

1 Tokenization

A language model never sees letters. It sees *lists of whole numbers*. **Tokenization** is the step that turns text into those numbers, and **detokenization** turns them back into text. Almost every strange behavior of chatbots with spelling, counting letters, or non-English text traces back to this step.

2 The big picture: why do we need tokens?

A neural network is a giant pile of arithmetic: multiply, add, repeat. It can only work with numbers. But we want to feed it sentences. So we need a *translation layer* between human text and numbers.

Figure 1 shows the whole journey of one piece of text through a language model. The tokenizer sits at both ends: it converts the text you type into numbers going in, and converts the model's chosen numbers back into text coming out.

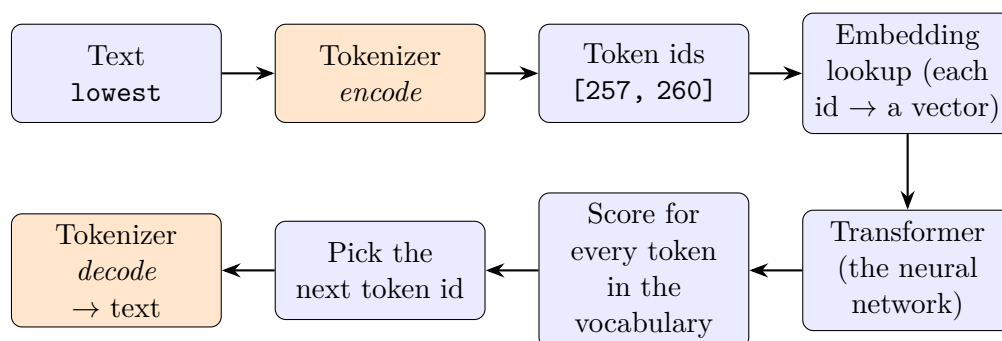


Figure 1: The life of a piece of text inside a language model. The orange boxes are the tokenizer; everything else is the neural network. A chatbot repeats the bottom row many times, one token at a time.

A small glossary

Term	Meaning
Token	One chunk of text that the model treats as a single unit (a letter, a piece of a word, a whole word, a space plus a word, ...).
Token id	The whole number that names a token, such as 257.
Vocabulary	The complete list of tokens the model knows, with their ids.
Encode	Text → list of token ids.
Decode	List of token ids → text.
Corpus	A big collection of example text used to <i>train</i> the tokenizer.
Merge rule	A BPE rule such as “whenever 1 is followed by o, replace them with the single token lo.”

3 What exactly is a token?

A token is *not* the same as a word. It is whatever chunk the tokenizer decided to use. For English text, a rough rule of thumb used by many systems is that one token is about four characters, or about three quarters of a word. So a 1000-word essay is roughly 1300 tokens. Here is the same sentence cut into tokens three different ways. (The subword split is illustrative; a real tokenizer might cut it a little differently.)

Characters (20)	T	o	k	e	n	i	z	a	t	i	o	n		i	s		f	u	n	!
Subwords (5)	Token				ization				is		fun		!							
Words (4)	Tokenization										is		fun		!					

Figure 2: “Tokenization is fun!” at three levels of granularity. The symbol stands for a space character. Notice that the subword tokens often carry the space *with* the word that follows it.

Checkpoint

Why is the second subword token written `is` instead of just `is`? (Answer: spaces are characters too. Many tokenizers attach the space to the front of the following word, so `is` and `is` are *different* tokens with different ids.)

4 Why tokenize at all? The reasons

1. **Neural networks need numbers.** Text must become numbers before any arithmetic can happen. Tokenization is the agreed-upon dictionary for that conversion.
2. **The model predicts from a fixed menu.** A language model’s output is a score for *every* token in its vocabulary (“which token comes next?”). That needs a fixed, finite list. You cannot have a menu with infinitely many possible words.
3. **Shorter sequences are cheaper.** A transformer compares every token with every other token, so the work grows with the *square* of the sequence length (Section 9). Packing text into fewer, bigger tokens makes models faster and cheaper to run.
4. **Longer effective memory.** A model can only look at a limited number of tokens at once (its *context window*). If each token carries more text, the same window holds more of a document.
5. **Tokens can carry meaning.** Chunks such as `un`, `ing`, or `tion` (or a common word like `the`) are useful building blocks. It is easier to learn from them than from isolated letters.

