Lecture 4: Tokenization

CSC 352 / CSC 552 · 2026-09-08

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

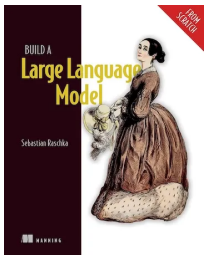
Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class we wrote a class that subclasses `torch.nn.Module`. What method did we have to define?
2. In the training loop, what is the order: `backward`, `optimizer.step`, `zero_grad`?
3. What does a `DataLoader` give you on each iteration: one row, or a batch?

Supplementary code for the Build a Large Language Model From Scratch book by Sebastian Raschka

Code repository:
https://github.com/rasbt/LLMs-from-scratch



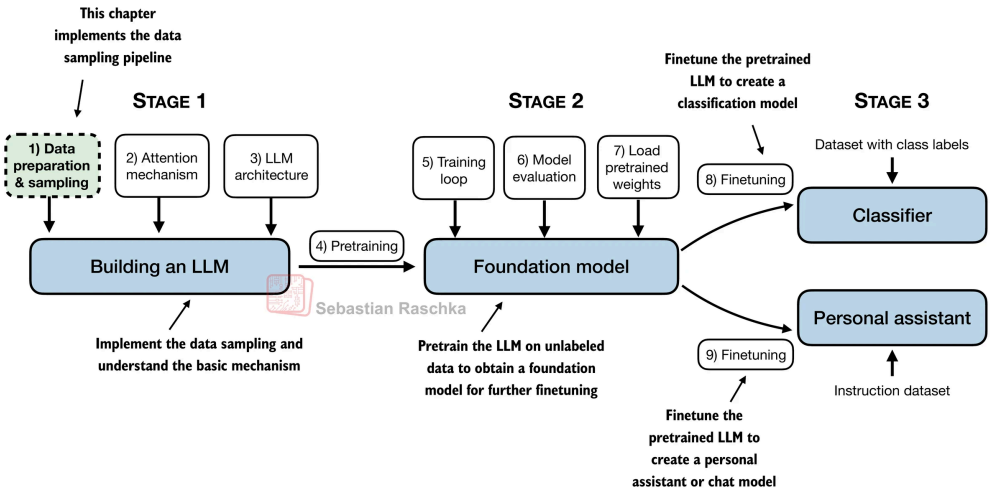
Packages that are being used in this notebook:

In [5]:
```
from importlib.metadata import version

print("torch version:", version("torch"))
print("tiktoken version:", version("tiktoken"))
```

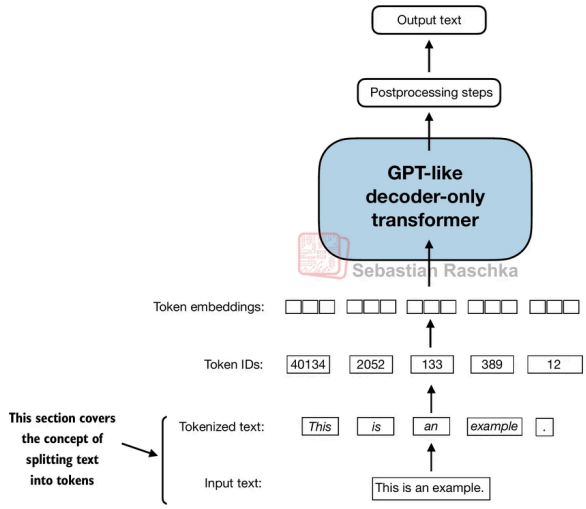
```
torch version: 2.8.0
tiktoken version: 0.14.0
```

- This chapter covers data preparation and sampling to get input data "ready" for the LLM



2.2 Tokenizing text

- In this section, we tokenize text, which means breaking text into smaller units, such as individual words and punctuation characters



- Load raw text we want to work with
- The Verdict by Edith Wharton is a public domain short story

In [18]:
```
import os
import requests

if not os.path.exists("the-verdict.txt"):
    url = (
        "https://raw.githubusercontent.com/rasbt/"
        "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
        "the-verdict.txt"
    )
```

```
    file_path = "the-verdict.txt"

    response = requests.get(url, timeout=30)
    response.raise_for_status()
    with open(file_path, "wb") as f:
        f.write(response.content)


# The book originally used the following code below
# However, urllib uses older protocol settings that
# can cause problems for some readers using a VPN.
# The `requests` version above is more robust
# in that regard.

"""
import os
import urllib.request

if not os.path.exists("the-verdict.txt"):
    url = ("https://raw.githubusercontent.com/rasbt/"
           "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
           "the-verdict.txt")
    file_path = "the-verdict.txt"
    urllib.request.urlretrieve(url, file_path)
"""
```

