CHAPTER 2
Tokens and Embeddings

Tokens and embeddings are two of the central concepts of using large language models (LLMs). As we've seen in the first chapter, they're not only important to understanding the history of Language AI, but we cannot have a clear sense of how LLMs work, how they're built, and where they will go in the future without a good sense of tokens and embeddings, as we can see in Figure 2-1.

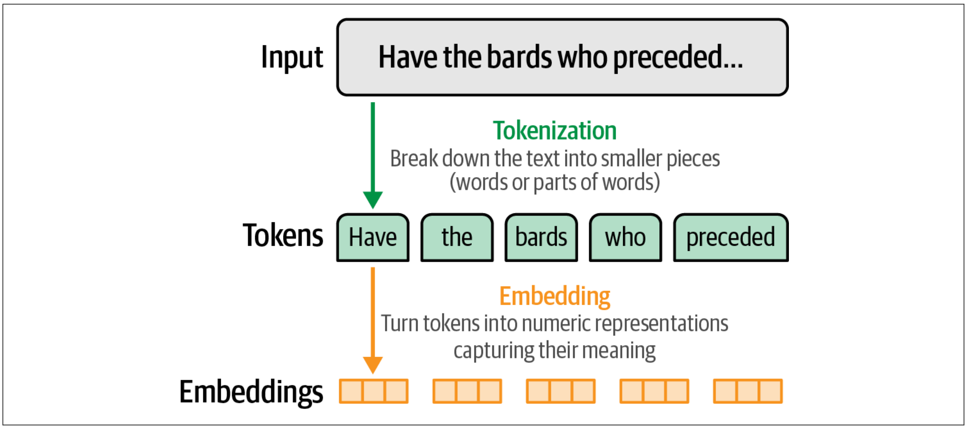

Input
Have the bards who preceded...
Tokenization
Break down the text into smaller pieces
(words or parts of words)
Tokens
Have
the
bards
who
preceded
Embedding
Turn tokens into numeric representations
capturing their meaning
Embeddings

Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

In this chapter, we look more closely at what tokens are and the tokenization methods used to power LLMs. We will then dive into the famous word2vec embedding method that preceded modern-day LLMs and see how it's extending the concept of token embeddings to build commercial recommendation systems that power a lot of the apps you use. Finally, we go from token embeddings into *sentence* or *text* embeddings, where a whole sentence or document can have one vector that represents it—enabling applications like semantic search and topic modeling that we see in Part II of this book.

LLM Tokenization

The way the majority of people interact with language models, at the time of this writing, is through a web playground that presents a chat interface between the user and a language model. You may notice that a model does not produce its output response all at once; it actually generates one token at a time.

But tokens aren't only the output of a model, they're also the way in which the model sees its inputs. A text prompt sent to the model is first broken down into tokens, as we'll now see.

How Tokenizers Prepare the Inputs to the Language Model

Viewed from the outside, generative LLMs take an input prompt and generate a response, as we can see in Figure 2-2.

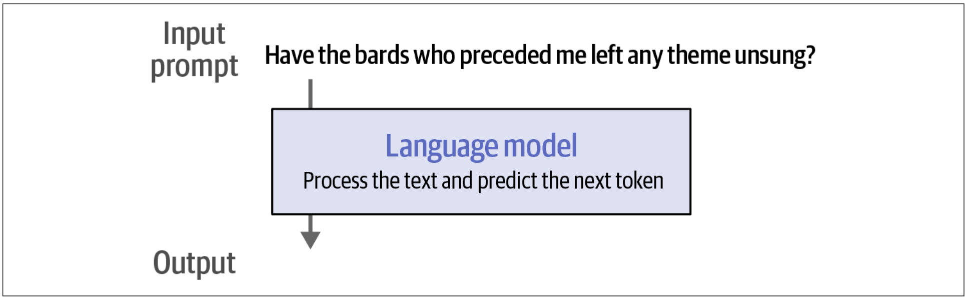
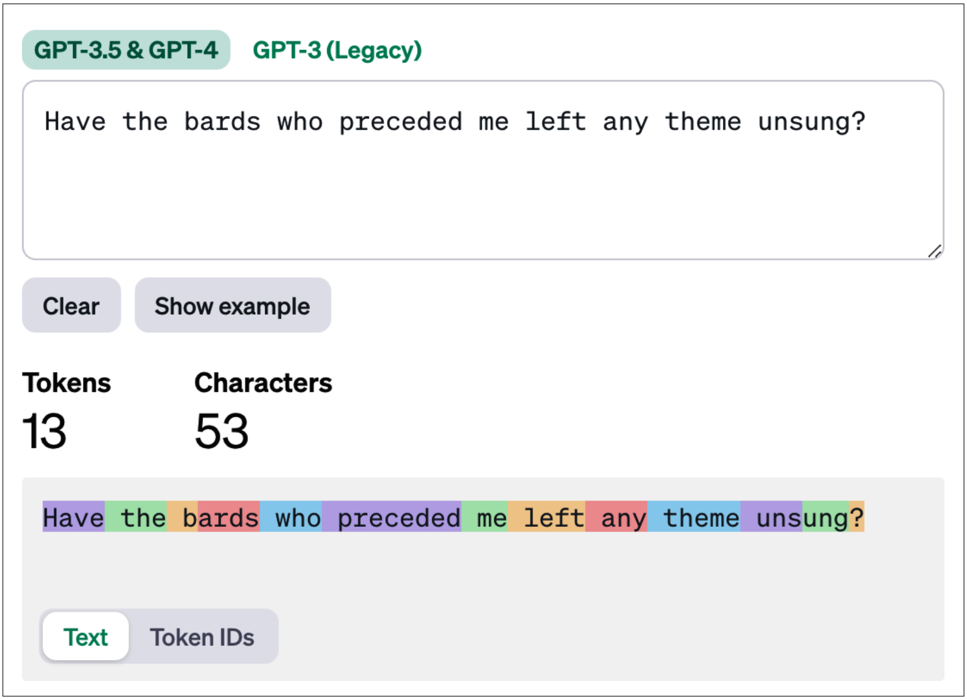

Input prompt
Have the bards who preceded me left any theme unsung?
Language model
Process the text and predict the next token
Output

Figure 2-2. High-level view of a language model and its input prompt.

Before the prompt is presented to the language model, however, it first has to go through a tokenizer that breaks it into pieces. You can find an example showing the tokenizer of GPT-4 on the OpenAI Platform. If we feed it the input text, it shows the output in Figure 2-3, where each token is shown in a different color.



Have the bards who preceded me left any theme unsung?

Tokens 13

Characters 53

Have the bards who preceded me left any theme unsung?

Figure 2-3. A tokenizer breaks down text into words or parts of words before the model processes the text. It does so according to a specific method and training procedure (from https://oreil.ly/ovUWO).

Let's look at a code example and interact with these tokens ourselves. Here we'll be downloading an LLM and seeing how to tokenize the input before generating text with the LLM.

Downloading and Running an LLM

Let's start by loading our model and its tokenizer as we've done in Chapter 1:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")
```

We can then proceed to the actual generation. We first declare our prompt, then tokenize it, then pass those tokens to the model, which generates its output. In this case, we're asking the model to only generate 20 new tokens:

```
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap.
Explain how it happened.<|assistant|>"

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to("cuda")

# Generate the text
generation_output = model.generate(
  input_ids=input_ids,
  max_new_tokens=20
)

# Print the output
print(tokenizer.decode(generation_output[0]))
```

Output:

```
<s> Write an email apologizing to Sarah for the tragic gardening mishap.
Explain how it happened.<|assistant|> Subject: My Sincere Apologies for the
Gardening Mishap

Dear
```

The text in bold is the 20 tokens generated by the model.

Looking at the code, we can see that the model does not in fact receive the text prompt. Instead, the tokenizers processed the input prompt, and returned the information the model needed in the variable `input_ids`, which the model used as its input.

Let's print `input_ids` to see what it holds inside:

```
tensor([[ 1, 14350, 385, 4876, 27746, 5281, 304, 19235, 363, 278, 25305, 293,
16423, 292, 286, 728, 481, 29889, 12027, 7420, 920, 372, 9559, 29889, 32001]],
device='cuda:0')
```

This reveals the inputs that LLMs respond to, a series of integers as shown in Figure 2-4. Each one is the unique ID for a specific token (character, word, or part of a word). These IDs reference a table inside the tokenizer containing all the tokens it knows.

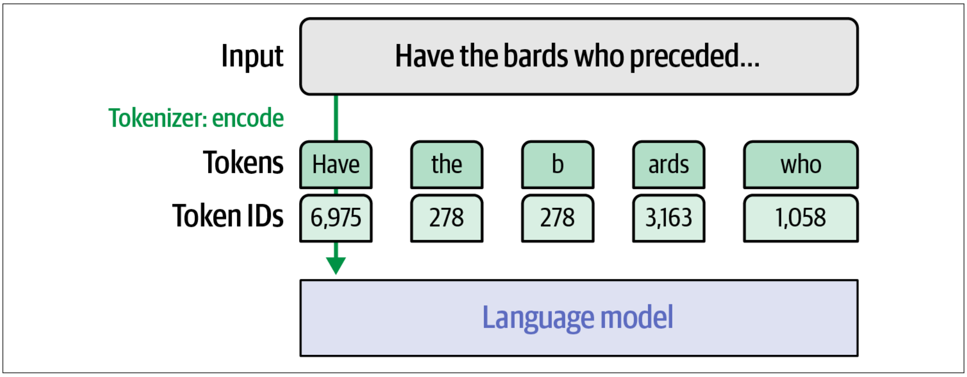


Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

If we want to inspect those IDs, we can use the tokenizer's decode method to translate the IDs back into text that we can read:

```
for id in input_ids[0]:
   print(tokenizer.decode(id))
```

This prints (each token is on a separate line):

```
<s>

Write

an

email

apolog

izing

to

Sarah

for

the

trag

ic

garden

ing

m

ish
```

```
ap

.

Exp

lain

how

it

happened

.

<|assistant|>
```

This is how the tokenizer broke down our input prompt. Notice the following:

- The first token is ID 1 (`<s>`), a special token indicating the beginning of the text.
- Some tokens are complete words (e.g., *Write*, *an*, *email*).
- Some tokens are parts of words (e.g., *apolog*, *izing*, *trag*, *ic*).
- Punctuation characters are their own token.

Notice how the space character does not have its own token. Instead, partial tokens (like "izing" and "ic") have a special hidden character at their beginning that indicates that they're connected with the token that precedes them in the text. Tokens without that special character are assumed to have a space before them.

On the output side, we can also inspect the tokens generated by the model by printing the `generation_output` variable. This shows the input tokens as well as the output tokens (we'll highlight the new tokens in bold):

```
tensor([[ 1, 14350, 385, 4876, 27746, 5281, 304, 19235, 363, 278,

25305, 293, 16423, 292, 286, 728, 481, 29889, 12027, 7420,

920, 372, 9559, 29889, 32001, 3323, 622, 29901, 1619, 317,

3742, 406, 6225, 11763, 363, 278, 19906, 292, 341, 728,

481, 13, 13, 29928, 799]], device='cuda:0')
```

This shows us the model generated the token 3323, `'Sub'`, followed by token 622, `'ject'`. Together they formed the word `'Subject'`. They were then followed by token 29901, which is the colon `':'`...and so on. Just like on the input side, we need the tokenizer on the output side to translate the token ID into the actual text. We do that using the tokenizer's decode method. We can pass it an individual token ID or a list of them:

```
print(tokenizer.decode(3323))
print(tokenizer.decode(622))
print(tokenizer.decode([3323, 622]))
print(tokenizer.decode(29901))
```

This outputs:

```
Sub

ject

Subject

:
```

How Does the Tokenizer Break Down Text?

There are three major factors that dictate how a tokenizer breaks down an input prompt.

First, at model design time, the creator of the model chooses a tokenization method. Popular methods include byte pair encoding (BPE) (widely used by GPT models) and WordPiece (used by BERT). These methods are similar in that they aim to optimize an efficient set of tokens to represent a text dataset, but they arrive at it in different ways.

Second, after choosing the method, we need to make a number of tokenizer design choices like vocabulary size and what special tokens to use. More on this in "Comparing Trained LLM Tokenizers" on page 46.

Third, the tokenizer needs to be trained on a specific dataset to establish the best vocabulary it can use to represent that dataset. Even if we set the same methods and parameters, a tokenizer trained on an English text dataset will be different from another trained on a code dataset or a multilingual text dataset.

In addition to being used to process the input text into a language model, tokenizers are used on the output of the language model to turn the resulting token ID into the output word or token associated with it, as Figure 2-5 shows.

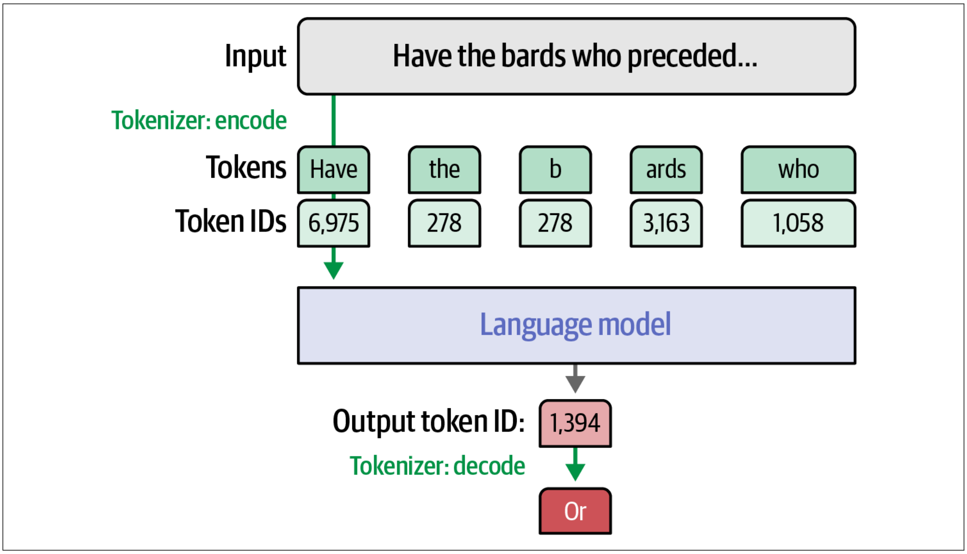


Figure 2-5. Tokenizers are also used to process the output of the model by converting the output token ID into the word or token associated with that ID.

Word Versus Subword Versus Character Versus Byte Tokens

The tokenization scheme we just discussed is called *subword tokenization*. It's the most commonly used tokenization scheme but not the only one. The four notable ways to tokenize are shown in Figure 2-6. Let's go over them:

Word tokens

This approach was common with earlier methods like word2vec but is being used less and less in NLP. Its usefulness, however, led it to be used outside of NLP for use cases such as recommendation systems, as we'll see later in the chapter.

One challenge with word tokenization is that the tokenizer may be unable to deal with new words that enter the dataset after the tokenizer was trained. This also results in a vocabulary that has a lot of tokens with minimal differences between them (e.g., apology, apologize, apologetic, apologist). This latter challenge is resolved by subword tokenization as it has a token for *apolog*, and then suffix tokens (e.g., *-y*, *-ize*, *-etic*, *-ist*) that are common with many other tokens, resulting in a more expressive vocabulary.

Subword tokens

This method contains full and partial words. In addition to the vocabulary expressivity mentioned earlier, another benefit of the approach is its ability to represent new words by breaking down the new token into smaller characters, which tend to be a part of the vocabulary.

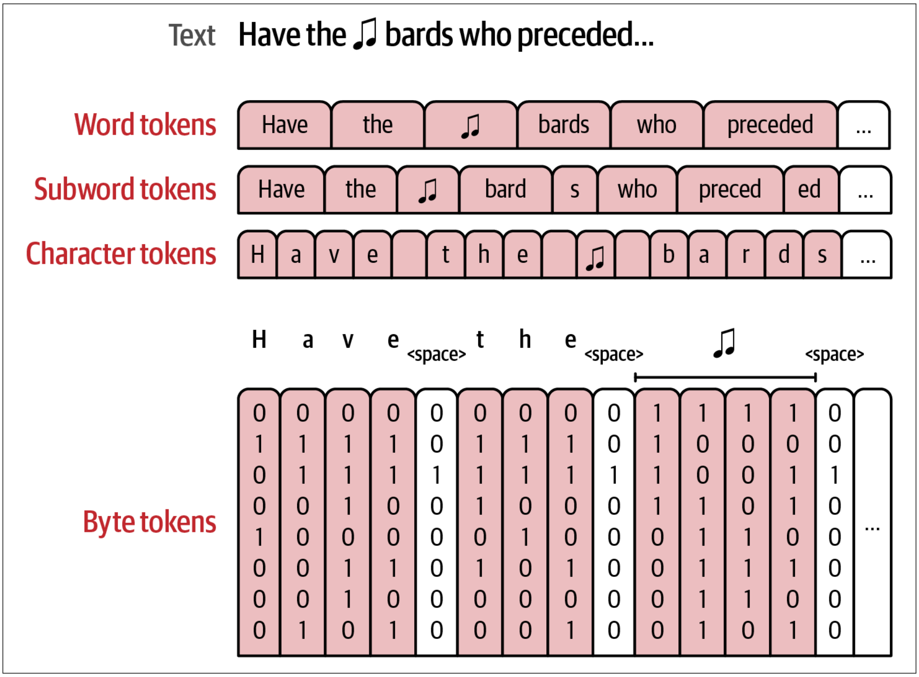

Text
Have the ♫ bards who preceded...
Word tokens
Have | the | ♫ | bards | who | preceded | ...
Subword tokens
Have | the | ♫ | bard | s | who | preced | ed | ...
Character tokens
H | a | v | e | | t | h | e | | ♫ | | b | a | r | d | s | ...
H a v e <space> t h e <space> ♫ <space>
Byte tokens
01001000 | 01100001 | 01110110 | 01100101 | 00100000 | 01110100 | 01101000 | 01100101 | 00100000 | 11110000 | 10011111 | 10001110 | 10110101 | 00100000 | ...

Figure 2-6. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

Character tokens

This is another method that can deal successfully with new words because it has the raw letters to fall back on. While that makes the representation easier to tokenize, it makes the modeling more difficult. Where a model with subword tokenization can represent "play" as one token, a model using character-level tokens needs to model the information to spell out "p-l-a-y" in addition to modeling the rest of the sequence.

Subword tokens present an advantage over character tokens in the ability to fit more text within the limited context length of a Transformer model. So with a model with a context length of 1,024, you may be able to fit about three times as much text using subword tokenization than using character tokens (subword tokens often average three characters per token).

Byte tokens

One additional tokenization method breaks down tokens into the individual bytes that are used to represent unicode characters. Papers like "CANINE: Pre-training an efficient tokenization-free encoder for language representation" outline methods like this, which are also called "tokenization-free encoding." Other works like "ByT5: Towards a token-free future with pre-trained byte-to-byte models" show that this can be a competitive method, especially in multilingual scenarios.

One distinction to highlight here: some subword tokenizers also include bytes as tokens in their vocabulary as the final building block to fall back to when they encounter characters they can't otherwise represent. The GPT-2 and RoBERTa tokenizers do this, for example. This doesn't make them tokenization-free byte-level tokenizers, because they don't use these bytes to represent everything, only a subset, as we'll see in the next section.

If you want to go deeper into tokenizers, they are discussed in more detail in *Designing Large Language Model Applications*.

Comparing Trained LLM Tokenizers

We've pointed out earlier three major factors that dictate the tokens that appear within a tokenizer: the tokenization method, the parameters and special tokens we use to initialize the tokenizer, and the dataset the tokenizer is trained on. Let's compare and contrast a number of actual, trained tokenizers to see how these choices change their behavior. This comparison will show us that newer tokenizers have changed their behavior to improve model performance, and we'll also see how specialized models (like code generation models, for example) often need specialized tokenizers.

We'll use a number of tokenizers to encode the following text:

```
text = """

English and CAPITALIZATION

🎵鸟
show_tokens False None elif == >= else: two tabs:" " Three tabs: "   "

12.0*50=600

"""
```

This will allow us to see how each tokenizer deals with a number of different kinds of tokens:

- Capitalization.
- Languages other than English.
- Emojis.
- Programming code with keywords and whitespaces often used for indentation (in languages like Python for example).

- Numbers and digits.
- Special tokens. These are unique tokens that have a role other than representing text. They include tokens that indicate the beginning of the text, or the end of the text (which is the way the model signals to the system that it has completed this generation), or other functions as we'll see.

Let's go from older to newer tokenizers to see how they tokenize this text and what that might say about the language model. We'll tokenize the text, and then print each token with a color background color using this function:

```
colors_list = [
    '102;194;165', '252;141;98', '141;160;203',
    '231;138;195', '166;216;84', '255;217;47'
]

def show_tokens(sentence, tokenizer_name):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    token_ids = tokenizer(sentence).input_ids
    for idx, t in enumerate(token_ids):
        print(
            f'\x1b[0;30;48;2;{colors_list[idx % len(colors_list)]}m' +
            tokenizer.decode(t) +
            '\x1b[0m',
            end=' '
        )
```

BERT base model (uncased) (2018)

Link to the model on the HuggingFace model hub

Tokenization method: WordPiece, introduced in "Japanese and Korean voice search":

Vocabulary size: 30,522

Special tokens:

`unk_token [UNK]`
An unknown token that the tokenizer has no specific encoding for.

`sep_token [SEP]`
A separator that enables certain tasks that require giving the model two texts (in these cases, the model is called a cross-encoder). One example is reranking, as we'll see in Chapter 8.

`pad_token [PAD]`
A padding token used to pad unused positions in the model's input (as the model expects a certain length of input, its context-size).

`cls_token [CLS]`
: A special classification token for classification tasks, as we'll see in Chapter 4.

`mask_token [MASK]`
: A masking token used to hide tokens during the training process.

Tokenized text:

```
[CLS] english and capital ##ization [UNK] [UNK] show _ token ##s false none
eli ##f = = > = else : two tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SEP]
```

BERT was released in two major flavors: cased (where the capitalization is kept) and uncased (where all capital letters are first turned into small cap letters). With the uncased (and more popular) version of the BERT tokenizer, we notice the following:

- The newline breaks are gone, which makes the model blind to information encoded in newlines (e.g., a chat log when each turn is in a new line).
- All the text is in lowercase.
- The word "capitalization" is encoded as two subtokens: `capital` `##ization`. The `##` characters are used to indicate this token is a partial token connected to the token that precedes it. This is also a method to indicate where the spaces are, as it is assumed tokens without `##` in front have a space before them.
- The emoji and Chinese characters are gone and replaced with the `[UNK]` special token indicating an "unknown token."

BERT base model (cased) (2018)

Link to the model on the HuggingFace model hub

Tokenization method: WordPiece

Vocabulary size: 28,996

Special tokens: Same as the uncased version

Tokenized text:

```
[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##ION [UNK] [UNK] show _ token
##s F ##als ##e None el ##if = = > = else : two ta ##bs : " " Three ta ##bs : " "
12 . 0 * 50 = 600 [SEP]
```

The cased version of the BERT tokenizer differs mainly in including uppercase tokens.

- Notice how "CAPITALIZATION" is now represented as eight tokens: `CA` `##PI` `##TA` `##L` `##I` `##Z` `##AT` `##ION`.

- Both BERT tokenizers wrap the input within a starting `[CLS]` token and a closing `[SEP]` token. `[CLS]` and `[SEP]` are utility tokens used to wrap the input text and they serve their own purposes. `[CLS]` stands for classification as it's a token used at times for sentence classification. `[SEP]` stands for separator, as it's used to separate sentences in some applications that require passing two sentences to a model (For example, in Chapter 8, we will use a `[SEP]` token to separate the text of the query and a candidate result.)

GPT-2 (2019)

Link to the model on the HuggingFace model hub

Tokenization method: Byte pair encoding (BPE), introduced in "Neural machine translation of rare words with subword units".

Vocabulary size: 50,257

Special tokens: `<|endoftext|>`

`English and CAP ITAL IZ ATION`

`� � � � � �`

`show _ t ok ens False None el if == >= else : two tabs :" " Three tabs : " "`

`12 . 0 * 50 = 600`

With the GPT-2 tokenizer, we notice the following:

- The newline breaks are represented in the tokenizer.
- Capitalization is preserved, and the word "CAPITALIZATION" is represented in four tokens.
- The 🎵鸟 characters are now represented by multiple tokens each. While we see these tokens printed as the � character, they actually stand for different tokens. For example, the 🎵 emoji is broken down into the tokens with token IDs 8582, 236, and 113. The tokenizer is successful in reconstructing the original character from these tokens. We can see that by printing `tokenizer.decode([8582, 236, 113])`, which prints out 🎵.
- The two tabs are represented as two tokens (token number 197 in that vocabulary) and the four spaces are represented as three tokens (number 220) with the final space being a part of the token for the closing quote character.

- The two tabs are represented as two tokens (token number 197 in that vocabulary) and the four spaces are represented as three tokens (number 220) with the final space being a part of the token for the closing quote character.

What is the significance of whitespace characters? These are important for models to understand or generate code. A model that uses a single token to represent four consecutive whitespace characters is more tuned to a Python code dataset. While a model can live with representing it as four different tokens, it does make the modeling more difficult as the model needs to keep track of the indentation level, which often leads to worse performance. This is an example of where tokenization choices can help the model improve on a certain task.

Flan-T5 (2022)

Tokenization method: Flan-T5 uses a tokenizer implementation called SentencePiece, introduced in "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing", which supports BPE and the *unigram language model* (described in "Subword regularization: Improving neural network translation models with multiple subword candidates").

Vocabulary size: 32,100

Special tokens:

- `unk_token <unk>`
- `pad_token <pad>`

Tokenized text:

```
English and CA PI TAL IZ ATION <unk> <unk> show _ to ken s Fal s e None e l if = = >
= else : two tab s : " " Three tab s : " " 12. 0 * 50 = 600 </s>
```

The Flan-T5 family of models use the SentencePiece method. We notice the following:

- No newline or whitespace tokens; this would make it challenging for the model to work with code.
- The emoji and Chinese characters are both replaced by the `<unk>` token, making the model completely blind to them.

GPT-4 (2023)

Tokenization method: BPE

Vocabulary size: A little over 100,000

Special tokens:

- `<|endoftext|>`
- Fill in the middle tokens. These three tokens enable the LLM to generate a completion given not only the text before it but also considering the text after it. This method is explained in more detail in the paper "Efficient training of language models to fill in the middle"; its exact details are beyond the scope of this book. These special tokens are:

  — `<|fim_prefix|>`

  — `<|fim_middle|>`

  — `<|fim_suffix|>`

Tokenized text:

`English and CAPITAL IZATION`

`� � � � � �`

`show _tokens False None elif == >= else : two tabs :"     " Three tabs : "     "`

`12 . 0 * 50 = 600`

The GPT-4 tokenizer behaves similarly to its ancestor, the GPT-2 tokenizer. Some differences are:

- The GPT-4 tokenizer represents the four spaces as a single token. In fact, it has a specific token for every sequence of whitespaces up to a list of 83 whitespaces.
- The Python keyword `elif` has its own token in GPT-4. Both this and the previous point stem from the model's focus on code in addition to natural language.
- The GPT-4 tokenizer uses fewer tokens to represent most words. Examples here include "CAPITALIZATION" (two tokens versus four) and "tokens" (one token versus three).
- Refer back to what we said about the GPT-2 tokenizer with regards to the Ł tokens.

StarCoder2 (2024)

StarCoder2 is a 15-billion parameter model focused on generating code described in the paper "StarCoder 2 and the stack v2: The next generation", which continues the work from the original StarCoder described in "StarCoder: May the source be with you!".

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 49,152

Example special tokens:

- `<|endoftext|>`
- Fill in the middle tokens:
  - `<fim_prefix>`
  - `<fim_middle>`
  - `<fim_suffix>`
  - `<fim_pad>`
- When representing code, managing the context is important. One file might make a function call to a function that is defined in a different file. So the model needs some way of being able to identify code that is in different files in the same code repository, while making a distinction between code in different repos. That's why StarCoder2 uses special tokens for the name of the repository and the filename:
  - `<filename>`
  - `<reponame>`
  - `<gh_stars>`

Tokenized text:

```
English and CAPITAL IZATION

� � � � �

show _ tokens False None elif == >= else : two tabs :"  " Three tabs : "   "

1 2 . 0 * 5 0 = 6 0 0
```

This is an encoder that focuses on code generation:

- Similar to GPT-4, it encodes the list of whitespaces as a single token.
- A major difference here to everything we've seen so far is that each digit is assigned its own token (so 600 becomes `6` `0` `0`). The hypothesis here is that this would lead to better representation of numbers and mathematics. In GPT-2, for example, the number 870 is represented as a single token. But 871 is represented as two tokens (`8` and `71`). You can intuitively see how that might be confusing to the model and how it represents numbers.

Galactica

The Galactica model described in "Galactica: A large language model for science" is focused on scientific knowledge and is trained on many scientific papers, reference materials, and knowledge bases. It pays extra attention to tokenization that makes it more sensitive to the nuances of the dataset it's representing. For example, it includes special tokens for citations, reasoning, mathematics, amino acid sequences, and DNA sequences.

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 50,000

Special tokens:

- `<s>`
- `<pad>`
- `</s>`
- `<unk>`
- References: Citations are wrapped within the two special tokens:
  - `[START_REF]`
  - `[END_REF]`
  - One example of usage from the paper is: `Recurrent neural networks, long short-term memory [START_REF]Long Short-Term Memory, Hochreiter[END_REF]`
- Step-by-step reasoning:
  - `<work>` is an interesting token that the model uses for chain-of-thought reasoning.

Tokenized text:

`English` `and` `CAP` `ITAL` `IZATION`

`�` `�` `�` `�` `�` `�` `�`

`show` `_` `tokens` `False` `None` `elif` `==` `>` `=` `else` `:` `two` `t` `abs` `:` `"` ` ` `"` `Three` `t` `abs` `:` `"` `  ` `"`

`1` `2` `.` `0` `*` `5` `0` `=` `6` `0` `0`

The Galactica tokenizer behaves similar to StarCoder2 in that it has code in mind. It also encodes whitespaces in the same way: assigning a single token to sequences of whitespace of different lengths. It differs in that it also does that for tabs, though. So from all the tokenizers we've seen so far, it's the only one that assigns a single token to the string made up of two tabs (`'\t\t'`).

Phi-3 (and Llama 2)

The Phi-3 model we look at in this book reuses the tokenizer of Llama 2 yet adds a number of special tokens.

Tokenization method: Byte pair encoding (BPE)

Vocabulary size: 32,000

Special tokens:

- `<|endoftext|>`
- Chat tokens: As chat LLMs rose to popularity in 2023, the conversational nature of LLMs started to be a leading use case. Tokenizers have been adapted to this direction by the addition of tokens that indicate the turns in a conversation and the roles of each speaker. These special tokens include:
  - — `<|user|>`
  - — `<|assistant|>`
  - — `<|system|>`

We can now recap our tour by looking at all these examples side by side:

BERT base model (uncased)	[CLS] english and capital ##ization [UNK] [UNK] show _ token ##s false none eli ##f = = > = else : two tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SEP]
BERT base model (cased)	[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##ION [UNK] [UNK] show _ token ##s F ##als ##e None el ##if = = > = else : two ta ##bs : " " Three ta ##bs : " " 12 . 0 * 50 = 600 [SEP]
GPT-2	English and CAP ITAL IZ ATION � � � � � � show _ t ok ens False None el if == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600
FLAN-T5	English and CA PI TAL IZ ATION <unk> <unk> show _ to ken s Fal s e None e l if = = > = else : two tab s : " " Three tab s : " " 12. 0 * 50 = 600 </s>
GPT-4	English and CAPITAL IZATION � � � � � � show _tokens False None elif == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600
StarCoder	English and CAPITAL IZATION � � � � � show _ tokens False None elif == >= else : two tabs :" " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0

Galactica	English and CAP ITAL IZATION � � � � � � � show _ tokens False None elif == > = else : two t abs : " " Three t abs : " " 1 2 . 0 * 5 0 = 6 0 0
Phi-3 and Llama 2	<s> English and C AP IT AL IZ ATION � � � � � � � show _ to kens False None elif == >= else : two tabs :" " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0

Tokenizer Properties

The preceding guided tour of trained tokenizers showed a number of ways in which actual tokenizers differ from each other. But what determines their tokenization behavior? There are three major groups of design choices that determine how the tokenizer will break down text: the tokenization method, the initialization parameters, and the domain of the data the tokenizer targets.

Tokenization methods

As we've seen, there are a number of tokenization methods with byte pair encoding (BPE) being the more popular one. Each of these methods outlines an algorithm for how to choose an appropriate set of tokens to represent a dataset. You can find a great overview of all these methods on the Hugging Face page that summarizes tokenizers.

Tokenizer parameters

After choosing a tokenization method, an LLM designer needs to make some decisions about the parameters of the tokenizer. These include:

Vocabulary size

How many tokens to keep in the tokenizer's vocabulary? (30K and 50K are often used as vocabulary size values, but more and more we're seeing larger sizes like 100K.)

Special tokens

What special tokens do we want the model to keep track of? We can add as many of these as we want, especially if we want to build an LLM for special use cases. Common choices include:

- Beginning of text token (e.g., <s>)
- End of text token
- Padding token

- Unknown token
- CLS token
- Masking token

Aside from these, the LLM designer can add tokens that help better model the domain of the problem they're trying to focus on, as we've seen with Galactica's `<work>` and `[START_REF]` tokens.

Capitalization

In languages such as English, how do we want to deal with capitalization? Should we convert everything to lowercase? (Name capitalization often carries useful information, but do we want to waste token vocabulary space on all-caps versions of words?)

The domain of the data

Even if we select the same method and parameters, tokenizer behavior will be different based on the dataset it was trained on (before we even start model training). The tokenization methods mentioned previously work by optimizing the vocabulary to represent a specific dataset. From our guided tour we've seen how that has an impact on datasets like code and multilingual text.

For code, for example, we've seen that a text-focused tokenizer may tokenize the indentation spaces like this (we'll highlight some tokens in color):

