.
* **Example – word "run":**
  + Split 1: R + U + N
  + Split 2: RU + N
  + Split 3: R + UN
* All splits must consist of tokens **present in current vocabulary**.

**🔸 Step 4: Compute Probabilities of Splits**

* Since tokens are assumed independent:

P(split)=P(t1)⋅P(t2)⋅...⋅P(tn)P(\text{split}) = P(t\_1) \cdot P(t\_2) \cdot ... \cdot P(t\_n)P(split)=P(t1​)⋅P(t2​)⋅...⋅P(tn​)

* Choose the **best split** with the **highest probability**.

**🔸 Step 5: Use Viterbi Algorithm (Optional)**

* **Viterbi Algorithm** is used to **efficiently compute** the best segmentation.
* It avoids brute-force computation of all segmentations.

**🔸 Step 6: Compute Log-Likelihood Loss**

* For each word:
  + Use **best split** from above.
  + Compute:

Loss=−log⁡P(best split)\text{Loss} = -\log P(\text{best split})Loss=−logP(best split)

* Multiply by word frequency in corpus.
* Total loss for the training corpus is the **sum of all word losses**.

**📌 EM Algorithm (Expectation Maximization)**

* Used to **optimize the vocabulary** by removing less useful tokens.
* EM = E-step + M-step (iterative optimization).

**🔸 E-step:**

* Estimate best splits and corresponding probabilities for all words.

**🔸 M-step:**

* Identify tokens in vocabulary that contribute **least** to reducing loss.
* **Remove** them from the vocabulary.

**📌 Token Removal Example**

* Suppose you remove the token un from vocabulary:
  + All segmentations using un become invalid.
  + Example:
    - For "run", a split like r + un is **no longer valid**.
    - Only other valid splits are considered.
  + Re-compute probabilities and **best splits**.
  + Re-compute loss.
  + If the overall log-likelihood **improves or doesn't degrade**, keep un removed.

**📌 Final Notes**

* The goal is to **reduce vocabulary size** while **minimizing log loss**.
* The vocabulary keeps shrinking until a **target size or convergence** is reached.
* Result: A compact vocabulary that gives **high-quality segmentations** considering multiple possibilities per word.