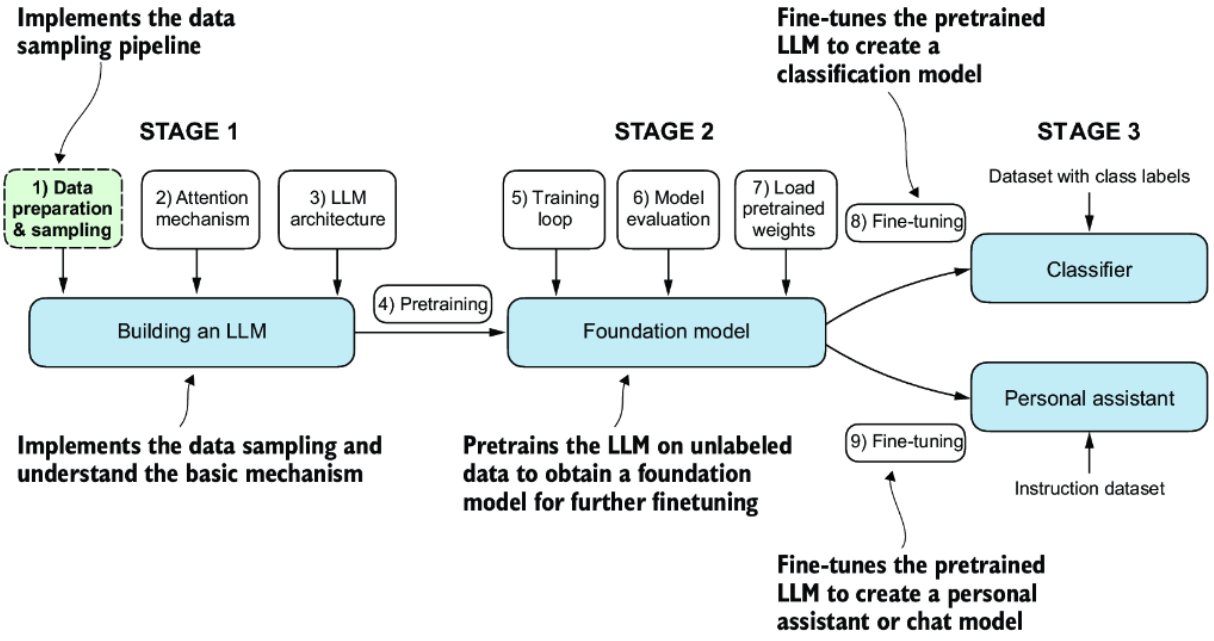
chapter 2

The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.



what are tensors:

Let's take a small 2×2 pixel image in RGB format as an example to explain how an image can be represented as a 3D matrix.

Example: 2x2 RGB Image

We have an image with the following pixel colors:

- Top-left pixel: **Red** (RGB = [255, 0, 0])
- Top-right pixel: **Green** (RGB = [0, 255, 0])
- Bottom-left pixel: **Blue** (RGB = [0, 0, 255])
- Bottom-right pixel: **White** (RGB = [255, 255, 255])

1. Image Representation (Pixels)

The image looks like this:

Red (255, 0, 0)	Green (0, 255, 0)

Blue (0, 0, 255)	White (255, 255, 255)

2. 3D Matrix Representation

The image can be represented as a 3D matrix where:

- The first dimension represents the **height** (2 rows of pixels).
- The second dimension represents the **width** (2 columns of pixels).
- The third dimension represents the **color channels** (RGB channels for each pixel).

So, the matrix will be structured as:

```
[
    [[255, 0, 0], [0, 255, 0]],   # First row: Red, Green
    [[0, 0, 255], [255, 255, 255]] # Second row: Blue, Whit
e
]
```

Here's a breakdown:

- The first row represents the first pixel (Red) and the second pixel (Green).
- The second row represents the third pixel (Blue) and the fourth pixel (White).

3. Matrix Explanation

Let's break down this 3D matrix step by step:

Height = 2	**Width = 2**	**Channels = 3 (RGB)**

For pixel positions:

- **(0,0)** (top-left): `[255, 0, 0]` → Red
- **(0,1)** (top-right): `[0, 255, 0]` → Green
- **(1,0)** (bottom-left): `[0, 0, 255]` → Blue
- **(1,1)** (bottom-right): `[255, 255, 255]` → White

So the matrix dimensions are `(2, 2, 3)`:

- **2 rows** for height,

- **2 columns** for width,
- **3 channels** for RGB.