Out[18]:
```
'\nimport os\nimport urllib.request\n\nif not os.path.exists("the-verdict.t
xt"):\n    url = ("https://raw.githubusercontent.com/rasbt/"\n           "L
LMs-from-scratch/main/ch02/01_main-chapter-code/"\n           "the-verdict.
txt")\n    file_path = "the-verdict.txt"\n    urllib.request.urlretrieve(ur
l, file_path)\n'
```

Reload the short story as a Python string.

In [20]:
```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

```
Total number of character: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fello
w enough--so it was no
```

- The goal is to tokenize and embed this text for an LLM
- Let's develop a simple tokenizer based on some simple sample text that we can then later apply to the text above
- The following regular expression will split on whitespaces

In [22]:
```
import re

text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
```

```
print(result)
```

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ', 'test.']
```

- We don't only want to split on whitespaces but also commas and periods, so let's modify the regular expression to do that as well

In [24]:
```
result = re.split(r'([,.]|\s)', text)

print(result)
```

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', ' ', 'is', '
', 'a', ' ', 'test', '.', '']
```

- As we can see, this creates empty strings, let's remove them

In [26]:
```
# Strip whitespace from each item and then filter out any empty strings.
result = [item for item in result if item.strip()]
print(result)
```

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

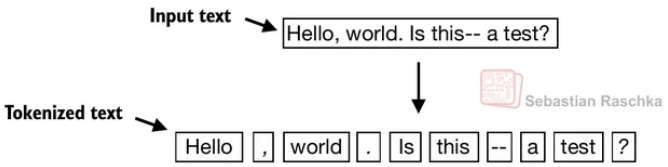
- This looks pretty good, but let's also handle other types of punctuation, such as periods, question marks, and so on

In [28]:
```
text = "Hello, world. Is this-- a test?"

result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

- This is pretty good, and we are now ready to apply this tokenization to the raw text



In [31]:
```
preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(preprocessed[:30])
```

```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap',
'genius', '--', 'though', 'a', 'good', 'fellow', 'enough', '--', 'so', 'it',
'was', 'no', 'great', 'surprise', 'to', 'me', 'to', 'hear', 'that', ',', 'i
n']
```

- Let's calculate the total number of tokens

In [33]:
```
print(len(preprocessed))
```

```
4690
```

Exercise (2.2 Tokenizing text)

Split `text_ex` using the same tokenizer regex as this section. Print the token list and how many tokens that is.

In [ ]:
```
# Exercise 2.2 — your attempt
import re
text_ex = "The LLM reads text."
tokens_ex = None
n_tokens = None
print(tokens_ex)
print(n_tokens)
```

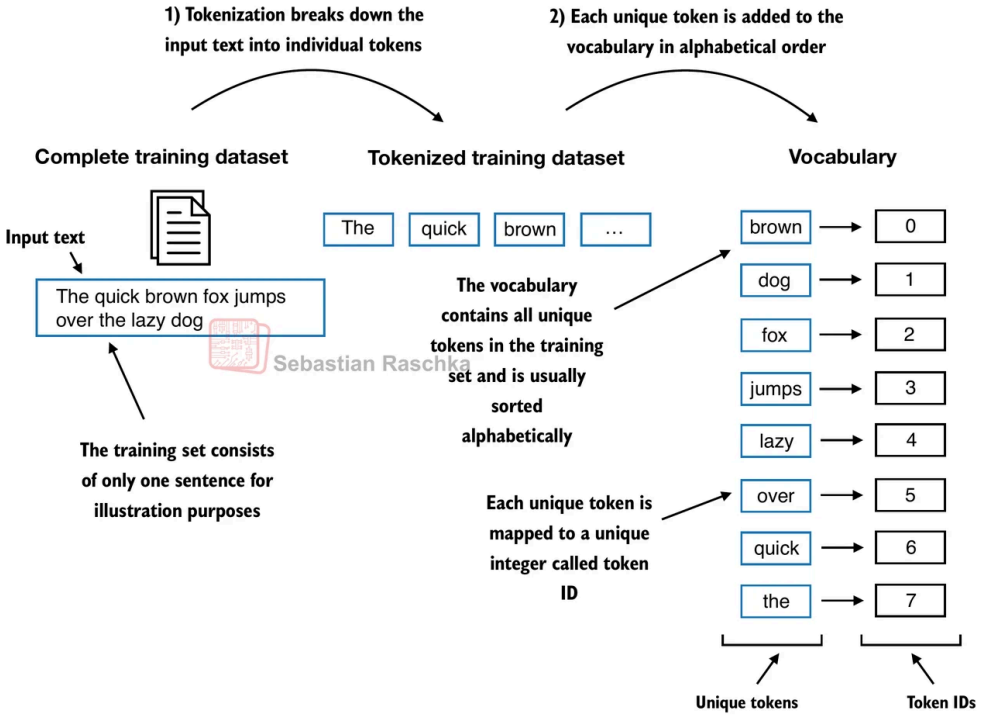
Solution (2.2). Punctuation is its own token. Spaces were stripped.