```
def add_numbers(a, b):

...."""Add the two numbers `a` and `b`."""

....return a + b
```

This may be suboptimal for a code-focused model. Code-focused models are often improved by making different tokenization choices:

```
def add_numbers(a, b):

...."""Add the two numbers `a` and `b`."""

....return a + b
```

These tokenization choices make the model's job easier and thus its performance has a higher probability of improving.

You can find a more detailed tutorial on training tokenizers in the Tokenizers section of the Hugging Face course and in *Natural Language Processing with Transformers, Revised Edition*.

Token Embeddings

Now that we understand tokenization, we have solved one part of the problem of representing language to a language model. In this sense, language is a sequence of tokens. And if we train a good-enough model on a large-enough set of tokens, it starts to capture the complex patterns that appear in its training dataset:

- If the training data contains a lot of English text, that pattern reveals itself as a model capable of representing and generating the English language.
- If the training data contains factual information (Wikipedia, for example), the model would have the ability to generate some factual information (see the following note).

The next piece of the puzzle is finding the best numerical representation for these tokens that the model can use to calculate and properly model the patterns in the text. These patterns reveal themselves to us as a model's coherence in a specific language, or capability to code, or any of the growing list of capabilities we expect from language models.

As we've seen in Chapter 1, that is what embeddings are. They are the numeric representation space utilized to capture the meanings and patterns in language.

> Oops: Achieving a good threshold of language coherence and better-than-average factual generation, however, starts to present a new problem. Some users start to trust the model's fact generation ability (e.g., at the beginning of 2023 some language models were being dubbed "Google killers"). It didn't take long for advanced users to recognize that generation models alone aren't reliable search engines. This led to the rise of retrieval-augmented generation (RAG), which combines search and LLMs. We cover RAG in more detail in Chapter 8.

A Language Model Holds Embeddings for the Vocabulary of Its Tokenizer

After a tokenizer is initialized and trained, it is then used in the training process of its associated language model. This is why a pretrained language model is linked with its tokenizer and can't use a different tokenizer without training.

The language model holds an embedding vector for each token in the tokenizer's vocabulary, as we can see in Figure 2-7. When we download a pretrained language model, a portion of the model is this embeddings matrix holding all of these vectors.

Before the beginning of the training process, these vectors are randomly initialized like the rest of the model's weights, but the training process assigns them the values that enable the useful behavior they're trained to perform.

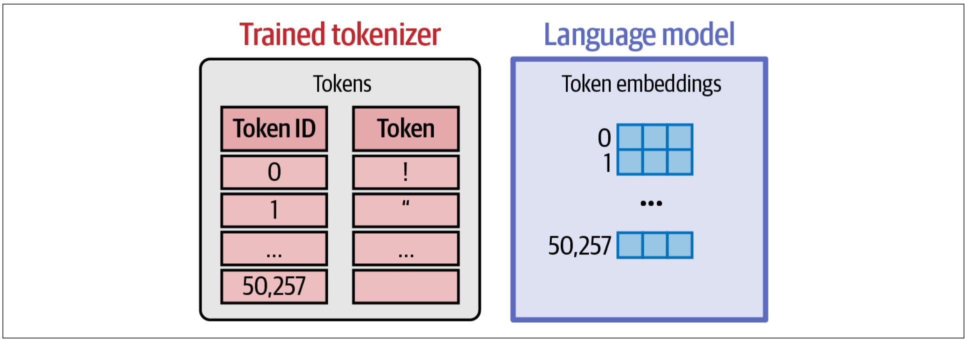


Figure 2-7. A language model holds an embedding vector associated with each token in its tokenizer.

Creating Contextualized Word Embeddings with Language Models

Now that we've covered token embeddings as the input to a language model, let's look at how language models can *create* better token embeddings. This is one of the primary ways to use language models for text representation. This empowers applications like named-entity recognition or extractive text summarization (which summarizes a long text by highlighting the most important parts of it, instead of generating new text as a summary).

Instead of representing each token or word with a static vector, language models create contextualized word embeddings (shown in Figure 2-8) that represent a word with a different token based on its context. These vectors can then be used by other systems for a variety of tasks. In addition to the text applications we mentioned in the previous paragraph, these contextualized vectors, for example, are what powers AI image generation systems like DALL·E, Midjourney, and Stable Diffusion, for example.

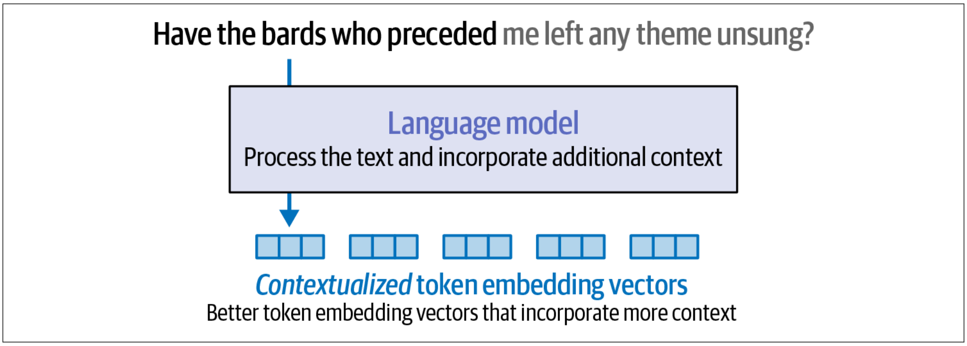

Have the bards who preceded me left any theme unsung?
Language model
Process the text and incorporate additional context
Contextualized token embedding vectors
Better token embedding vectors that incorporate more context

Figure 2-8. Language models produce contextualized token embeddings that improve on raw, static token embeddings.

Let's look at how we can generate contextualized word embeddings; the majority of this code should be familiar to you by now:

```
from transformers import AutoModel, AutoTokenizer

# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-base")

# Load a language model
model = AutoModel.from_pretrained("microsoft/deberta-v3-xsmall")

# Tokenize the sentence
tokens = tokenizer('Hello world', return_tensors='pt')

# Process the tokens
output = model(**tokens)[0]
```

The model we're using here is called DeBERTa v3, which at the time of writing is one of the best-performing language models for token embeddings while being small and highly efficient. It is described in the paper "DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training gradient-disentangled embedding sharing".

This code downloads a pretrained tokenizer and model, then uses them to process the string "Hello world". The output of the model is then saved in the output variable. Let's inspect that variable by first printing its dimensions (we expect it to be a multidimensional array):