Visualization

If we print the 3D matrix:

```
[
    [ [255, 0, 0], [0, 255, 0] ],        # First row
    [ [0, 0, 255], [255, 255, 255] ]     # Second row
]
```

- The **first sub-array** `[255, 0, 0]` represents the RGB values of the top-left pixel (Red).
- The **second sub-array** `[0, 255, 0]` represents the RGB values of the top-right pixel (Green).
- The **third sub-array** `[0, 0, 255]` represents the RGB values of the bottom-left pixel (Blue).
- The **fourth sub-array** `[255, 255, 255]` represents the RGB values of the bottom-right pixel (White).

This is a very simple representation, but you can easily scale it up for larger images where the structure follows the same logic: height × width × color channels.

Example in Python (using NumPy)

Here's how you can represent this image in Python:

```
import numpy as np

# 2x2 image with RGB channels
image_matrix = np.array([[[255, 0, 0], [0, 255, 0]],   # Fi
rst row: Red, Green
                         [[0, 0, 255], [255, 255, 255]]]) #
Second row: Blue, White

# Print the shape of the 3D matrix (it will be 2x2x3)
print(image_matrix.shape)  # Output: (2, 2, 3)
```

```
# Print the 3D matrix
print(image_matrix)
```

Output:

```
(2, 2, 3)
[[[255   0   0]
  [  0 255   0]]

 [[  0   0 255]
  [255 255 255]]]
```

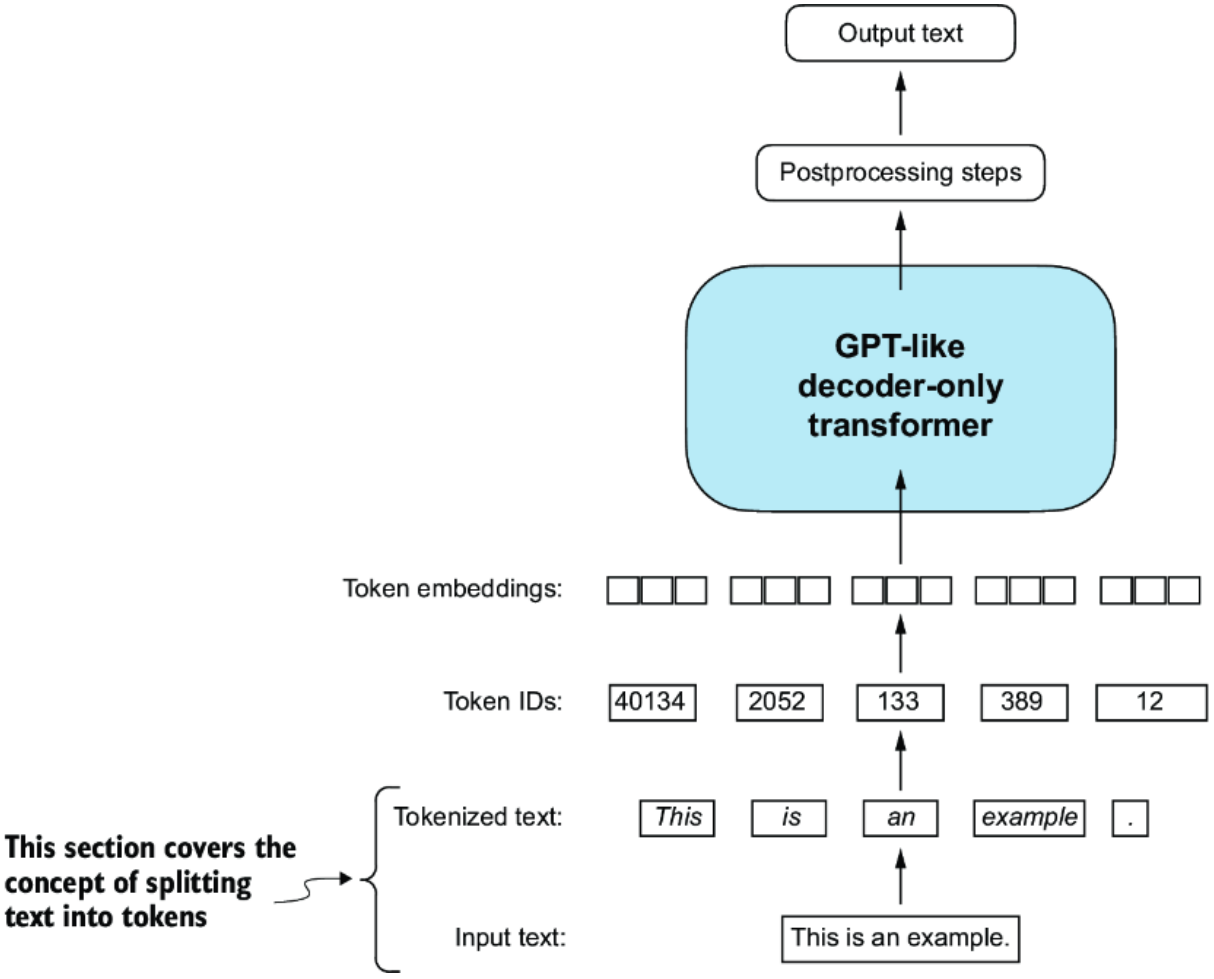
This small example demonstrates how an image can be represented as a 3D matrix where each element is a pixel's color values in the RGB channels.

