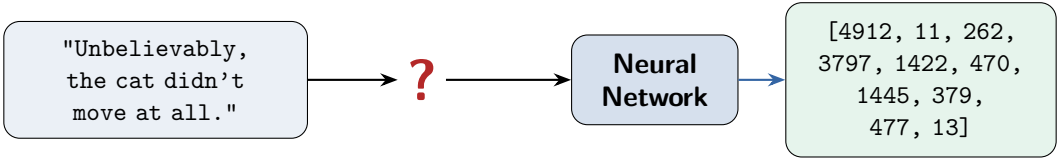
Tokenization

Word · Character · Subword · BPE · WordPiece · SentencePiece

Models need numbers, not text



Tokenization is the first step in any NLP pipeline:
split raw text into discrete units (tokens) and map each to an integer ID.
The choice of tokenizer affects **everything** downstream.

Three levels of granularity

Word-level

`["Unbelievably", "the", "cat", "didn't", "move"]`

Vocab size: ∼100k+

OOV problem

Subword

`["Un", "believ", "ably", ",", "the", "cat", "didn", "'t"]`

Vocab size: ∼30k–50k

The sweet spot

Character

`["U","n","b","e","l","i","e","v","a","b","l","y",...]`

Vocab size: ∼256

Very long sequences

Used by all modern LLMs

Word-level tokenization and the OOV problem

Vocabulary (fixed at training):

`the, cat, sat, on, mat,`
`dog, run, happy, sad, ...`

Size: 50,000–200,000 words

At test time:

`"The cryptocurrency market plummeted after the CEO's tweet."`

Words not in vocab → `[UNK]` [UNK] market [UNK] after the [UNK] tweet.

Problems: • New words, names, typos → `[UNK]` • Huge vocabulary → huge embedding matrix • Morphology lost: "run", "runs", "running" are unrelated tokens

Character-level: no unknowns, but...

Input: "The cat sat on the mat."

`The | cat | sat | on | the | mat | .`

7 tokens (word-level)

`T|h|e| |c|a|t| |s|a|t| |o|n| |t|h|e| |m|a|t|.`

23 tokens (character-level)

Drawbacks:

- Sequences $\sim$4–5$\times$ longer
- $O(n^2)$ attention cost explodes
- Characters carry little meaning
- Harder long-range dependen

Advantages:

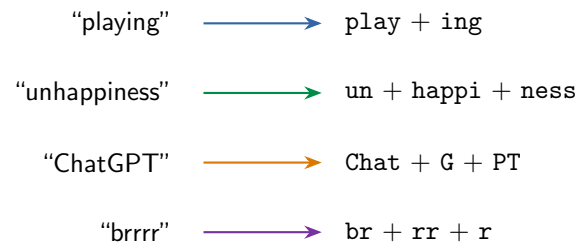
- Tiny vocabulary ($\sim$256)
- Zero unknown tokens
- Works for any language
- Handles typos, code, URLs

Too fine-grained on its own — but the idea of starting from characters inspires **subword** methods.

Subword tokenization: the key insight

Common words stay whole: `the`, `cat`, `and`

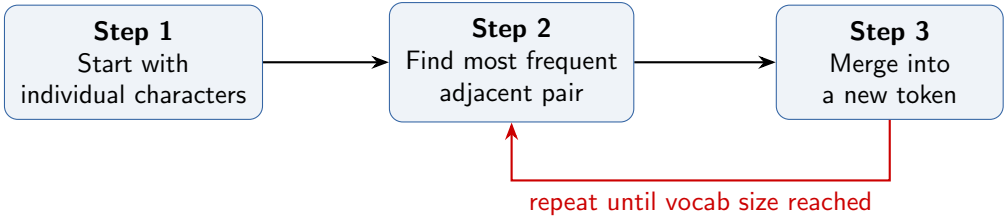
Rare words are split into known pieces: `un` + `believ` + `ably`



Vocab size ∼30k–50k • No `[UNK]` tokens • Reasonable sequence lengths • Morphology is partially captured

Byte-Pair Encoding (BPE): the idea

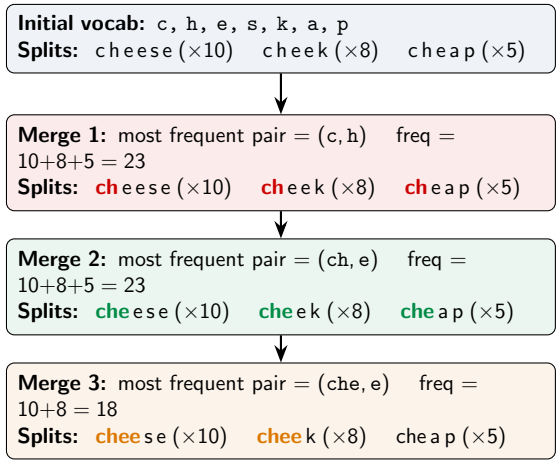
Training: learn merge rules from a corpus. **Inference:** apply merge rules to new text.