In [37]:
```
# Solution 2.2
import re
text_ex = "The LLM reads text."
tokens_ex = re.split(r'([,.:;?_!"()\']|--|\s)', text_ex)
tokens_ex = [item.strip() for item in tokens_ex if item.strip()]
print(tokens_ex)
print(len(tokens_ex))
```

```
['The', 'LLM', 'reads', 'text', '.']
5
```

2.3 Converting tokens into token IDs

- Next, we convert the text tokens into token IDs that we can process via embedding layers later



- From these tokens, we can now build a vocabulary that consists of all the unique tokens

In [42]:
```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)

print(vocab_size)
```

```
1130
```

Turn each unique token into an integer id.

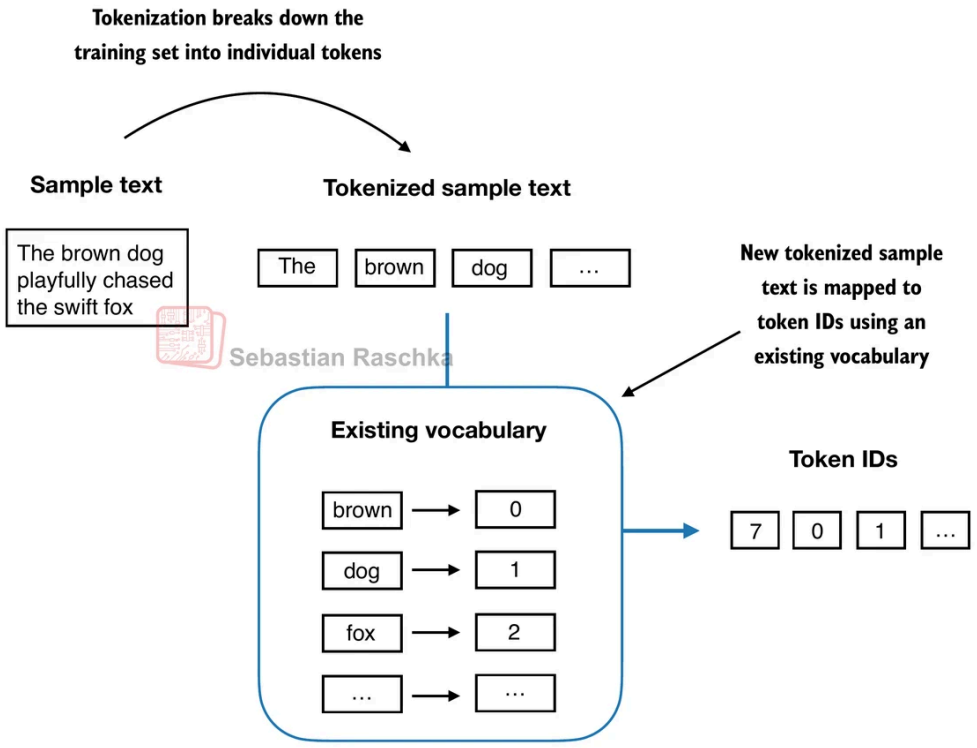
In [44]:
```
vocab = {token:integer for integer,token in enumerate(all_words)}
```

- Below are the first 50 entries in this vocabulary:

In [46]:
```
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

```
('!', 0)
('"', 1)
("'", 2)
('(', 3)
(')', 4)
(',', 5)
('--', 6)
('.', 7)
(':', 8)
(';', 9)
('?', 10)
('A', 11)
('Ah', 12)
('Among', 13)
('And', 14)
('Are', 15)
('Arrt', 16)
('As', 17)
('At', 18)
('Be', 19)
('Begin', 20)
('Burlington', 21)
('But', 22)
('By', 23)
('Carlo', 24)
('Chicago', 25)
('Claude', 26)
('Come', 27)
('Croft', 28)
('Destroyed', 29)
('Devonshire', 30)
('Don', 31)
('Dubarry', 32)
('Emperors', 33)
('Florence', 34)
('For', 35)
('Gallery', 36)
('Gideon', 37)
('Gisburn', 38)
('Gisburns', 39)
('Grafton', 40)
('Greek', 41)
('Grindle', 42)
('Grindles', 43)
('HAD', 44)
('Had', 45)
('Hang', 46)
('Has', 47)
('He', 48)
('Her', 49)
('Hermia', 50)
```

- Below, we illustrate the tokenization of a short sample text using a small vocabulary:



- Putting it now all together into a tokenizer class

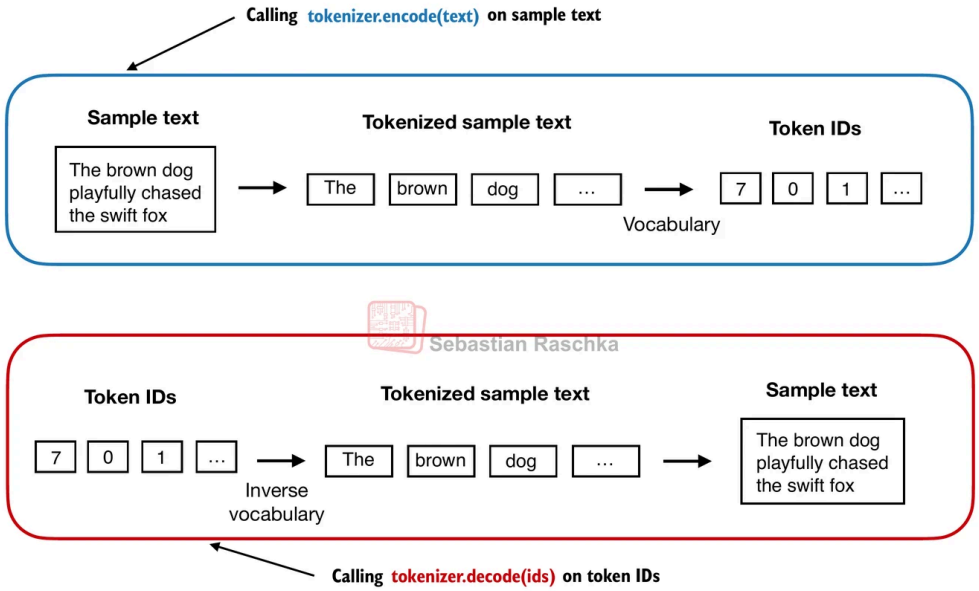
In [50]:
```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)

        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        # Replace spaces before the specified punctuations
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

- The `encode` function turns text into token IDs
- The `decode` function turns token IDs back into text



- We can use the tokenizer to encode (that is, tokenize) texts into integers
- These integers can then be embedded (later) as input of/for the LLM

In [54]:
```
tokenizer = SimpleTokenizerV1(vocab)

text = """It's the last he painted, you know,"
           Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)
```