While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for *retrieval-augmented generation*.

While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand. We will implement such embedding layers later in this chapter. (LLMs can also create contextualized output embeddings, as we discuss in chapter 3.)

Next, we will walk through the required steps for preparing the embeddings used by an LLM, which include splitting text into words, converting words into tokens, and turning tokens into embedding vectors.

A view of the text processing steps in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters.



1. The text we will tokenize for LLM training is “The Verdict,” a short story by Edith Wharton, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict, and you can copy and paste it into a text file, which I copied into a text file `"verdict.txt".`
2. Next, we can load the `the-verdict.txt` file using Python’s standard file reading utilities.
   The print command prints the total number of characters followed by the first 100 characters of this file for illustration purposes

```
with open("verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

3. Our goal is to tokenize this 20,479-character short story into individual words and special characters that we can then turn into embeddings for LLM training.

   Note It's common to process millions of articles and hundreds of thousands of books—many gigabytes of text—when working with LLMs. However, for educational purposes, it's sufficient to work with smaller text samples like a single book to illustrate the main ideas behind the text processing steps and to make it possible to run it in a reasonable time on consumer hardware.

   Building a Tokenizer:

   - Use the "re" python library to split text and special characters into individual items of a list.

     ```
     import re
     with open("verdict.txt", "r", encoding="utf-8") as f:
         raw_text = f.read()
     print("Total number of character:", len(raw_text))
     # print(raw_text[:99])
     preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_
     preprocessed = [item.strip() for item in preprocessed i
     # print(len(preprocessed))
     print(preprocessed[:30])
     ```

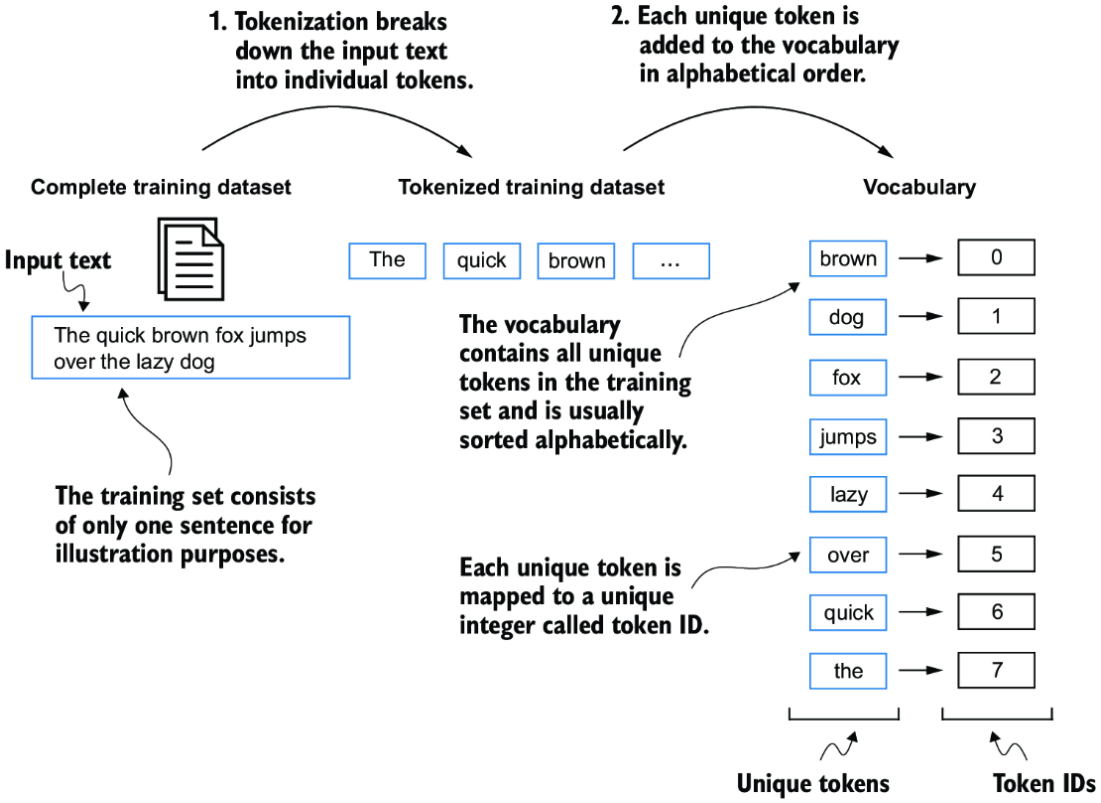
     Total number of character: 20480
     ['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap', 'genius', '--', 'though', 'a', 'good', 'fellow', 'enough', '--', 'so', 'it', 'was', 'no', 'great', 'surprise', 'to', 'me', 'to', 'hear', 'that', ',', 'in']

4. **Converting tokens into token IDs:**

   convert these tokens from a Python string to an integer representation to produce the token IDs. This conversion is an intermediate step before converting the token IDs into embedding vectors.

   - To map the previously generated tokens into token IDs, we have to build a vocabulary first. This vocabulary defines how we map each unique word and special character to a unique integer

- We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposefully small and contains no punctuation or special characters for simplicity.



- Now that we have tokenized Edith Wharton's short story and assigned it to a Python variable called `preprocessed`, let's create a list of all unique tokens and sort them alphabetically to determine the vocabulary size:

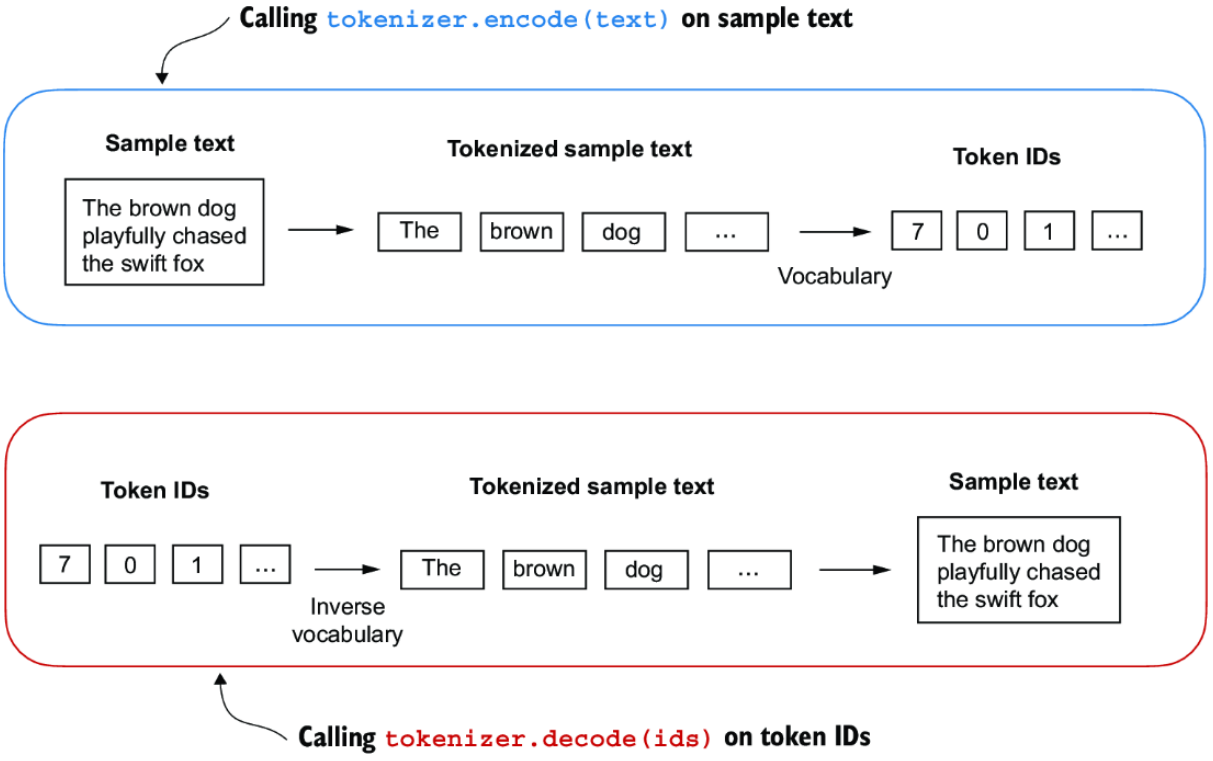
```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