Originally a **data compression** algorithm (Gage, 1994).
Adopted for NLP by Sennrich et al. (2016). Used by **GPT**, **GPT-2**, **RoBERTa**, **BART**, **LLaMA**.

BPE: worked example

Training corpus (word × frequency): `cheese` ×10, `cheek` ×8, `cheap` ×5



Merge rules (ordered):

1. `c + h → ch`
2. `ch + e → che`
3. `che + e → chee`

⋮

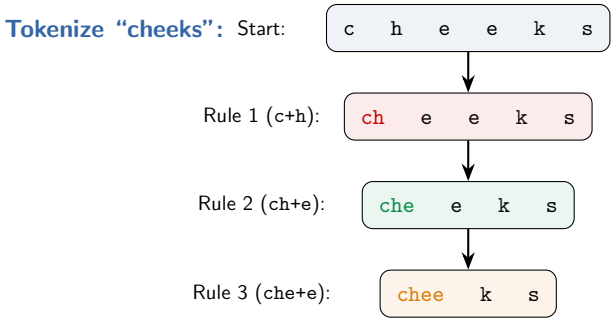
Vocab after 3 merges:

`c, h, e, s, k, a, p, ch, che, chee`

Notice: "cheap" stops merging at step 3 — it has no "ee". Common prefixes get merged first.

BPE: tokenizing new text

Given the learned merge rules, tokenize a word **never seen** in training:



Key point:

Apply merge rules in the *same order* they were learned.

"cheeks" was never in the training corpus, but BPE splits it into known pieces: `chee` + `k` + `s`

Result: `["chee", "k", "s"]` → token IDs `[9, 4, 3]`

Byte-level BPE (GPT-2, GPT-3, LLaMA)

Standard BPE

Base vocab = Unicode characters

Vocab size = ∼30k–50k

Unknown chars → `[UNK]`

Problem: Chinese, emoji, etc. can hit unknown characters

upgrade

Byte-level BPE

Base vocab = **256 byte values**

Vocab size = 256 + merges (GPT-2: 50,257 total)

No `[UNK]` ever

Any byte sequence is representable

Every text is ultimately a sequence of bytes (UTF-8 encoding).
By starting from **bytes** instead of characters, BPE can tokenize *any* input: English, Chinese, Arabic, code, emoji, binary data — all with the same vocabulary.

WordPiece (BERT, DistilBERT)

BPE

Merge criterion:
most frequent pair

Greedy count-based

Used by: GPT, LLaMA, etc.

WordPiece

Merge criterion:
pair that **maximizes likelihood** of the training corpus

Used by: BERT, DistilBERT

$$\text{score}(a, b) = \frac{\text{freq}(ab)}{\text{freq}(a) \times \text{freq}(b)}$$

Notation: WordPiece marks *continuation* subwords with `##`

"unbelievably" → `["un", "##believ", "##ably"]`

BPE instead marks *word-initial* subwords (e.g., GPT-2 uses `Ġ` = space prefix)

WordPiece: BPE vs WordPiece merge decision

Corpus: cheese ($\times 10$), cheek ($\times 8$), cheap ($\times 5$), seep ($\times 7$) Current state: all split to characters

Which pair to merge: (e, e) or (e, p) ?

BPE: count frequency

(e, e): cheese($\times 10$) + cheek($\times 8$) = **18**

(e, p): cheap($\times 5$) + seep($\times 7$) = 12

Winner: (e, e) — higher count

WordPiece: likelihood score

$$\frac{\text{freq(ee)}}{\text{freq(e)} \times \text{freq(e)}} = \frac{18}{43 \times 43} = 0.0097$$

$$\frac{\text{freq(ep)}}{\text{freq(e)} \times \text{freq(p)}} = \frac{12}{43 \times 12} = \mathbf{0.0233}$$

Winner: (e, p) — higher score

WordPiece prefers merging pairs whose co-occurrence is **surprising** relative to individual frequencies.

"e" is very common, so "ee" is not surprising.
But "ep" appearing together is more informative.

freq(e) = 43 (summing all e's in the corpus), freq(p) = 12 (from "cheap" and "seep")

Unigram LM and SentencePiece

Unigram LM

Start with a *large* vocab

Iteratively **remove** tokens that hurt likelihood the least

Top-down (vs BPE's bottom-up)

Used by: T5, ALBERT, XLNet

Kudo, 2018

SentencePiece

Not an algorithm — a **library**

Treats input as raw byte stream (no pre-tokenization needed)

Supports both **BPE** and **Unigram**

Language-agnostic: no need for space-based word splitting

Kudo & Richardson, 2018

BPE = bottom-up (merge frequent pairs) vs
Unigram = top-down (prune unlikely tokens)

Both converge to similar subword vocabularies in practice.