6. **Rare and brand-new words still work.** A good tokenizer can always fall back on smaller pieces, so a word it has never seen before (a new slang word, a typo, a person’s name) can still be represented.
7. **One scheme for every language and symbol.** English, Spanish, Chinese, emoji, and computer code can all be handled by the same method.

The art of tokenization is a balancing act between two opposite goals: *small vocabulary* (easy for the model to learn and store) versus *short sequences* (fast and able to hold lots of text). Different strategies sit at different points on this seesaw.

5 Three simple strategies

5.1 Character-level tokens

Each letter, digit, punctuation mark, and space is its own token. The vocabulary is tiny (maybe 100 to a few hundred symbols) and you never meet an “unknown” symbol. The price: sequences are long (“hello world” is 11 tokens), and the model must learn from scratch that the letters h-e-l-l-o together mean something.

5.2 Word-level tokens

Split on spaces and punctuation, and give every distinct word an id. Sequences are short, and each token has a clear meaning. But the problems are serious:

- **Huge vocabulary.** English has hundreds of thousands of words, and then *run*, *runs*, *running*, *runner* would all be separate entries.
- **Unknown words.** A word not in the list (a typo like “helllo”, a new word, a name) has no id. Systems used a special <UNK> token and simply lost the information.
- **Not all languages use spaces.** Chinese and Japanese are normally written without spaces between words, so “split on spaces” does not work.

5.3 Subword tokens (the winner)

Cut text into pieces that are *between* letters and words in size. Common words stay whole (**the**, **house**). Rare words are assembled from common pieces (**un** + **happi** + **ness**). The vocabulary is a manageable size (tens of thousands to a couple hundred thousand), sequences are reasonably short, and any word can be built from parts. **Byte-Pair Encoding** is the most popular way to *find* good subword pieces.

	Characters	Words	Subwords (BPE)
Typical vocabulary size	small (100–300)	huge (100,000+)	medium (30,000–200,000)
Sequence length	very long	short	medium
Unknown words?	never	often	essentially never
Works without spaces?	yes	no	yes
Tokens carry meaning?	weakly	strongly	usually

6 Text inside a computer: Unicode and UTF-8

Before we can do BPE, we need to know what a computer actually stores when it holds text.

Step 1: every character has a number. **Unicode** is a giant international table that gives each character a number called its *code point*. For example the letter **A** is 65, the accented letter **é** is 233, the euro sign is 8364, and a smiley-face emoji is 128512. Unicode covers well over 100,000 characters from nearly every writing system.

Step 2: numbers are stored as bytes. A **byte** is a whole number from 0 to 255 (eight on/off switches, or bits, give $2^8 = 256$ possibilities). The most common way to store Unicode in bytes is **UTF-8**. Its trick is to use *fewer bytes for common characters* and more bytes for rarer ones:

Character	Code point	UTF-8 bytes (decimal)	# bytes
A	65	65	1
é (e with accent)	233	195, 169	2
Euro sign	8364	226, 130, 172	3
Smiley-face emoji	128512	240, 159, 152, 128	4

Plain English letters, digits, and punctuation (the old ASCII set) take exactly one byte each. Characters from other alphabets and emoji take two to four.

You can see this yourself in Python:

```
>>> list("Hi!".encode("utf-8"))
[72, 105, 33]
>>> bytes([72, 105, 33]).decode("utf-8")
'Hi!'
```

Any text in any language is just a list of numbers from 0 to 255. That gives us a perfect starting vocabulary of exactly 256 symbols. Starting from bytes means there is *no such thing as an unknown character*: the tokenizer can always fall back to bytes. This is called **byte-level** tokenization and is the approach used by GPT-2 and many later models.

