Chapter 2 - Tokens and Embeddings

Tokens and embeddings are two of the central concepts of using LLMs, 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.

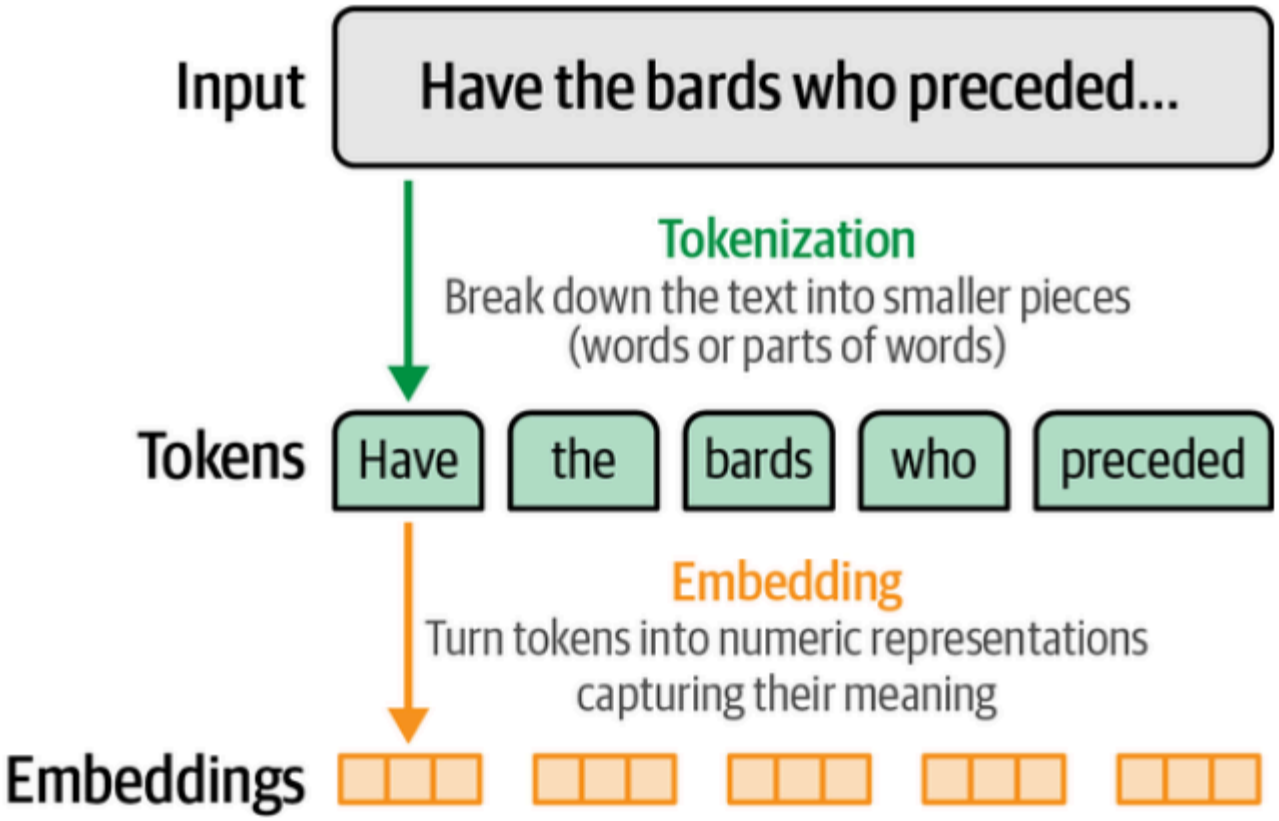


Figure 2-1. Language models deal with text in small chunks called tokens. For the language model to compute language, it needs to turn tokens into numeric representations called embeddings.

LLM Tokenization

The way majority of people interact with language models is through the web where a chat interface connects the user to a language model. This leaves many with the assumption that the language model takes and processes their input directly as they are.

In the rest of this chapter, we will see how text input is processed to allow the language model to understand it.

Let's start by loading our model and its tokenizer:

```
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")
```

How Tokenizers Prepare the Inputs to the Language Model

Before a prompt is presented to the language model, it first has to go through a tokenizer that breaks it into pieces.

Below, you can see how the tokenizer of GPT-4 from OpenAI works.