- vocabulary size is 1,130 via this code
- we create the vocabulary and print its first 51 entries for illustration purposes

```
vocab = {token:integer for integer,token in enumerate(all_
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

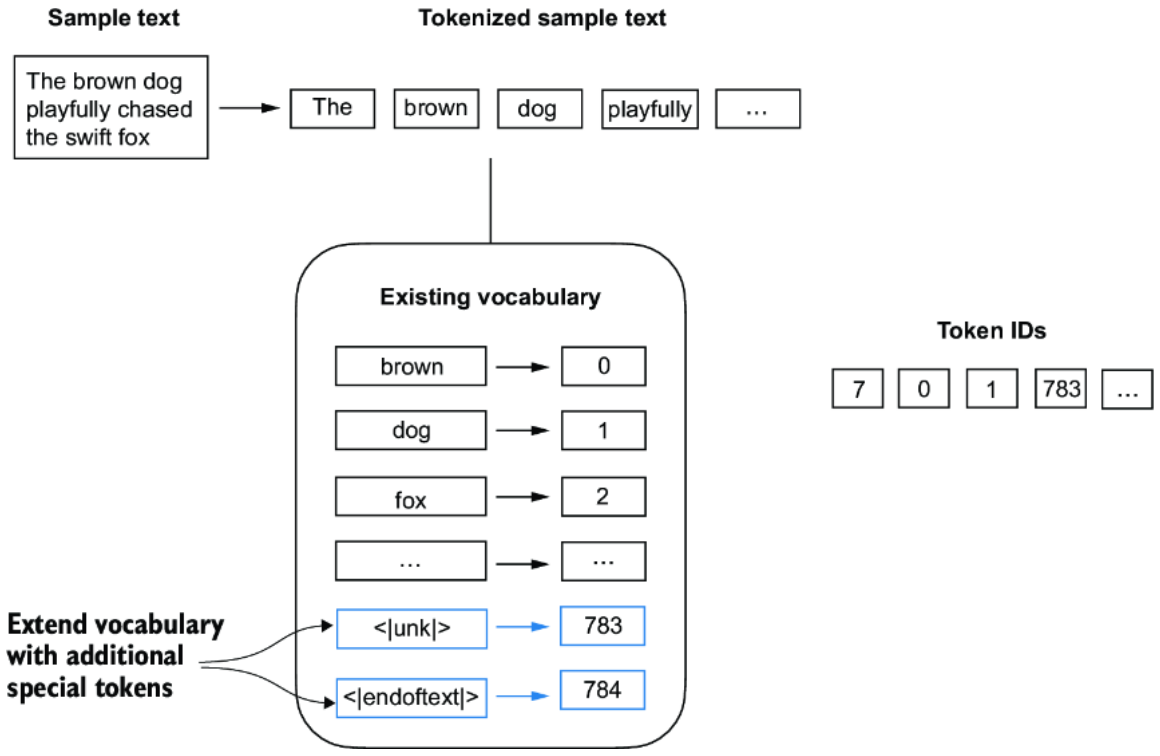
('!', 0)
('"', 1)
("'", 2)
...
('Her', 49)
('Hermia', 50)

- Converting new text into token id's (eg. a user prompt): (ENCODING)
- Converting token id's into text (eg. LLM response to readable text) (DECODING)



5. **Adding special context tokens:**

   We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an <|unk|> token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an <|endoftext|> token that we can use to separate two unrelated text sources.



- Let's now modify the vocabulary to include these two special tokens, `<unk>` and `<|endoftext|>`, by adding them to our list of all unique words:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(a

print(len(vocab.items()))
```

```
# A simple text tokenizer that handles unknown words
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
        preprocessed = [
            item.strip() for item in preprocessed if it
        ]
        preprocessed = [item if item in self.str_to_int
                        else "<|unk|>" for item in prep

        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 id

        text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', tex
        return text
```