7 Byte-Pair Encoding (BPE)

7.1 The idea

BPE began in 1994 as a simple data-compression trick, and was adapted for neural language models around 2016. Its core idea is one you already use when texting:

If a pair of things appears together *all the time*, give the pair its own short name.

(If you type “see you later” constantly, you invent “cul8r.”) BPE does this automatically, over and over: look for the most common pair of neighboring tokens, replace it with a new single token, and repeat. Common words end up as one token; rare words end up as several pieces.

7.2 Training: learning the merge rules

Training the tokenizer means reading a large sample of text (the corpus) and learning an ordered list of merge rules. It happens once, before the language model itself is trained.

1. **Start.** Convert the corpus to bytes. The vocabulary begins as the 256 byte tokens, with ids 0 to 255.
2. **Count** how often each pair of neighboring tokens appears.
3. **Pick** the most frequent pair. (If there is a tie, take the one found first.)
4. **Create** a new token for that pair, with the next unused id (256, then 257, ...). Write down the rule.
5. **Replace** every occurrence of the pair in the data with the new token.
6. **Repeat** steps 2–5 until you have made as many merges as you wanted.

The *result* of training is (a) the ordered list of merge rules and (b) the vocabulary table that says which bytes each token id stands for. The number of merges is a design choice: vocabulary size = 256 + number of merges (plus a few special tokens, see Section 11). For example GPT-2 has 256 byte tokens + 50,000 merges + 1 special token = 50,257 tokens.

The ordered merge list *is* the tokenizer. Give two people the same list and the same text, and they will always get exactly the same tokens.

7.3 A complete worked example

Our tiny training corpus (62 bytes, including spaces) is:

```
low low low low lower lower newest newest newest widest widest
```

So the data starts as 62 byte-tokens. Let’s do the first few steps by hand.

Merge 1. Count neighboring pairs. The top counts are: `l+o` appears 6 times (in *low* four times and *lower* twice), `o+w` also 6 times, and then several pairs at 5 (such as `␣+l` and `e+s`). There is a tie at 6; we take the first one found: `l+o`. Give it the new id 256, standing for the bytes `lo`. Every `lo` in the data is replaced, so we now have 56 tokens.

Merge 2. Recount. Now `lo+w` is the winner (6 times). It becomes id 257, meaning `low`. We are down to 50 tokens. Notice something important: rule 257 is *built on top of* rule 256. Later merges can combine earlier merges into ever-longer chunks.

Merge 3. Now the best pair is a space followed by `low` (5 times). It becomes id 258, the token `␣low` (with its leading space). 45 tokens remain.

Continuing the same way for ten merges gives the table below. Every number in it was produced by running the Python program in Section 8.

#	Merge (left + right → new)	New id	Seen	Tokens left
0	(start: raw bytes)	—	—	62
1	<code>l + o → lo</code>	256	6	56
2	<code>lo + w → low</code>	257	6	50
3	<code>␣+ low → ␣low</code>	258	5	45
4	<code>e + s → es</code>	259	5	40
5	<code>es + t → est</code>	260	5	35
6	<code>est + ␣→ est␣</code>	261	4	31
7	<code>␣low + ␣low → ␣low␣low</code>	262	3	29
8	<code>n + e → ne</code>	263	3	26
9	<code>ne + w → new</code>	264	3	23
10	<code>new + est␣→ newest␣</code>	265	3	20

The corpus shrank from 62 tokens to 20, so on this (very repetitive!) text each token stands for about $62/20 = 3.1$ bytes on average. Real English with a big vocabulary averages about 4 characters per token.

Figure 3 shows how the final token `newest␣` was assembled from raw bytes by repeated merging. Each node is labeled with its token id.

7.4 Encoding: turning a string into tokens

Now suppose we are given the finished merge list and a *new* string. We do **not** count frequencies in the new text. Instead we replay the rules, **always using the earliest-learned rule that still applies**:

1. Convert the string to UTF-8 bytes (a list of numbers 0–255).
2. Look at all neighboring pairs. Among those that appear in the merge list, find the one whose rule was learned *earliest*.