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 110
8, 754, 793, 7]
```

- We can decode the integers back into text

In [56]:
```
tokenizer.decode(ids)
```

Out[56]:
```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable
pride.'
```

Decode those ids and read the reconstructed string.

In [58]:
```
tokenizer.decode(tokenizer.encode(text))
```

Out[58]:
```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable
pride.'
```

Exercise (2.3 Token IDs)

Look up two tokens in `vocab` from this section: the period and the word I. Print both integer ids.

In [ ]:
```
# Exercise 2.3 — your attempt
id_period = None
id_I = None
print('id of . =', id_period)
print('id of I =', id_I)
```

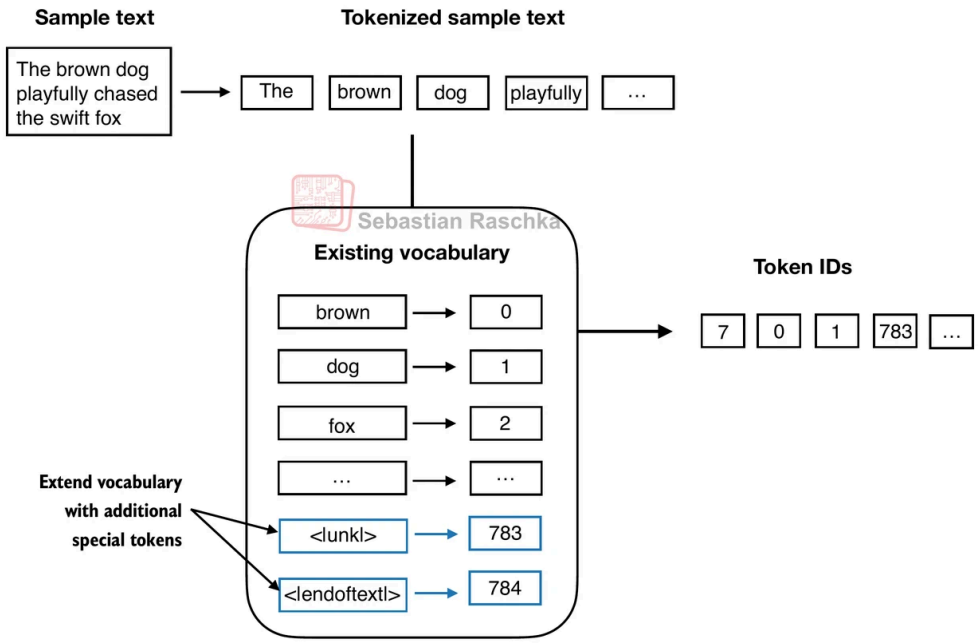
Solution (2.3). Ids come from sorting unique tokens, not from spelling.

In [62]:
```
# Solution 2.3
print('id of . =', vocab['.'])
print('id of I =', vocab['I'])
```

```
id of . = 7
id of I = 53
```

2.4 Adding special context tokens

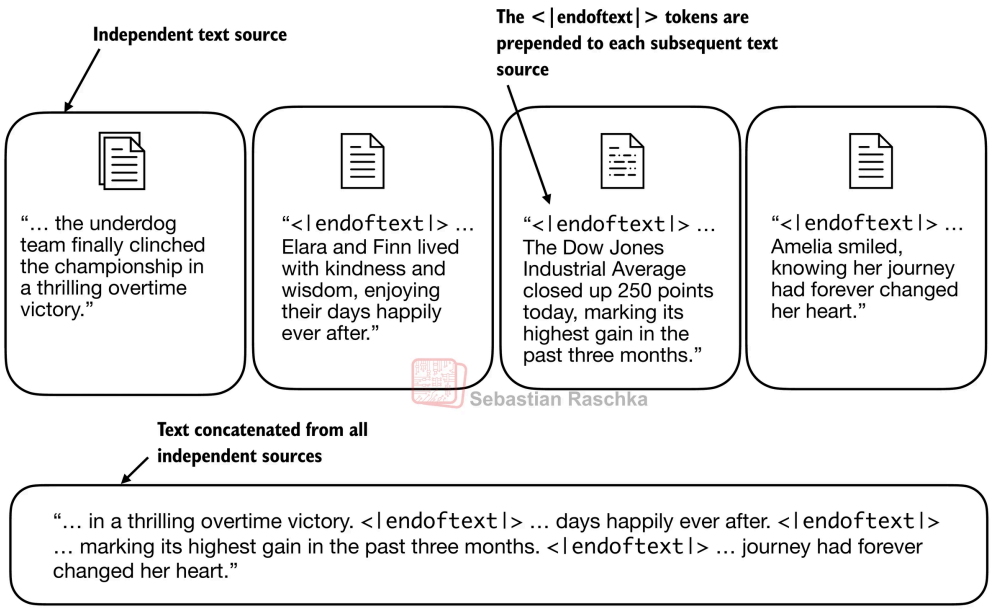
- It's useful to add some "special" tokens for unknown words and to denote the end of a text



- Some tokenizers use special tokens to help the LLM with additional context
- Some of these special tokens are
  - `[BOS]` (beginning of sequence) marks the beginning of text
  - `[EOS]` (end of sequence) marks where the text ends (this is usually used to concatenate multiple unrelated texts, e.g., two different Wikipedia articles or two different books, and so on)

    - `[PAD]` (padding) if we train LLMs with a batch size greater than 1 (we may include multiple texts with different lengths; with the padding token we pad the shorter texts to the longest length so that all texts have an equal length)
- `[UNK]` to represent words that are not included in the vocabulary
- Note that GPT-2 does not need any of these tokens mentioned above but only uses an `<|endoftext|>` token to reduce complexity
- The `<|endoftext|>` is analogous to the `[EOS]` token mentioned above
- GPT also uses the `<|endoftext|>` for padding (since we typically use a mask when training on batched inputs, we would not attend padded tokens anyways, so it does not matter what these tokens are)
- GPT-2 does not use an `<UNK>` token for out-of-vocabulary words; instead, GPT-2 uses a byte-pair encoding (BPE) tokenizer, which breaks down words into subword units which we will discuss in a later section

- We use the `<|endoftext|>` tokens between two independent sources of text:



- Let's see what happens if we tokenize the following text:

In [70]:
```
tokenizer = SimpleTokenizerV1(vocab)