- Depending on the LLM, some researchers also consider additional special tokens such as the following:
- `[BOS]` *(beginning of sequence)*—This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- `[EOS]` *(end of sequence)*—This token is positioned at the end of a text and is especially useful when concatenating multiple unrelated texts, similar to `<|endoftext|>`. For instance, when combining two different Wikipedia articles or books, the `[EOS]` token indicates where one ends and the next begins.
- `[PAD]` *(padding)*—When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the `[PAD]` token, up to the length of the longest text in the batch.

6. **Byte pair encoding:**

The BPE tokenizer was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.
Since implementing BPE can be relatively complicated, we will use an existing Python open source library called *tiktoken* (https://github.com/openai/tiktoken), which implements the BPE algorithm very efficiently based on source code in Rust. Similar to other Python libraries, we can install the tiktoken library via Python's `pip` installer from the terminal:
pip install tiktoken

-

The usage of this tokenizer is similar to the `SimpleTokenizerV2` we implemented previously via an `encode` method:

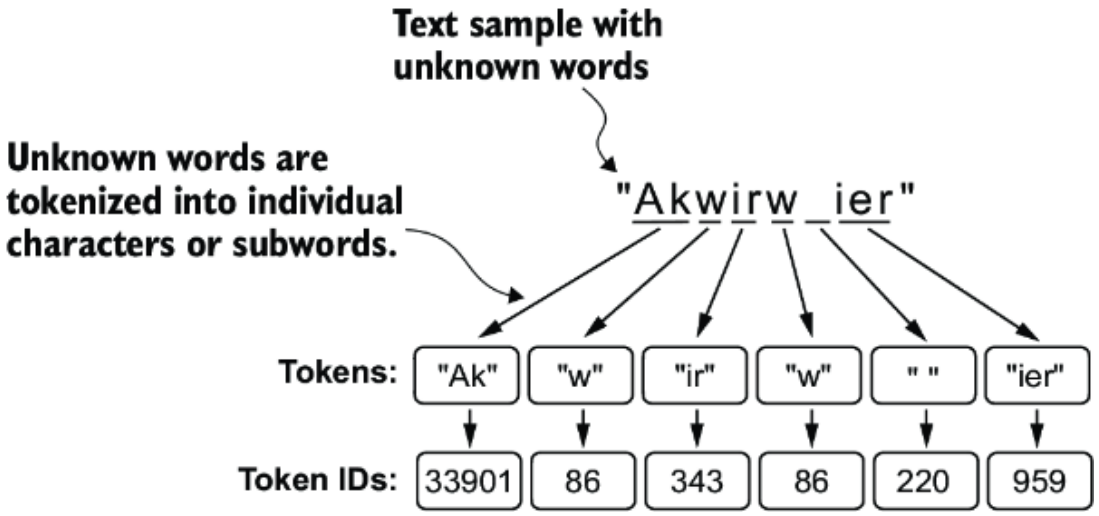
```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
tokenizer = tiktoken.get_encoding("gpt2")
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit t
     "of someunknownPlace."
)
ids = tokenizer.encode(text, allowed_special={"<|endoftext
print(ids)
```

- We can then convert the token IDs back into text using the decode method, similar to our `SimpleTokenizerV2`:

  ```
  strings = tokenizer.decode(ids)
  print(strings)
  ```

- We can make two noteworthy observations based on the token IDs and decoded text. First, the `<|endoftext|>` token is assigned a relatively large token ID, namely, `50256`. In fact, the BPE tokenizer, which was used to train models such as GPT-2, GPT-3, and the original model used in ChatGPT, has a total vocabulary size of 50,257, with `<|endoftext|>` being assigned the largest token ID.