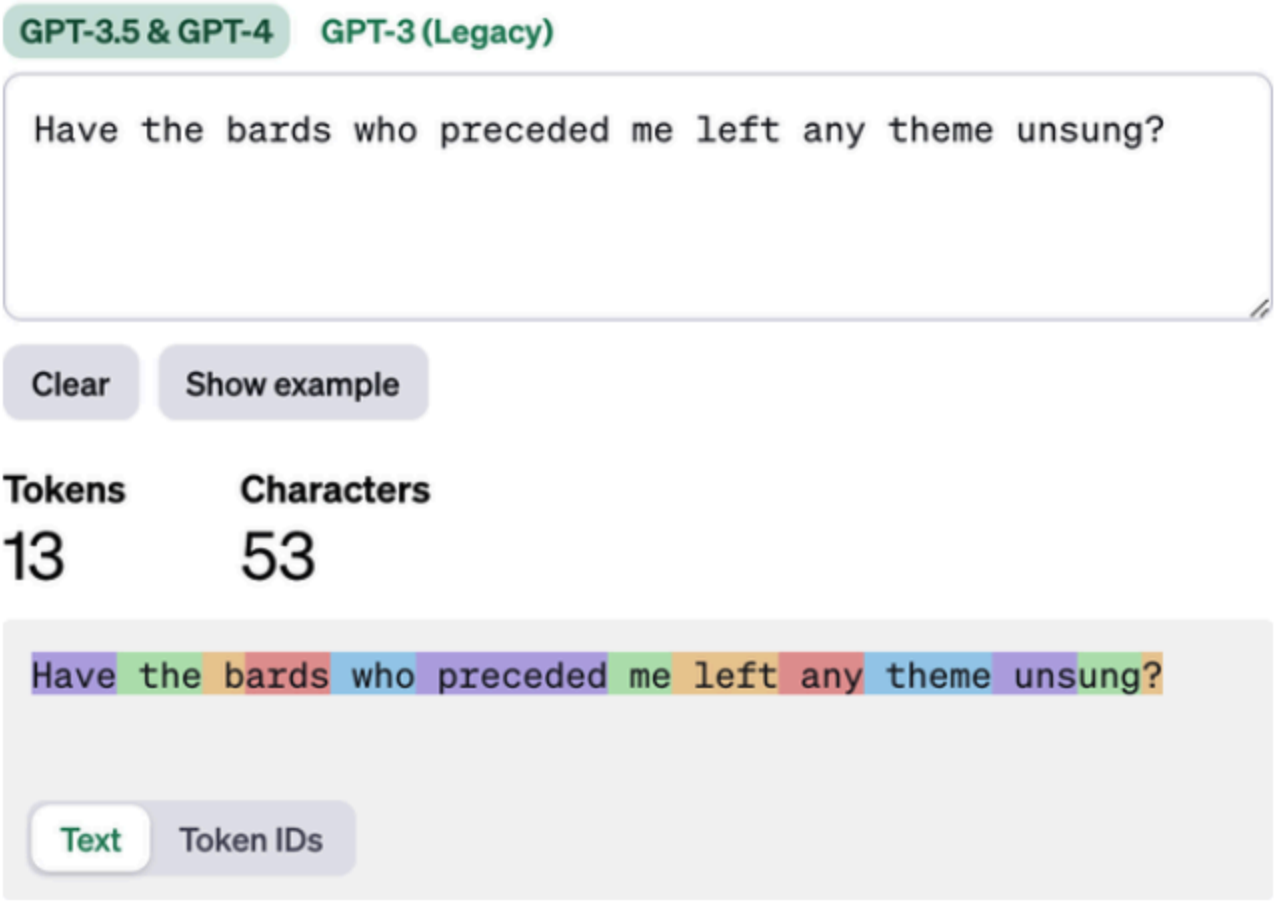


Figure 2-3. A tokenizer breaks down text into words or parts of words before the model processes the text. It does so according to a specific method and training procedure (from https://oreil.ly/ovUWO).

If you run the code below,

```
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]))
```

Looking at the code, you 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.

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 has 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 input the LLMs respond to, a series of integers as shown above.