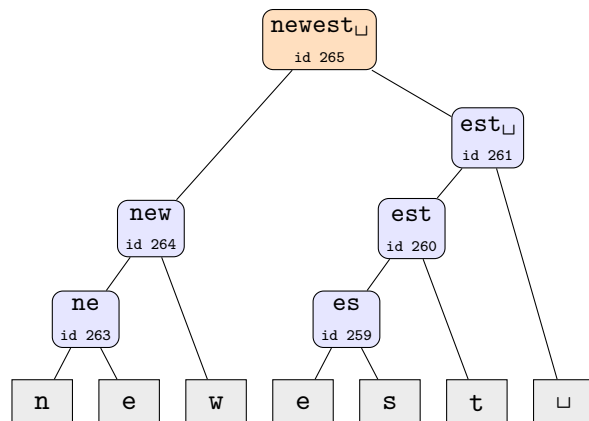


Figure 3: How BPE builds the token `newest_` out of single bytes. Merge number k has id $255 + k$, so id 259 was learned fourth and id 265 last.

3. Merge every occurrence of that pair into its new token id.
4. Repeat steps 2–3 until none of the neighboring pairs is in the merge list.

Example: encode `lowest`. The bytes are `l o w e s t`, which are the numbers [108, 111, 119, 101, 115, 116].

Step	Tokens so far	What happened
start	[108, 111, 119, 101, 115, 116]	Raw bytes: <code>l o w e s t</code> .
1	[256, 119, 101, 115, 116]	Rule 256 (<code>l+o</code>) is the earliest rule that applies.
2	[257, 101, 115, 116]	Rule 257 (<code>lo+w</code>).
3	[257, 259, 116]	Rule 259 (<code>e+s</code>).
4	[257, 260]	Rule 260 (<code>es+t</code>).
done	[257, 260]	No more rules apply.

So `lowest` $\rightarrow$ [257, 260], which are the two tokens `low` and `est`. Notice that *lowest* never appeared in our corpus! The tokenizer built it from two pieces it learned from *low* and *newest*. That is exactly how BPE handles new words.

Why must we use the earliest rule first? Because the merge rules are *ordered by priority*, and later rules were built assuming earlier ones had already happened. Applying them in training order guarantees that the encoder reproduces what training would have done. Here is a subtle consequence. Encoding `newest low` gives [264, 260, 258], which is the three tokens `new`, `est`, `_low`. The space after *newest* got absorbed into `_low` (rule 258) *before* rule 261 (`est+_`) had a chance to use it, because 258 was learned earlier than 261.

What about text we never trained on? Our toy corpus has no accented letters or emoji. Encoding `café` plus a smiley emoji simply gives the raw UTF-8 bytes [99, 97,

102, 195, 169, 32, 240, 159, 152, 128]: ten tokens, one per byte. It is inefficient but *it never fails*. A tokenizer trained on text that contained lots of such characters would have learned merges for them and used fewer tokens.

7.5 Decoding: turning tokens back into a string

Decoding is the easy direction:

1. Look up each token id in the vocabulary table to get its bytes.
2. Glue all the bytes together, in order.
3. Interpret the resulting bytes as UTF-8 text.

Example. Decode [257, 260]. Token 257 is the bytes `low`, token 260 is the bytes `est`. Gluing them gives `lowest`. Done.

A multi-byte example. Decode [99, 97, 102, 195, 169]. Each id below 256 is just its own single byte. The glued bytes are `c a f` then the two bytes 195 and 169, which together are the UTF-8 code for `é`. Result: `café`.

Tokens do not always line up with whole characters! A single emoji may be split across several tokens (one per byte). If you decode only *part* of such a sequence, e.g. [99, 97, 102, 195], the final byte 195 is half of a character and is not valid on its own. Python's `errors="replace"` setting shows the special replacement character instead of crashing. Chat applications that stream a model's answer token by token must deal with this by waiting until the bytes form a complete character.

