

Tokenisers

Aayush Bajaj

July 10, 2026

Contents

1	why subword	1
2	byte-pair encoding	2
2.1	a worked example, computed by code	2
3	wordpiece and unigram-lm	4
4	byte-level bpe, sentencepiece, special tokens	4
5	fertility and compression	5
6	see also	5

before a transformer sees a single number, text must be cut into pieces and each piece mapped to an integer. the tokeniser is that cut — the least glamorous and most consequential preprocessing step in the whole pipeline, since it fixes the vocabulary, the sequence length, and what the model *can* represent at all.

*most ‘the model can’t spell /
can’t do arithmetic’ failures
trace back to the tokeniser, not
the weights*

1 why subword

two extremes, both bad:

- **word-level.** one id per word. vocabulary explodes (english has millions of surface forms once you count inflections and typos), and any word unseen at training becomes a single `<unk>` token — total information loss. morphology is invisible: `run`, `running`, `ran` are three unrelated atoms.
- **character-level.** one id per character. tiny vocabulary, zero out-of-vocabulary (oov) problem, but sequences become brutally long and the model must relearn that `t-h-e` is a unit every time — burning attention budget ($O(n^2)$ in **trans-formers**) on trivial composition.

subword tokenisation is the compromise: frequent words stay whole, rare words fracture into meaningful pieces. it delivers three things at once:

- **bounded vocabulary** — pick a target size (30k–100k typical) and the algorithm hits it.
- **no oov** — worst case, a word decomposes to bytes or characters, so *everything* is representable.
- **morphology for free** — `unhappiness` naturally splits `un + happi + ness`,

sharing the `un` and `ness` pieces with thousands of other words.

2 byte-pair encoding

bpe (sennrich et al. 2016, *neural machine translation of rare words with subword units*) borrows a 1994 data-compression algorithm: repeatedly replace the most frequent adjacent pair of symbols with a new merged symbol. run on text, the merges *are* the vocabulary.

the training algorithm:

- **input:** a corpus of words with frequencies; represent each word as a sequence of characters plus an end-of-word marker `</w>` (so merges cannot cross word boundaries and the marker records where words end).
- **count:** tally every adjacent symbol pair across the corpus, weighted by word frequency.
- **merge:** take the most frequent pair (a, b) , add the new symbol ab to the vocabulary, and replace every occurrence in the corpus.
- **loop:** repeat count-and-merge until the vocabulary reaches the target size (or a fixed number of merges).
- **output:** the ordered list of merges — this *is* the tokeniser, applied to new text by replaying the merges in order.

2.1 a worked example, computed by code

corpus `low` $\times 5$, `lower` $\times 2$, `newest` $\times 6$, `widest` $\times 3$. the first pass counts pairs: `e s` appears in `newest` and `widest` for $6 + 3 = 9$, the corpus maximum, so it merges first. the code below runs eight merges and then applies the learned rules to unseen words.

```
1 from collections import Counter
2
3 # toy corpus with word frequencies (sennrich et al.'s classic example,
4   ↪ extended)
5 corpus = {"low": 5, "lower": 2, "newest": 6, "widest": 3}
6
7 # represent each word as a tuple of symbols, with an end-of-word marker
8 words = {tuple(w) + ("</w>"), f for w, f in corpus.items()}
9
10 def pair_counts(words):
11     counts = Counter()
12     for sym, f in words.items():
13         for a, b in zip(sym, sym[1:]):
14             counts[(a, b)] += f
15     return counts
16
17 def merge(words, pair):
18     a, b = pair
19     out = {}
20     for sym, f in words.items():
21         new, i = [], 0
22         while i < len(sym):
23             if i < len(sym) - 1 and sym[i] == a and sym[i+1] == b:
24                 new.append(a + b); i += 2
25             else:
26                 new.append(sym[i]); i += 1
```

```

26     out[tuple(new)] = f
27     return out
28
29 merges = []
30 for step in range(1, 9):
31     counts = pair_counts(words)
32     best, freq = counts.most_common(1)[0]
33     words = merge(words, best)
34     merges.append(best)
35     print(f"merge {step}: {best[0]!r} + {best[1]!r} -> {best[0]+best[1]!r}
36         ↪ (count {freq})")
37
38 print("\nfinal segmentations:")
39 for sym, f in words.items():
40     print(f" {sym} {f}")
41
42 def encode(word, merges):
43     """apply learned merges, in order, to a new word."""
44     sym = list(word) + ["</w>"]
45     for a, b in merges:
46         i = 0
47         while i < len(sym) - 1:
48             if sym[i] == a and sym[i+1] == b:
49                 sym[i:i+2] = [a + b]
50             else:
51                 i += 1
52     return sym
53
54 print("\nunseen words:")
55 for w in ("lowest", "newer", "widow"):
56     print(f" {w} -> {encode(w, merges)}")

```

```

merge 1: 'e' + 's' -> 'es' (count 9)
merge 2: 'es' + 't' -> 'est' (count 9)
merge 3: 'est' + '</w>' -> 'est</w>' (count 9)
merge 4: 'l' + 'o' -> 'lo' (count 7)
merge 5: 'lo' + 'w' -> 'low' (count 7)
merge 6: 'n' + 'e' -> 'ne' (count 6)
merge 7: 'ne' + 'w' -> 'new' (count 6)
merge 8: 'new' + 'est</w>' -> 'newest</w>' (count 6)

```

final segmentations:

```

low </w>          x5
low e r </w>      x2
newest</w>       x6
w i d est</w>    x3

```

unseen words:

```

lowest -> ['low', 'est</w>']
newer  -> ['new', 'e', 'r', '</w>']
widow  -> ['w', 'i', 'd', 'o', 'w', '</w>']

```

the payoff is the last block: **lowest** was never in the corpus, yet it tokenises cleanly as **low** + **est</w>** — the two productive morphemes bpe discovered — even though neither **lowest** nor **est** appeared as a whole word. that is generalisation from a compression objective.