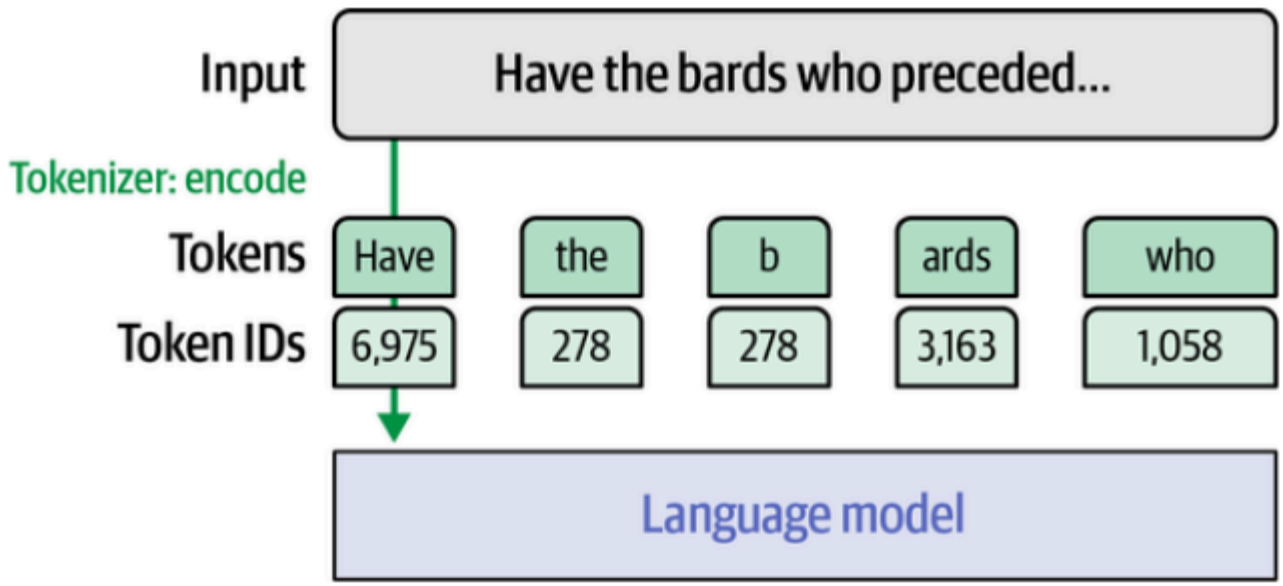


Figure 2-4. A tokenizer processes the input prompt and prepares the actual input into the language model: a list of token IDs. The specific token IDs in the figure are just demonstrative.

To inspect those `ids` in `input_ids`, we can run:

```
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|>
```

The output shown above demonstrates how the tokenizer processed the input prompt. There are important observations:

- The first **token ID 1** (< `s` >) is a special token indicating its starting point (every tokenizer has a different indicator)
- While some tokens are complete words (e.g., Write, an, email),
- Some are parts of words (e.g., `apolog` , `izing` , `trag` , `ic`)
- Punctuations are their own tokens

Just like for the input, we can also see the tokens of the output by printing the `generate_output` variable:

```
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')
```

- Focusing on the bold token ids, from `3323`, these are the output tokens.
- Starting from `Sub`, `ject`, `->` `3323`, `622`

If you try to print these tokens separately:

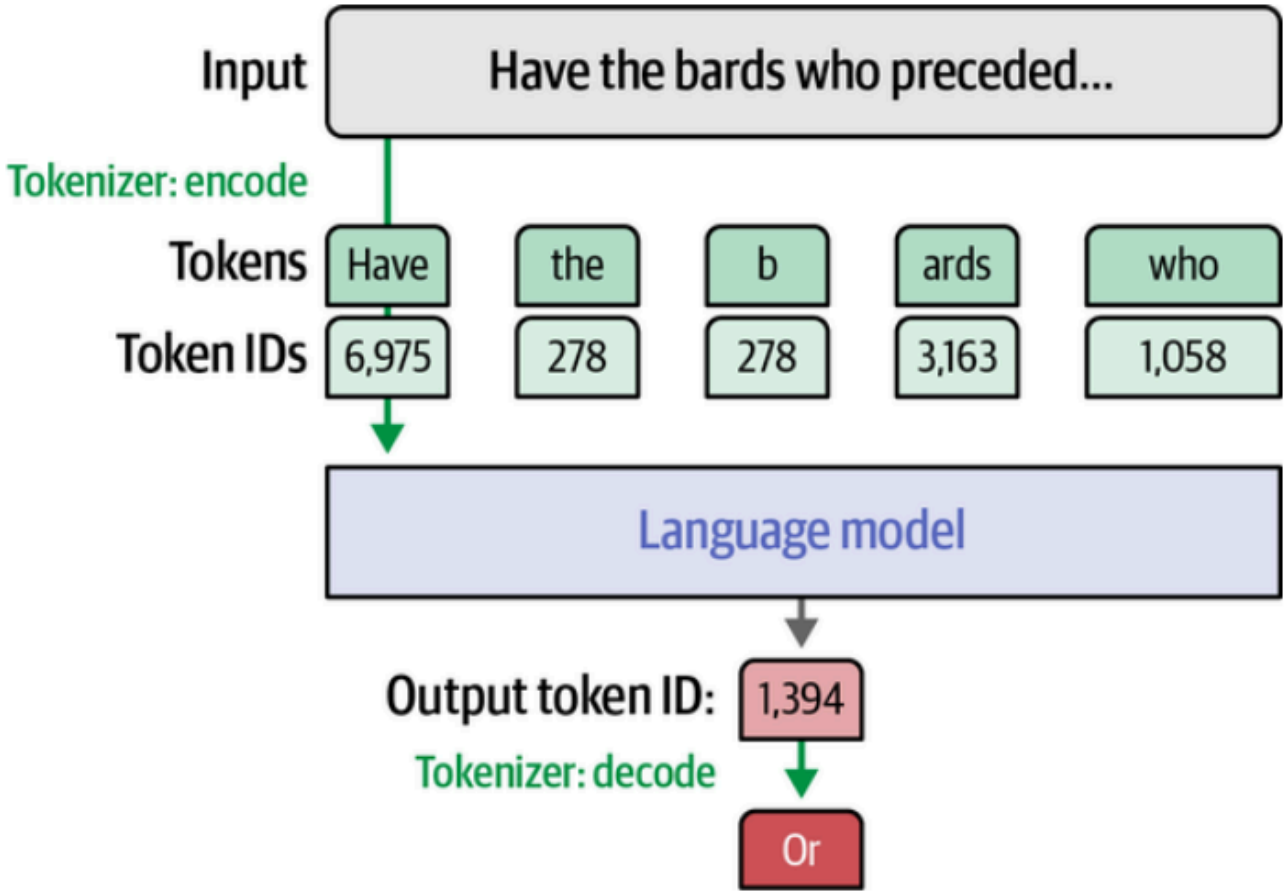
```
print(tokenizer.decode(3323))
print(tokenizer.decode(622))
print(tokenizer.decode([3323, 622]))
print(tokenizer.decode(29901))
```