Encoding must replay the merges in order; decoding is just a table lookup. For byte-level BPE, $\text{decode}(\text{encode}(s)) = s$ for *every* string s : the process is perfectly reversible and loses nothing.

8 The Python code

Below is a complete, working byte-level BPE tokenizer in plain Python. It uses no outside libraries, only the `Counter` helper that ships with Python.

8.1 Part 0: text to bytes, bytes to text

```
"""A tiny byte-level BPE tokenizer, written to be read by students."""
from collections import Counter

# -----
# Step 0: text <-> bytes
```

```

# -----
def text_to_bytes(text):
    """Turn a string into a list of integers, each between 0 and 255."""
    return list(text.encode("utf-8"))

def bytes_to_text(byte_values):
    """Turn a list of integers (0-255) back into a string."""
    return bytes(byte_values).decode("utf-8", errors="replace")

```

8.2 Part 1: the two helper operations

`count_pairs` counts how often each neighboring pair appears. `merge_pair` walks along the list, and whenever it sees the chosen pair it writes the new id instead and skips ahead *two* places.

```

# -----
# Step 1: the two helper operations BPE is built from
# -----
def count_pairs(ids):
    """Count how often each neighbouring pair of tokens occurs."""
    counts = Counter()
    for left, right in zip(ids, ids[1:]):
        counts[(left, right)] += 1
    return counts

def merge_pair(ids, pair, new_id):
    """Replace every occurrence of `pair` in `ids` by the single token `
        new_id`."""
    result = []
    i = 0
    while i < len(ids):
        if i + 1 < len(ids) and (ids[i], ids[i + 1]) == pair:
            result.append(new_id)
            i += 2          # we used up two tokens
        else:
            result.append(ids[i])
            i += 1
    return result

```

8.3 Part 2: training

This is the loop from Section 7.3. It returns `merges` (the ordered rules, as a dictionary from a pair of ids to the new id) and `vocab` (a dictionary from each id to the bytes it stands for).

```

# -----
# Step 2: training = learning the merge rules from example text
# -----
def train_bpe(text, num_merges, verbose=False):

```

```

"""Learn up to `num_merges` merge rules.

Returns
    merges: dict (left_id, right_id) -> new_id (in the order learned)
    vocab : dict token_id -> the bytes that token stands for
    """
ids = text_to_bytes(text)
vocab = {i: bytes([i]) for i in range(256)} # the 256 starting
      tokens
merges = {}

for step in range(num_merges):
    counts = count_pairs(ids)
    if not counts:
        break
    best = max(counts, key=counts.get) # most frequent pair
    if counts[best] < 2: # nothing worth merging
        break
    new_id = 256 + step
    ids = merge_pair(ids, best, new_id)
    merges[best] = new_id
    vocab[new_id] = vocab[best[0]] + vocab[best[1]]
    if verbose:
        print(f"merge {step + 1}: {vocab[best[0]]} + {vocab[best[1]]}
              "
              f"-> {vocab[new_id]} (id {new_id}), seen {counts[best]}x
              ",
              f"{len(ids)} tokens left")
return merges, vocab

```

8.4 Part 3: encoding (string $\rightarrow$ tokens)

This follows Section 7.4. The key line uses min with the rule's id as the "score": a smaller id means a rule learned earlier, so it has priority. Pairs with no rule get a score of infinity, so they lose. If even the best pair has no rule, we are done.

```

# -----
# Step 3: encoding = text -> token ids
# -----
def encode(text, merges, verbose=False):
    """Convert a string to token ids by replaying the learned merges."""
    ids = text_to_bytes(text)
    if verbose:
        print("start:", ids)
    while len(ids) >= 2:
        counts = count_pairs(ids)
        # Among the pairs present, pick the one learned EARLIEST in
        # training.
        pair = min(counts, key=lambda p: merges.get(p, float("inf")))
        if pair not in merges:
            break # no more merges apply
        ids = merge_pair(ids, pair, merges[pair])

```

```

        if verbose:
            print(f"apply {pair} -> {merges[pair]}:", ids)
    return ids