- Second, the BPE tokenizer encodes and decodes unknown words, such as `someunknownPlace`, correctly. The BPE tokenizer can handle any unknown word. How does it achieve this without using `<|unk|>` tokens?

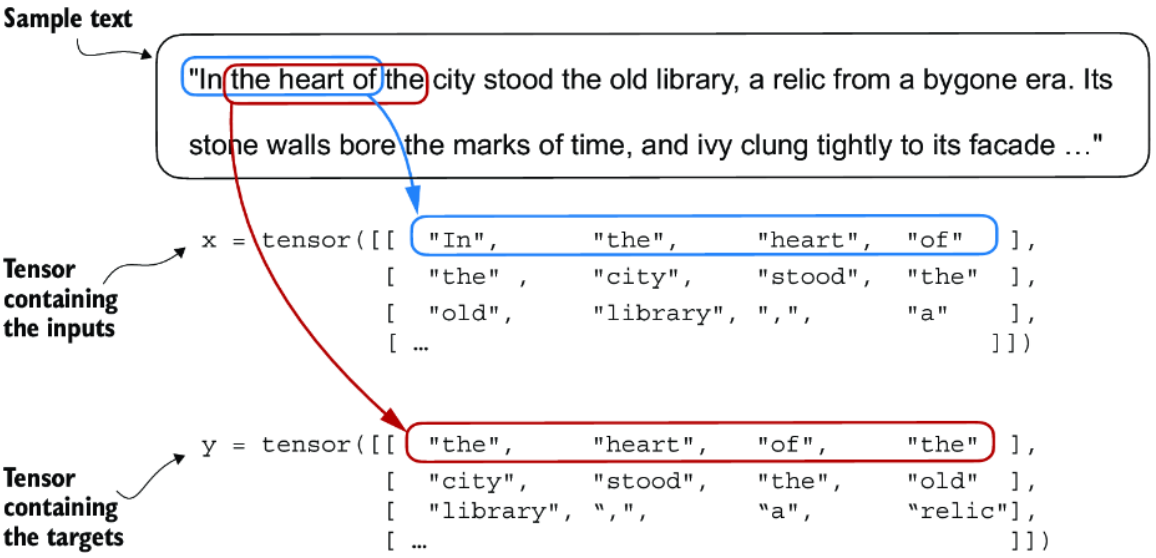


- BPE builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary ("a," "b," etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, "d" and "e" may be merged into the subword "de," which is common in many English words like "define," "depend," "made," and "hidden." The merges are determined by a frequency cutoff.

7. There's only one more task before we can turn the tokens into embeddings: implementing an efficient data loader that iterates over the input dataset and returns the inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure

- the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

- **To implement efficient data loaders, we collect the inputs in a tensor, `x`, where each row represents one input context. A second tensor, `y`, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.**