```
output.shape
```

This prints out:

```
torch.Size([1, 4, 384])
```

Skipping the first dimension, we can read this as four tokens, each one embedded in a vector of 384 values. The first dimension is the batch dimension used in cases (like training) when we want to send multiple input sentences to the model at the same time (they're processed at the same time, which speeds up the process).

But what are these four vectors? Did the tokenizer break the two words into four tokens, or is something else happening here? We can use what we've learned about tokenizers to inspect them:

```
for token in tokens['input_ids'][0]:
    print(tokenizer.decode(token))
```

This prints out:

```
[CLS]
Hello
world
[SEP]
```

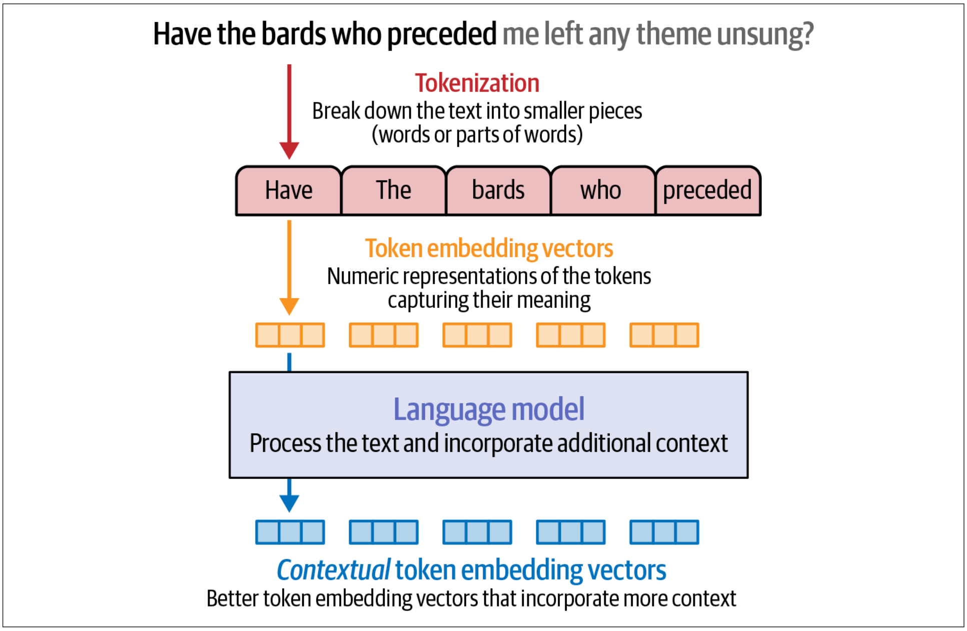
This particular tokenizer and model operate by adding the `[CLS]` and `[SEP]` tokens to the beginning and end of a string.

Our language model has now processed the text input. The result of its output is the following:

```
tensor([[
[-3.3060, -0.0507, -0.1098, ..., -0.1704, -0.1618, 0.6932],
[ 0.8918, 0.0740, -0.1583, ..., 0.1869, 1.4760, 0.0751],
[ 0.0871, 0.6364, -0.3050, ..., 0.4729, -0.1829, 1.0157],
[-3.1624, -0.1436, -0.0941, ..., -0.0290, -0.1265, 0.7954]
]], grad_fn=<NativeLayerNormBackward0>)
```

This is the raw output of a language model. The applications of large language models build on top of outputs like this.

We recap the input tokenization and resulting outputs of a language model in Figure 2-9. Technically, the switch from token IDs into raw embeddings is the first step that occurs inside a language model.



Have the bards who preceded me left any theme unsung?
Tokenization
Break down the text into smaller pieces (words or parts of words)
Have
The
bards
who
preceded
Token embedding vectors
Numeric representations of the tokens capturing their meaning
Language model
Process the text and incorporate additional context
Contextual token embedding vectors
Better token embedding vectors that incorporate more context

Figure 2-9. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

A visual like this is essential for the next chapter when we start to look at how Transformer-based LLMs work.

Text Embeddings (for Sentences and Whole Documents)

While token embeddings are key to how LLMs operate, a number of LLM applications require operating on entire sentences, paragraphs, or even text documents. This has led to special language models that produce text embeddings—a single vector that represents a piece of text longer than just one token.

We can think of text embedding models as taking a piece of text and ultimately producing a single vector that represents that text and captures its meaning in some useful form. Figure 2-10 shows that process.

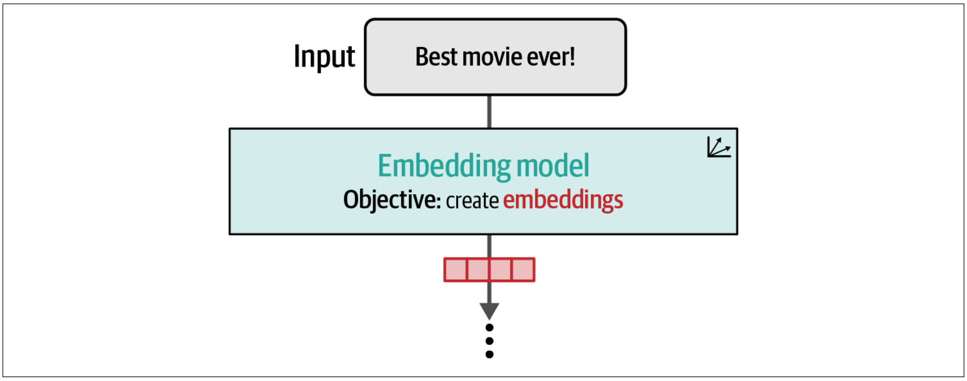


Figure 2-10. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

There are multiple ways of producing a *text* embedding vector. One of the most common ways is to average the values of all the *token* embeddings produced by the model. Yet high-quality text embedding models tend to be trained specifically for text embedding tasks.

We can produce text embeddings with `sentence-transformers`, a popular package for leveraging pretrained embedding models.[1] The package, like `transformers` in the previous chapter, can be used to load publicly available models. To illustrate creating embeddings, we use the all-mpnet-base-v2 model. Note that in Chapter 4, we will further explore how you can choose an embedding model for your task.

```
from sentence_transformers import SentenceTransformer

# Load model
model = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")

# Convert text to text embeddings
vector = model.encode("Best movie ever!")
```

The number of values, or the dimensions, of the embedding vector depends on the underlying embedding model. Let's explore that for our model:

```
vector.shape
```

```
(768,)
```

[1] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence embeddings using Siamese BERT-networks." *arXiv preprint arXiv:1908.10084* (2019).

This sentence is now encoded in this one vector with a dimension of 768 numerical values. In Part II of this book, once we start looking at applications, we'll start to see the immense usefulness of these text embeddings vectors in powering everything from categorization to semantic search to RAG.

Word Embeddings Beyond LLMs

Embeddings are useful even outside of text and language generation. Embeddings, or assigning meaningful vector representations to objects, turns out to be useful in many domains, including recommender engines and robotics. In this section, we'll look at how to use pretrained word2vec embeddings and touch on how the method creates word embeddings. Seeing how word2vec is trained will prime you to learn about contrastive training in Chapter 10. Then in the following section, we'll see how those embeddings can be used for recommendation systems.

Using pretrained Word Embeddings

Let's look at how we can download pretrained word embeddings (like word2vec or GloVe) using the Gensim library:

```
import gensim.downloader as api

# Download embeddings (66MB, glove, trained on wikipedia, vector size: 50)
# Other options include "word2vec-google-news-300"
# More options at https://github.com/RaRe-Technologies/gensim-data
model = api.load("glove-wiki-gigaword-50")
```

Here, we've downloaded the embeddings of a large number of words trained on Wikipedia. We can then explore the embedding space by seeing the nearest neighbors of a specific word, "king" for example:

```
model.most_similar([model['king']], topn=11)
```

This outputs:

```
[('king', 1.0000001192092896),
('prince', 0.8236179351806641),
('queen', 0.7839043140411377),
('ii', 0.7746230363845825),
('emperor', 0.7736247777938843),
('son', 0.766719400882721),
('uncle', 0.7627150416374207),
('kingdom', 0.7542161345481873),
('throne', 0.7539914846420288),
('brother', 0.7492411136627197),
('ruler', 0.7434253692626953)]
```

The Word2vec Algorithm and Contrastive Training

The word2vec algorithm described in the paper "Efficient estimation of word representations in vector space" is described in detail in The Illustrated Word2vec. The central ideas are condensed here as we build on them when discussing one method for creating embeddings for recommendation engines in the following section.

Just like LLMs, word2vec is trained on examples generated from text. Let's say, for example, we have the text "Thou shalt not make a machine in the likeness of a human mind" from the *Dune* novels by Frank Herbert. The algorithm uses a sliding window to generate training examples. We can, for example, have a window size two, meaning that we consider two neighbors on each side of a central word.

The embeddings are generated from a classification task. This task is used to train a neural network to predict if words commonly appear in the same context or not (*context* here means in many sentences in the training dataset we're modeling). We can think of this as a neural network that takes two words and outputs 1 if they tend to appear in the same context, and 0 if they do not.

In the first position for the sliding window, we can generate four training examples, as we can see in Figure 2-11.

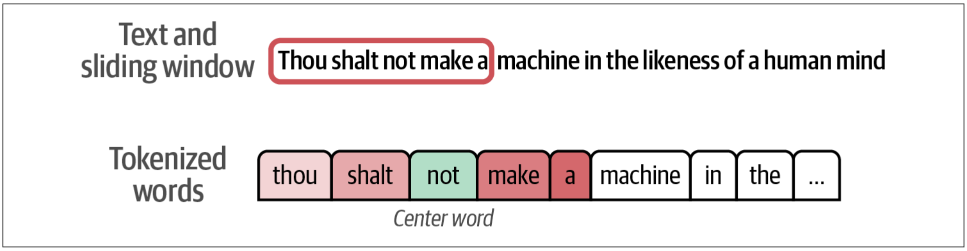
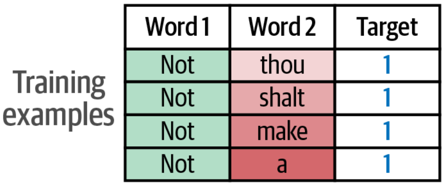


Figure 2-11. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

In each of the produced training examples, the word in the center is used as one input, and each of its neighbors is a distinct second input in each training example. We expect the final trained model to be able to classify this neighbor relationship and output 1 if the two input words it receives are indeed neighbors. These training examples are visualized in Figure 2-12.

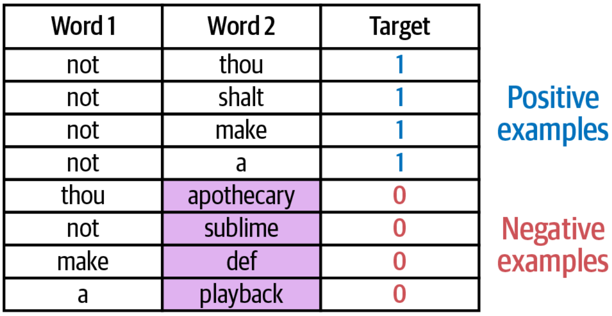


Training
examples

Word 1	Word 2	Target
Not	thou	1
Not	shalt	1
Not	make	1
Not	a	1

Figure 2-12. Each generated training example shows a pair of neighboring words.

If, however, we have a dataset of only a target value of 1, then a model can cheat and ace it by outputting 1 all the time. To get around this, we need to enrich our training dataset with examples of words that are not typically neighbors. These are called negative examples and are shown in Figure 2-13.



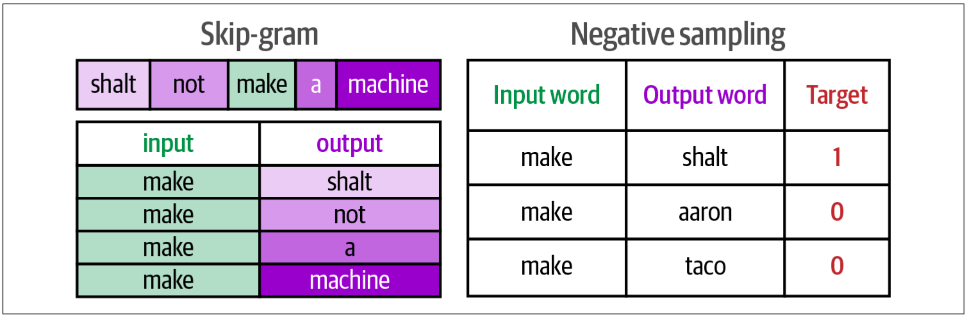
Word 1	Word 2	Target
not	thou	1
not	shalt	1
not	make	1
not	a	1
thou	apothecary	0
not	sublime	0
make	def	0
a	playback	0

Positive
examples
Negative
examples

Figure 2-13. We need to present our models with negative examples: words that are not usually neighbors. A better model is able to better distinguish between the positive and negative examples.

It turns out that we don't have to be too scientific in how we choose the negative examples. A lot of useful models result from the simple ability to detect positive examples from randomly generated examples (inspired by an important idea called *noise-contrastive estimation* and described in "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models"). So in this case, we get random words and add them to the dataset and indicate that they are not neighbors (and thus the model should output 0 when it sees them).

With this, we've seen two of the main concepts of word2vec (Figure 2-14): skip-gram, the method of selecting neighboring words, and negative sampling, adding negative examples by random sampling from the dataset.



shalt	not	make	a	machine

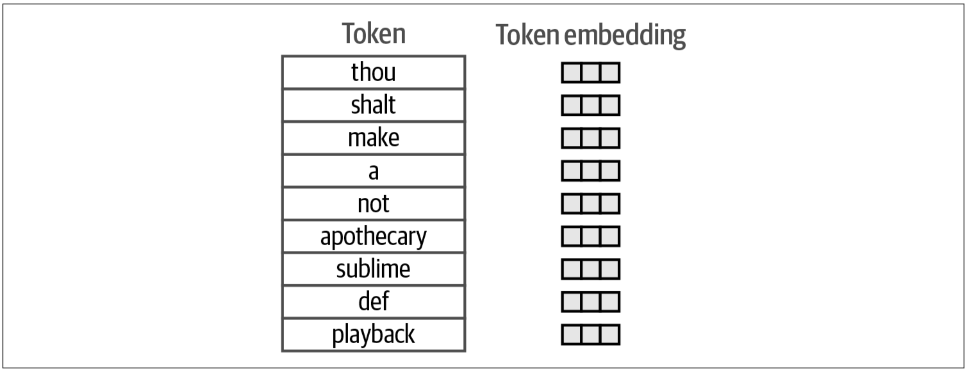
input	output
make	shalt
make	not
make	a
make	machine

Input word	Output word	Target
make	shalt	1
make	aaron	0
make	taco	0

Figure 2-14. Skip-gram and negative sampling are two of the main ideas behind the word2vec algorithm and are useful in many other problems that can be formulated as token sequence problems.

We can generate millions and even billions of training examples like this from running text. Before proceeding to train a neural network on this dataset, we need to make a couple of tokenization decisions, which, just like we've seen with LLM tokenizers, include how to deal with capitalization and punctuation and how many tokens we want in our vocabulary.

We then create an embedding vector for each token, and randomly initialize them, as can be seen in Figure 2-15. In practice, this is a matrix of dimensions `vocab_size x embedding_dimensions`.



Token	Token embedding
thou	□□□
shalt	□□□
make	□□□
a	□□□
not	□□□
apothecary	□□□
sublime	□□□
def	□□□
playback	□□□

Figure 2-15. A vocabulary of words and their starting, random, uninitialized embedding vectors.

A model is then trained on each example to take in two embedding vectors and predict if they're related or not. We can see what this looks like in Figure 2-16.

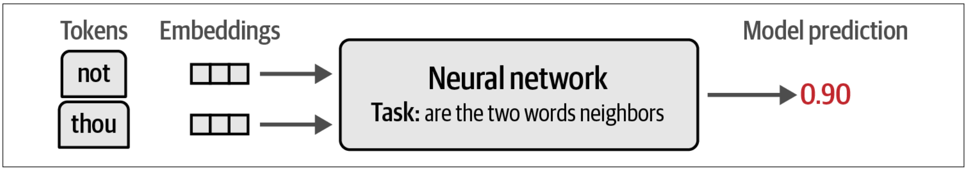


Figure 2-16. A neural network is trained to predict if two words are neighbors. It updates the embeddings in the training process to produce the final, trained embeddings.

Based on whether its prediction was correct or not, the typical machine learning training step updates the embeddings so that the next time the model is presented with those two vectors, it has a better chance of being more correct. And by the end of the training process, we have better embeddings for all the tokens in our vocabulary.

This idea of a model that takes two vectors and predicts if they have a certain relation is one of the most powerful ideas in machine learning, and time after time has proven to work very well with language models. This is why we're dedicating Chapter 10 to this concept and how it optimizes language models for specific tasks (like sentence embeddings and retrieval).

The same idea is also central to bridging modalities like text and images, which is key to AI image generation models, as we'll see in Chapter 9 on multimodal models. In that formulation, a model is presented with an image and a caption, and it should predict whether that caption describes the image or not.

Embeddings for Recommendation Systems

As we've mentioned, the concept of embeddings is useful in so many other domains. In industry, it's widely used for recommendation systems, for example.

Recommending Songs by Embeddings

In this section we'll use the word2vec algorithm to embed songs using human-made music playlists. Imagine if we treated each song as we would a word or token, and we treated each playlist like a sentence. These embeddings can then be used to recommend similar songs that often appear together in playlists.

The dataset we'll use was collected by Shuo Chen from Cornell University. It contains playlists from hundreds of radio stations around the US. Figure 2-17 demonstrates this dataset.

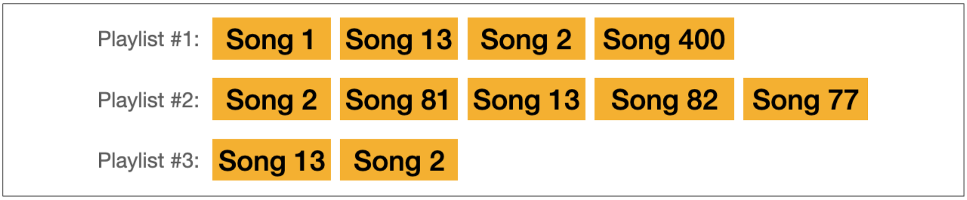


Figure 2-17. For song embeddings that capture song similarity we'll use a dataset made up of a collection of playlists, each containing a list of songs.

Let's demonstrate the end product before we look at how it's built. So let's give it a few songs and see what it recommends in response.

Let's start by giving it Michael Jackson's "Billie Jean," the song with ID 3822:

```
# We will define and explore this function in detail below
print_recommendations(3822)
```

id	Title	artist
4181	Kiss	Prince & The Revolution
12749	Wanna Be Startin' Somethin'	Michael Jackson
1506	The Way You Make Me Feel	Michael Jackson
3396	Holiday	Madonna
500	Don't Stop 'Til You Get Enough	Michael Jackson

That looks reasonable. Madonna, Prince, and other Michael Jackson songs are the nearest neighbors.

Let's step away from pop and into rap, and see the neighbors of 2Pac's "California Love":

```
print_recommendations(842)
```

id	Title	artist
413	If I Ruled the World (Imagine That) (w\/ Lauryn Hill)	Nas
196	I'll Be Missing You	Puff Daddy & The Family
330	Hate It or Love It (w\/ 50 Cent)	The Game
211	Hypnotize	The Notorious B.I.G.
5788	Drop It Like It's Hot (w\/ Pharrell)	Snoop Dogg

Another quite reasonable list! Now that we know it works, let's see how to build such a system.

Training a Song Embedding Model

We'll start by loading the dataset containing the song playlists as well as each song's metadata, such as its title and artist:

```
import pandas as pd
from urllib import request

# Get the playlist dataset file
data = request.urlopen('https://storage.googleapis.com/maps-premium/data
set/yes_complete/train.txt')

# Parse the playlist dataset file. Skip the first two lines as
# they only contain metadata
lines = data.read().decode("utf-8").split('\n')[2:]

# Remove playlists with only one song
playlists = [s.rstrip().split() for s in lines if len(s.split()) > 1]

# Load song metadata
songs_file = request.urlopen('https://storage.googleapis.com/maps-premium/data
set/yes_complete/song_hash.txt')
songs_file = songs_file.read().decode("utf-8").split('\n')
songs = [s.rstrip().split('\t') for s in songs_file]
songs_df = pd.DataFrame(data=songs, columns = ['id', 'title', 'artist'])
songs_df = songs_df.set_index('id')
```

Now that we've saved them, let's inspect the `playlists` list. Each element inside it is a playlist containing a list of song IDs:

```
print( 'Playlist #1:\n ', playlists[0], '\n')
print( 'Playlist #2:\n ', playlists[1])
```

```
Playlist #1: ['0', '1', '2', '3', '4', '5', ..., '43']
Playlist #2: ['78', '79', '80', '3', '62', ..., '210']
```

Let's train the model:

```
from gensim.models import Word2Vec

# Train our Word2Vec model
model = Word2Vec(
    playlists, vector_size=32, window=20, negative=50, min_count=1, workers=4
)
```

That takes a minute or two to train and results in embeddings being calculated for each song that we have. Now we can use those embeddings to find similar songs exactly as we did earlier with words:

```
song_id = 2172

# Ask the model for songs similar to song #2172
model.wv.most_similar(positive=str(song_id))
```

This outputs:

```
[('2976', 0.9977465271949768),
 ('3167', 0.9977430701255798),
 ('3094', 0.9975950717926025),
 ('2640', 0.9966474175453186),
 ('2849', 0.9963167905807495)]
```

That is the list of the songs whose embeddings are most similar to song 2172.

In this case, the song is:

```
print(songs_df.iloc[2172])
```

```
title Fade To Black
artist Metallica
Name: 2172 , dtype: object
```

This results in recommendations that are all in the same heavy metal and hard rock genre:

```
import numpy as np

def print_recommendations(song_id):
    similar_songs = np.array(
        model.wv.most_similar(positive=str(song_id),topn=5)
    )[:,0]
    return  songs_df.iloc[similar_songs]

# Extract recommendations
print_recommendations(2172)
```

id	Title	artist
11473	Little Guitars	Van Halen
3167	Unchained	Van Halen
5586	The Last in Line	Dio
5634	Mr. Brownstone	Guns N' Roses
3094	Breaking the Law	Judas Priest

Summary

In this chapter, we have covered LLM tokens, tokenizers, and useful approaches to using token embeddings. This prepares us to start looking closer at language models in the next chapter, and also opens the door to learn about how embeddings are used beyond language models.

We explored how tokenizers are the first step in processing input to an LLM, transforming raw textual input into token IDs. Common tokenization schemes include breaking text down into words, subword tokens, characters, or bytes, depending on the specific requirements of a given application.

A tour of real-world pretrained tokenizers (from BERT to GPT-2, GPT-4, and other models) showed us areas where some tokenizers are better (e.g., preserving information like capitalization, newlines, or tokens in other languages) and other areas where tokenizers are just different from each other (e.g., how they break down certain words).

Three of the major tokenizer design decisions are the tokenizer algorithm (e.g., BPE, WordPiece, SentencePiece), tokenization parameters (including vocabulary size, special tokens, capitalization, treatment of capitalization and different languages), and the dataset the tokenizer is trained on.

Language models are also creators of high-quality contextualized token embeddings that improve on raw static embeddings. Those contextualized token embeddings are what's used for tasks including named-entity recognition (NER), extractive text summarization, and text classification. In addition to producing token embeddings, language models can produce text embeddings that cover entire sentences or even documents. This empowers plenty of applications that will be shown in Part II of this book covering language model applications

Before LLMs, word embedding methods like word2vec, GloVe, and fastText were popular. In language processing, this has largely been replaced with contextualized word embeddings produced by language models. The word2vec algorithm relies on two main ideas: skip-gram and negative sampling. It also uses contrastive training similar to the type we'll see in Chapter 10.

Embeddings are useful for creating and improving recommender systems as we discussed in the music recommender we built from curated song playlists.

In the next chapter, we will take a deep dive into the process after tokenization: how does an LLM process these tokens and generate text? We will look at some of the main intuitions of how LLMs that use the Transformer architecture work.