```

8.5 Part 4: decoding (tokens → string)

Three short lines, matching the three steps of Section 7.5.

```

# -----
# Step 4: decoding = token ids -> text
# -----
def decode(ids, vocab):
    """Convert token ids back to a string."""
    raw = b"".join(vocab[i] for i in ids)      # glue the bytes
                                              # together
    return raw.decode("utf-8", errors="replace") # bytes -> text

```

8.6 Part 5: a demonstration

```

# -----
# Demo
# -----
if __name__ == "__main__":
    corpus = "low low low low lower lower newest newest newest widest
              widest"
    merges, vocab = train_bpe(corpus, num_merges=10, verbose=True)

    print()
    word = "lowest"
    ids = encode(word, merges, verbose=True)
    print("token ids      :", ids)
    print("token pieces:", [vocab[i] for i in ids])
    print("decoded       :", decode(ids, vocab))
    print("round trip ok:", decode(ids, vocab) == word)

    print()
    for s in ["newer", "low low", "caf\u00e9 \U0001F600", "zzz", ""]:
        ids = encode(s, merges)
        # ascii() shows non-English characters as escape codes
        print(ascii(s), "->", ids, "->", ascii(decode(ids, vocab)),
              decode(ids, vocab) == s)

    print()
    print("bytes in corpus :", len(text_to_bytes(corpus)))
    print("tokens in corpus:", len(encode(corpus, merges)))

```

8.7 What the program prints

```

merge 1: b'l' + b'o' -> b'lo' (id 256), seen 6x, 56 tokens left
merge 2: b'lo' + b'w' -> b'low' (id 257), seen 6x, 50 tokens left
merge 3: b' ' + b'low' -> b' low' (id 258), seen 5x, 45 tokens left
merge 4: b'e' + b's' -> b'es' (id 259), seen 5x, 40 tokens left
merge 5: b'es' + b't' -> b'est' (id 260), seen 5x, 35 tokens left
merge 6: b'est' + b' ' -> b'est ' (id 261), seen 4x, 31 tokens left
merge 7: b' low' + b' low' -> b' low low' (id 262), seen 3x, 29 tokens
      left
merge 8: b'n' + b'e' -> b'ne' (id 263), seen 3x, 26 tokens left
merge 9: b'ne' + b'w' -> b'new' (id 264), seen 3x, 23 tokens left
merge 10: b'new' + b'est ' -> b'newest ' (id 265), seen 3x, 20 tokens left

start: [108, 111, 119, 101, 115, 116]
apply (108, 111) -> 256: [256, 119, 101, 115, 116]
apply (256, 119) -> 257: [257, 101, 115, 116]
apply (101, 115) -> 259: [257, 259, 116]
apply (259, 116) -> 260: [257, 260]
token ids   : [257, 260]
token pieces: [b'low', b'est']
decoded     : lowest
round trip ok: True

'newer' -> [264, 101, 114] -> 'newer' True
'low low' -> [257, 258] -> 'low low' True
'caf\xe9 \U0001f600' -> [99, 97, 102, 195, 169, 32, 240, 159, 152, 128] ->
    'caf\xe9 \U0001f600' True
'zzz' -> [122, 122, 122] -> 'zzz' True
'' -> [] -> '' True

bytes in corpus : 62
tokens in corpus: 20

```

Compare the three blocks of output with the tables in Sections 7.3 and 7.4: they are the same numbers. Things to point out to yourself:

- **'newer'** becomes **new** + **e** + **r**. The letters **e** and **r** were never merged, because in ten merges the pair **er** was never the most common.
- The strings **zzz** and the empty string work too. The empty string gives an empty list.
- The final line shows the compression: 62 bytes became 20 tokens.

9 Can a transformer just read raw characters?

Yes! There is nothing in the transformer design that requires subword tokens. You can feed it characters (or raw bytes), with a vocabulary of a few hundred symbols, and it will learn to model language. Researchers have done exactly this, and it works. Character-level and byte-level transformers have been trained since at least 2018–2019, and systems such as ByT5 (a byte-level version of the T5 model) show competitive results on some tasks. So why does almost everybody still use subword tokens?