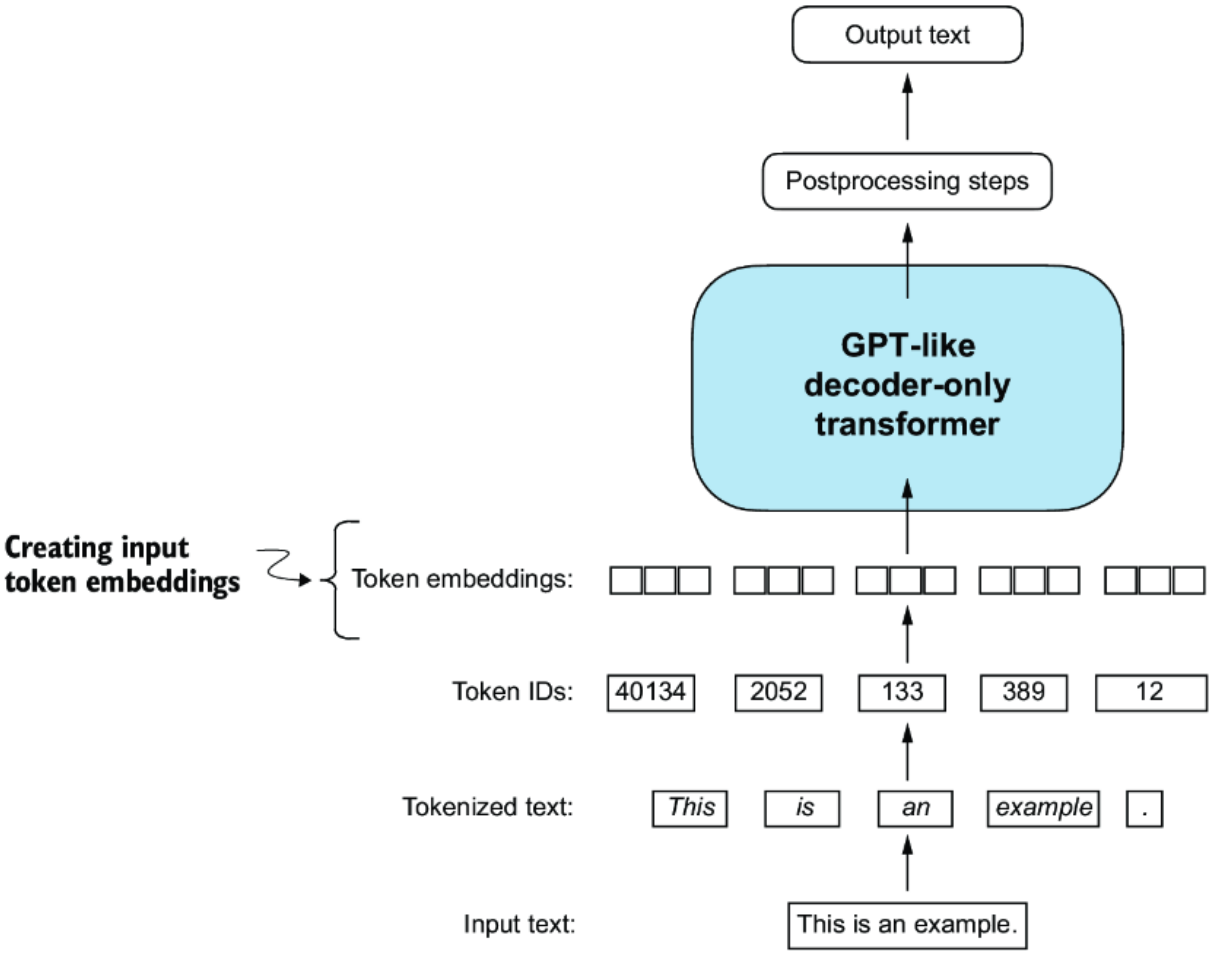


- **Note** For the efficient data loader implementation, we will use PyTorch's built-in `Dataset` and `DataLoader` classes. For additional information and guidance on installing PyTorch, please see section A.2.1.3 in appendix A.
-

8. Creating token embeddings

The last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors, as shown in figure 2.15. As a preliminary step, we must initialize these embedding weights with random values. This initialization serves as the starting point for the LLM's learning process.

Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.



Let's see how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and :

```
input_ids = torch.tensor([2, 3, 5, 1])
```

For the sake of simplicity, suppose we have a small vocabulary of only 6 words (instead of the 50,257 words in the BPE tokenizer vocabulary), and we want to create embeddings of size 3 (in GPT-3, the embedding size is 12,288 dimensions):

```
vocab_size = 6
output_dim = 3
```

Using the `vocab_size` and `output_dim`, we can instantiate an embedding layer in PyTorch, setting the random seed to `123` for reproducibility purposes:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

The print statement prints the embedding layer's underlying weight matrix:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178,  1.5810,  1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-1.1589,  0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

The weight matrix of the embedding layer contains small, random values. These values are optimized during LLM training as part of the LLM optimization itself. Moreover, we can see that the weight matrix has six rows and three columns. There is one row for each of the six possible tokens in the vocabulary, and there is one column for each of the three embedding dimensions.

Now, let's apply it to a token ID to obtain the embedding vector:

```
print(embedding_layer(torch.tensor([3])))
```

The returned embedding vector is

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

Pytorch