Why tokenization matters: LLM quirks

Bad at arithmetic

"12345" → `["123", "45"]`
The model never sees the individual digits together!

Poor non-English efficiency

English "hello" = 1 token
Korean "annyeong" = 3–5 tokens
Same meaning, 3–5× the cost!

Sensitive to formatting

"`Hello World`" and
"`Hello  World`" produce different token sequences.

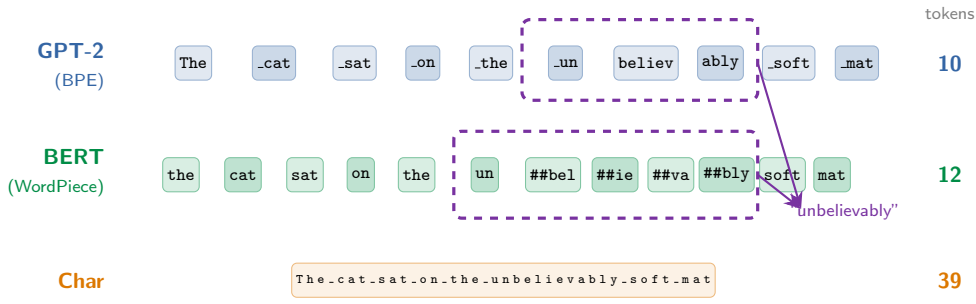
Can't count letters

"How many r's in strawberry?"
"`straw`" + "`berry`" — the model can't see individual letters.

Many apparent "reasoning failures" of LLMs are actually **tokenization artifacts**.
The model literally cannot see what you think it sees.

Visualizing: the same sentence, different tokenizers

Input: "The cat sat on the unbelievably soft mat"



Same input, very different representations. "unbelievably" = **3 tokens** (GPT-2), **5 tokens** (BERT), **12 characters**.
Fewer tokens = cheaper inference (less compute per sequence).

Special tokens

[CLS]
Classification token (BERT)

[SEP]
Separator between segments

[PAD]
Padding to equal length

[UNK]
Unknown token (fallback)

⟨BOS⟩
Beginning of sequence

⟨EOS⟩
End of sequence

[MASK]
Masked position (BERT MLM)

Special tokens are **not** in the original text — they're added by the tokenizer to give the model structural signals: where sequences begin/end, what to predict, etc.

Comparison of tokenization methods

Method	Direction	Criterion	Vocab size	Used by
BPE	Bottom-up	Frequency	30k–50k	GPT, LLaMA
WordPiece	Bottom-up	Likelihood	30k	BERT
Unigram	Top-down	Likelihood	30k–50k	T5, XLNet
Byte BPE	Bottom-up	Frequency	50k–100k	GPT-2/3/4
Character	—	—	256	ByT5

In practice, the differences between BPE, WordPiece, and Unigram are **small**. What matters most: vocab size, training corpus, and whether byte-level is used.

Byte-level BPE is the current default for new large language models.

Practical: tokenizers in action

```python
# GPT-4 tokenizer
import tiktoken
enc = tiktoken.encoding_for_model(
  "gpt-4")
tokens = enc.encode(
  "Hello world!")
# [9906, 1917, 0]
enc.decode(tokens)
# "Hello world!"
```

```python
# BERT tokenizer
from transformers import
  AutoTokenizer
tok = AutoTokenizer.from_pretrained(
  "bert-base-uncased")
tok.tokenize(
  "unbelievably")
# ["un", "##bel", "##ie",
#  "##va", "##bly"]
```

Try it yourself: `https://platform.openai.com/tokenizer`

Paste any text and see how GPT tokenizes it. Pay attention to: numbers, non-English text, code, and whitespace.

Further reading

Subword Tokenization

- Sennrich et al. (2016), "Neural Machine Translation of Rare Words with Subword Units" (BPE)
- Kudo & Richardson (2018), "SentencePiece: A simple and language independent subword tokenizer"
- Kudo (2018), "Subword Regularization: Improving Neural Network Translation Mod-

WordPiece & Byte-Level

- Schuster & Nakajima (2012), "Japanese and Korean Voice Search" (WordPiece)
- Wang et al. (2020), "Neural Machine Translation with Byte-Level Subwords"

Practical Resources

- HuggingFace Tokenizers library documentation — `huggingface.co/docs/tokenizers`
- OpenAI Tokenizer tool — `platform.openai.com/tokenizer`

Questions?

Next: Evaluation — Perplexity, BLEU, ROUGE