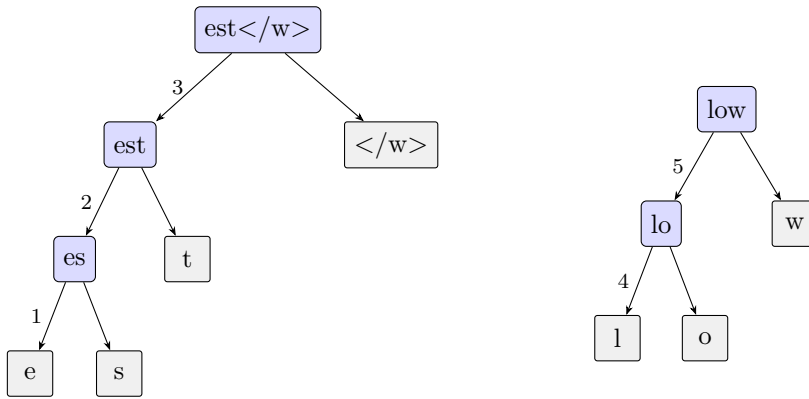


Figure 1: merge tree for the two learned stems. each internal node is a bpe merge, annotated with its order; leaves are the initial characters. reading the frontier at any merge count gives the vocabulary at that stage.

3 wordpiece and unigram-lm

bpe merges the most *frequent* pair. two alternatives change the criterion:

- **wordpiece** (schuster & nakajima 2012; the bert tokeniser) merges the pair that most increases the *likelihood* of the corpus under a unigram language model. concretely it picks the pair maximising $\frac{\text{count}(ab)}{\text{count}(a) \text{count}(b)}$ — a pointwise-mutual-information-flavoured score rather than raw frequency, so it prefers pairs that co-occur more than chance would predict, not merely pairs that are common. it marks continuation pieces with **##** (playing to play **##ing**).
- **unigram lm** (kudo 2018, *subword regularization*) inverts the whole procedure. instead of building up from characters, it starts with a large superset of candidate subwords and *prunes*, keeping the vocabulary that best explains the corpus under a unigram model $p(\text{piece})$. tokenising a word is then a search for the highest-probability segmentation

$$\arg \max_{s_1 \dots s_k = w} \sum_{i=1}^k \log p(s_i), \quad (1)$$

solved exactly by **viterbi** over the lattice of possible splits — the same dynamic program used in hmm decoding, with subword pieces as states.

the practical difference is modest; unigram-lm tends to produce slightly more linguistically clean pieces, bpe is simpler and deterministic. both beat word- and character-level decisively.

*because it scores every segmentation, unigram-lm can also *sample* alternatives at training time — subword regularisation, a cheap augmentation*

4 byte-level bpe, sentencepiece, special tokens

- **byte-level bpe** (gpt-2, radford et al. 2019 (Radford, Alec and Wu, Jeffrey and Child, Rewon and Luan, David and Amodei, Dario and Sutskever, Ilya, 2019)) runs bpe over raw utf-8 **bytes** instead of unicode characters. the base alphabet is exactly 256 symbols, so there is *provably* no oov — any string, any language, any emoji, any binary garbage encodes. no **<unk>** token exists at all. the cost is that non-latin scripts spend several bytes per character and thus tokenise less efficiently.

- **sentencepiece** (kudo & richardson 2018, *sentencepiece*) is the tooling that made subword tokenisers language-agnostic: it treats the input as a raw stream (spaces included, encoded as `_`), so it needs no pretokenisation and round-trips losslessly — essential for languages like japanese and chinese that do not delimit words with spaces. it can train either bpe or unigram-lm underneath.
- **special tokens** are reserved ids the model treats structurally, not as text: `<bos>` / `<eos>` (sequence boundaries), `<pad>` (batch padding), `<unk>` (fallback, absent in byte-level schemes), and chat/instruct control tokens like `<|im_start|>`. they are added to the vocabulary after training so no ordinary text can collide with them.

5 fertility and compression

how good is a tokeniser? two intrinsic metrics:

- **fertility** — average tokens per word. lower is better (fewer, larger pieces): a fertility of 1.1 means most words are one token. english on a well-matched vocabulary sits near 1.3–1.5; the same tokeniser on an under-represented language can hit 3–5, which directly inflates cost and shrinks effective context.
- **compression** — bytes per token, or characters per token. higher means each token carries more text, so a fixed context window holds more content. these two are the reason multilingual models with skewed training data are quietly more expensive to use in some languages than others — a fairness issue baked into the vocabulary.

6 see also

- *transformers* — what consumes the integer sequence a tokeniser emits
- *llms* — where vocabulary size trades against embedding-table cost
- *nanogpt* — a character-level tokeniser, the simplest possible baseline

references

Radford, Alec and Wu, Jeffrey and Child, Rewon and Luan, David and Amodei, Dario and Sutskever, Ilya (2019). *Language Models are Unsupervised Multitask Learners*.