9.1 The main problem: cost grows with the square

Inside a transformer, *attention* compares every token with every other token. If the sequence has n tokens there are about $n \times n = n^2$ pairs to score, in every layer. Double the length and you do four times the work.

Take a 1000-word essay: about 6000 characters, but only about 1300 subword tokens.

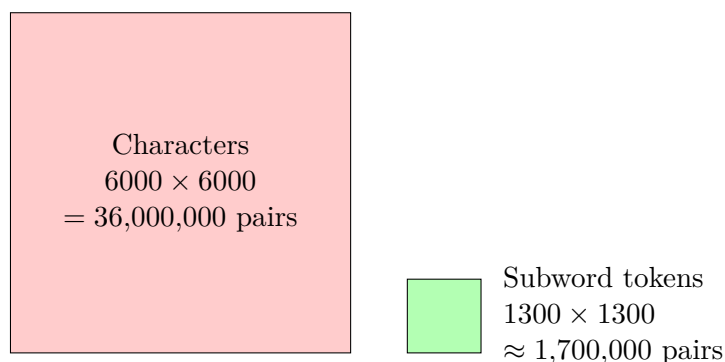


Figure 4: Attention score table for the same 1000-word essay (areas drawn to scale). Working with characters needs about 21 times as many comparisons.

9.2 Other costs of using characters

- **Slow text generation.** A model writes one token per step. Writing a 6000-character answer takes 6000 steps instead of about 1300.
- **Shorter effective memory.** The context window is measured in tokens, so a window of 8000 *characters* holds only a few pages of text, whereas 8000 subword tokens hold roughly five times as much.
- **More work to learn.** Every time, the model has to reassemble letters into words before it can think about meaning, which uses up model capacity and training time.

9.3 The benefits of using characters or bytes

- **No tokenizer to build, no unknown words,** and no strange token-boundary behavior.
- **Robust to typos,** unusual spellings, and rare languages.
- **Spelling-aware.** The model sees the letters, so counting the letters in a word is much more natural.
- **Simpler pipeline.** No training stage for a separate tokenizer.

9.4 Middle paths (active research)

Several research projects try to keep the benefits of bytes while avoiding the cost. They typically group bytes into *patches* inside the model, process the patches with a big transformer, and use small local models to handle the bytes within each patch. Examples include CANINE, MegaByte, and Meta’s Byte Latent Transformer. Despite this progress, most of the large language models in wide use today still use subword tokenizers such as BPE.

10 Other tokenization methods (alternatives to BPE)

Method	Examples of use	Main idea
Character byte level	Early research models; ByT5	One token per character or byte. Tiny vocabulary, long sequences.
Word level	Older systems	One token per word, with an <UNK> token for unknown words. Large vocabulary, loses rare words.
BPE	GPT-2 and later GPT models, Llama 3	Repeatedly merge the most frequent neighboring pair. Usually run on bytes (byte-level BPE).
WordPiece	BERT	Similar to BPE but chooses merges that most increase the likelihood of the training data (roughly, it favors pairs that occur together more often than chance). Marks pieces inside a word with a prefix such as ##, e.g. play + ##ing .
Unigram language model	T5, ALBERT (via SentencePiece)	Opposite direction: start with a <i>huge</i> candidate list and keep throwing away the least useful tokens. Chooses the most probable way to split a string, and can produce several alternative splits.
SentencePiece (a toolkit)	T5, Llama 1 and 2, many multilingual models	Treats the text as one raw stream including spaces, so it needs no language-specific word splitting. Can run either BPE or Unigram inside.

A few related ideas worth knowing:

- **Subword regularization / BPE-dropout.** During training of the *language model*, sometimes use a slightly different (randomly chosen) split of the same text. This makes the model more robust to how words are cut.
- **Tokenizer-free models.** The byte/patch approaches from the last section.
- **Non-text tokens.** Images are cut into small square patches (each patch is a “token”), and audio is chopped into short frames. The same transformer machinery applies.