text = "Hello, do you like tea. Is this-- a test?"

tokenizer.encode(text)
```

```
Traceback (most recent call last):
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classroom_nb.py", line 227, in execute_notebook
    last_val, had_expr = _exec_with_result(runnable, env, f"cell_{i}")
  File "/Users/rongyulin/Desktop/vibe_projects/Teaching/fall2026_prep/classroom_nb.py", line 150, in _exec_with_result
    last_val = eval(compile(ast.Expression(tree.body[-1].value), filename, "eval"), env)
  File "cell_69", line 5, in <module>
  File "cell_49", line 12, in encode
  File "cell_49", line 12, in <listcomp>
KeyError: 'Hello'
```

- The above produces an error because the word "Hello" is not contained in the vocabulary
- To deal with such cases, we can add special tokens like `"<|unk|>"` to the vocabulary to represent unknown words
- Since we are already extending the vocabulary, let's add another token called `"<|endoftext|>"` which is used in GPT-2 training to denote the end of a text (and it's also used between concatenated text, like if our training datasets consists of multiple articles, books, etc.)

In [72]:
```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])

vocab = {token:integer for integer,token in enumerate(all_tokens)}
```

Vocabulary size after adding the special tokens.

In [74]:
```
len(vocab.items())
```

Out[74]: 1132

The last two rows should be the special tokens we just added.

In [76]:
```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

- We also need to adjust the tokenizer accordingly so that it knows when and how to use the new `<unk>` token

In [78]:
```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
```

```
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip(
        preprocessed = [
            item if item in self.str_to_int
            else "<|unk|>" for item in preprocessed
        ]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        # Replace spaces before the specified punctuations
        text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', text)
        return text
```

Let's try to tokenize text with the modified tokenizer:

In [80]:
```
tokenizer = SimpleTokenizerV2(vocab)

text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."

text = " <|endoftext|> ".join((text1, text2))

print(text)
```

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of the palace.
```

Encode the joined text; unknown words should become `<|unk|>`.

In [82]:
```
tokenizer.encode(text)
```

Out[82]:
```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131,
7]
```

Decode those ids and read the reconstructed string.

In [84]:
```
tokenizer.decode(tokenizer.encode(text))
```

Out[84]:
```
'<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of the <|un
k|>.'
```

Exercise (2.4 Special tokens)

Encode `Hello palace.` with `SimpleTokenizerV2`. Print the ids and the decoded string. Which pieces became the unknown token?

In [ ]:
```
# Exercise 2.4 — your attempt
tok_v2 = SimpleTokenizerV2(vocab)
ids_ex = None
```

```
print(ids_ex)
print()
```

Solution (2.4). Words not in *The Verdict* vocab become `<|unk|>`. `.` is in the vocab.

In [88]:
```
# Solution 2.4
tok_v2 = SimpleTokenizerV2(vocab)
ids_ex = tok_v2.encode('Hello palace.')
print(ids_ex)
print(tok_v2.decode(ids_ex))
```