11 Practical details real tokenizers add

Our toy program captures the heart of BPE. Production tokenizers add some important refinements:

1. **Pre-tokenization.** Before BPE runs, text is first split into rough chunks using a pattern (separating words, numbers, punctuation, and whitespace). Merges never cross chunk boundaries. Without this, BPE might learn separate tokens for “dog.” “dog!” and “dog?”, wasting vocabulary.
2. **Special tokens.** Reserved tokens that never come from merges, for example `<|endoftext|>` (end of a document), tokens for the start or end of a sequence, padding, and markers for who is speaking in a chat. They give the model structure.
3. **Vocabulary size is a design decision.** Typical sizes are around 30,000 (BERT), 50,000 (GPT-2), 100,000 (GPT-4), and 128,000 (Llama 3). A bigger vocabulary gives shorter sequences but needs a bigger embedding table. For example 50,000 tokens $\times$ 768 numbers per token is about 38 million numbers, just for the embedding table of a small model.
4. **Embeddings.** The model does not use the integer id directly, since “257” is not “bigger” or “more” than “256” in any meaningful way. Each id selects a row from a learned table, giving a list of numbers (a vector) that the network can work with. These vectors are learned during training so that tokens with similar uses end up with similar vectors.
5. **The training corpus shapes the tokenizer.** A tokenizer trained mostly on English learns many English merges, and handles other languages and code less efficiently, meaning more tokens for the same content. That affects cost, speed, and how much fits in the context window.
6. **Text normalization.** Some tokenizers first clean the text, for example by lowercasing or standardizing accented characters. This helps compression but can lose information. Byte-level BPE usually avoids this so the round trip is exact.
7. **The tokenizer is frozen and separate.** It is trained first, then fixed. The language model is trained afterwards on token ids from that particular tokenizer. You cannot swap in a different tokenizer without retraining the model, because the ids would no longer mean the same thing.
8. **Speed.** Our encoder recounts all pairs after each merge, which is simple but slow on long text. Production code uses much more efficient bookkeeping and is written in fast languages.
9. **Tokens are the unit of cost and capacity.** Context windows, speed and the price of using AI services are all measured in tokens, not in words.

12 Why tokenization explains odd AI behavior

Because the model sees token ids and not letters, a number of well-known surprises become understandable:

- **Counting or spelling letters.** Asked how many times a letter appears in a long word, a model may stumble: the word may be one or two tokens, and the letters inside are not directly visible to it.
- **Arithmetic.** Numbers are cut into chunks that may not line up with place value (some tokenizers group up to three digits, others one at a time), which makes digit-by-digit calculation harder.
- **Capitalization and spaces matter.** `Hello`, `hello`, and `␣hello` are typically three different tokens with three unrelated ids. A stray trailing space in a prompt changes the tokens.
- **Different languages cost different amounts.** The same sentence can take several times as many tokens in a language that was underrepresented in the tokenizer’s training text.
- **Glitchy rare tokens.** A token can end up in the vocabulary because it was common in the tokenizer’s data (say, a username), yet be rare in the model’s own training data. The model then has almost no experience with it and may behave strangely when it appears.
- **Code and whitespace.** Programming code, with its runs of indentation, is only handled efficiently if the tokenizer has merges for whitespace.

13 Key takeaways

1. Models work on lists of integers; the tokenizer converts text to ids and back.
2. Tokens are chunks between letters and words in size, found automatically from data.
3. Starting from UTF-8 bytes gives a universal 256-symbol alphabet and guarantees that nothing is ever “unknown.”
4. **BPE training:** repeatedly merge the most frequent pair of neighboring tokens; record the rules in order.
5. **BPE encoding:** convert to bytes, then apply the learned rules, earliest first. **Decoding:** look up bytes, join, and read as UTF-8.
6. Transformers *can* work on raw characters or bytes, but sequences get much longer and attention cost grows with the square of the length.
7. Alternatives include WordPiece, Unigram/SentencePiece, and tokenizer-free byte-level models.

8. Many quirks of language models (spelling, counting, arithmetic, multilingual cost) come from tokenization.