```
[1131, 1131, 7]
<|unk|> <|unk|>.
```

2.5 BytePair encoding

- GPT-2 used BytePair encoding (BPE) as its tokenizer
- it allows the model to break down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words
- For instance, if GPT-2's vocabulary doesn't have the word "unfamiliarword," it might tokenize it as ["unfam", "iliar", "word"] or some other subword breakdown, depending on its trained BPE merges
- The original BPE tokenizer can be found here: https://github.com/openai/gpt-2/blob/master/src/encoder.py
- In this chapter, we are using the BPE tokenizer from OpenAI's open-source tiktoken library, which implements its core algorithms in Rust to improve computational performance
- I created a notebook in the ./bytepair_encoder that compares these two implementations side-by-side (tiktoken was about 5x faster on the sample text)

In [91]:
```
# pip install tiktoken
```

The next cell runs `import importlib import tiktoken print("tiktoken version:", importlib.me`.

In [93]:
```
import importlib
import tiktoken

print("tiktoken version:", importlib.metadata.version("tiktoken"))
```

```
tiktoken version: 0.14.0
```

Load the GPT-2 BPE table (already trained — this is a lookup, not a re-train).

In [95]:
```
tokenizer = tiktoken.get_encoding("gpt2")
```

A sentence with `<|endoftext|>` and a made-up word. Encode it next.

In [97]:
```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
     "of someunknownPlace."
)

integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

print(integers)
```

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 881
2, 2114, 1659, 617, 34680, 27271, 13]
```

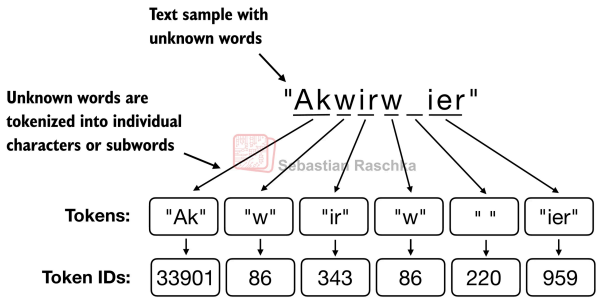
Decode the BPE ids. Unknown spellings come back as subwords, not `<|unk|>`.

In [99]:
```
strings = tokenizer.decode(integers)

print(strings)
```

```
Hello, do you like tea? <|endoftext|> In the sunlit terracesof someunknownPl
ace.
```

- BPE tokenizers break down unknown words into subwords and individual characters:



Exercise (2.5 BPE)

Encode `someunknownPlace` with the GPT-2 tokenizer already in memory. Decode each id by itself so you can see the subword pieces.

In [ ]:
```
# Exercise 2.5 — your attempt
ids_bpe = None
print(ids_bpe)
for i in ids_bpe:
    piece = None
    print(i, '->', piece)
```

Solution (2.5). BPE does not need `<|unk|>` here: it breaks the spelling into pieces already in the table.

In [105…
```python
# Solution 2.5
ids_bpe = tokenizer.encode('someunknownPlace')
print(ids_bpe)
for i in ids_bpe:
    print(i, '->', tokenizer.decode([i]))
```

```
[11246, 34680, 27271]
11246 -> some
34680 -> unknown
27271 -> Place
```

Comprehensive exercises

Exercise (comprehensive: tokenize with BPE)

Take the sentence `Hello, do you like tea?`. Encode it with the GPT-2 BPE tokenizer. Print the id list, how many ids that is, and the string you get back after decoding.

In [ ]:
```python
# Exercise comprehensive: your attempt
import tiktoken
enc = tiktoken.get_encoding("gpt2")
raw = "Hello, do you like tea?"
ids = None
print(ids)
print()
print()
```

Solution. The GPT-2 table already knows these English pieces, so you get a short id list, not `<|unk|>`.

In [ ]:
```python
# Solution comprehensive
import tiktoken
enc = tiktoken.get_encoding("gpt2")
raw = "Hello, do you like tea?"
ids = enc.encode(raw)
print(ids)
print(len(ids))
print(enc.decode(ids))
```

Takeaways

1. **Tokenize** means split text into pieces (words, punctuation, later subwords).
2. A **vocabulary** maps each piece to a **token id** (an integer).
3. Special tokens such as `<|unk|>` and `<|endoftext|>` are extra rows in that table.
4. **BPE** (GPT-2 / `tiktoken`) breaks unknown spellings into subwords instead of crashing.

5. Next class: windows of ids for training, then an embedding table turns ids into vectors.