You'll get:

```
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:
>
> 1. The **tokenization method***: Popular methods include ***byte pair encoding*** (BPE) (widely used by GPT models), and ***WordPiece*** (used by BERT).
> 2. **Design Choices**: Design choices like vocabulary size, and what special tokens to use.
> 3. **Training Data**: The tokenizer needs to be trained on a specific dataset to establish the best vocabulary it can use to represent that dataset.



Word Versus Subword Versus Character Versus Byte

1. ***Word tokens*** This approach was common with the earlier methods like **word2vec** but is being used less and less in NLP. However, one issue with word tokenization is that it may be unable to deal with new words that enter the dataset after the tokenizer was trained.

   The token of unknown words would be `[UNK]`. This would not translate the real meaning of the new word to the model and therefore the model would not be able to generate the correct or appropriate response.

   This also results in a vocabulary that has a lot of tokens with minimal differences between them (e.g., `apology`, `apologize`, `apologetic`, `apologist`).

   Although this has been resolved by subword tokenization as it has taken `apolog` as a token and the rest of the word as suffix tokens (e.g., `-ize`, `-etic`, `-ist`).
2. ***Subword tokens***: As we've seen in how GPT-4 tokenizes its input prompts, this method contains full and partial words.

   In addition to richer meaning, another benefit is being able to tokenize ***most*** new words by breaking them down to subwords.

   For example, if the tokenizer learned `critic` and `ize`, and `apolog` and `etic`, it would be able to handle a new word such as `apologize`, by breaking it down to `apolog` and `ize`. This is how WordPiece (used by GPT-4) works.

3. ***Character tokens***: This is another method that can deal with new words successfully, as it can deal with raw letters. While this tokenization method makes representation/tokenization straightforward, it makes the modeling more difficult.

   Another disadvantage is that it requires a higher context length for the same sentence length, compared to WordPiece. The word `play` would equal 1 token for a subword tokenizer while it would equal 5 for a character tokenizer.
4. ***Byte tokens***: This method represents text as raw bytes (numbers 0-255) instead of characters or subwords. It can handle ANY text - any language, emoji, or symbol - without needing special tokens or vocabulary, making it perfect for multilingual models.

   However, it creates really long sequences since each character becomes multiple bytes. For example, the word `hello` would be 1 token for a subword tokenizer but 5 bytes for byte-level. Also, non-English text explodes in length - `你` (Chinese character) would be 1 character token but 3 bytes, making Chinese text 3x longer in general.

   A hybrid: **Byte fallback (GPT-2, RoBERTa)**, These tokenizers primarily use subwords but fall back to bytes for unknown characters. For instance, `hello` stays as one subword token, but if they encounter an unusual character like `ź`, only that character converts to bytes `[197, 186]`. This gives the best of both worlds - efficient tokenization for common text, and universal coverage for rare characters.

The image below shows the different tokenization types and how they work:

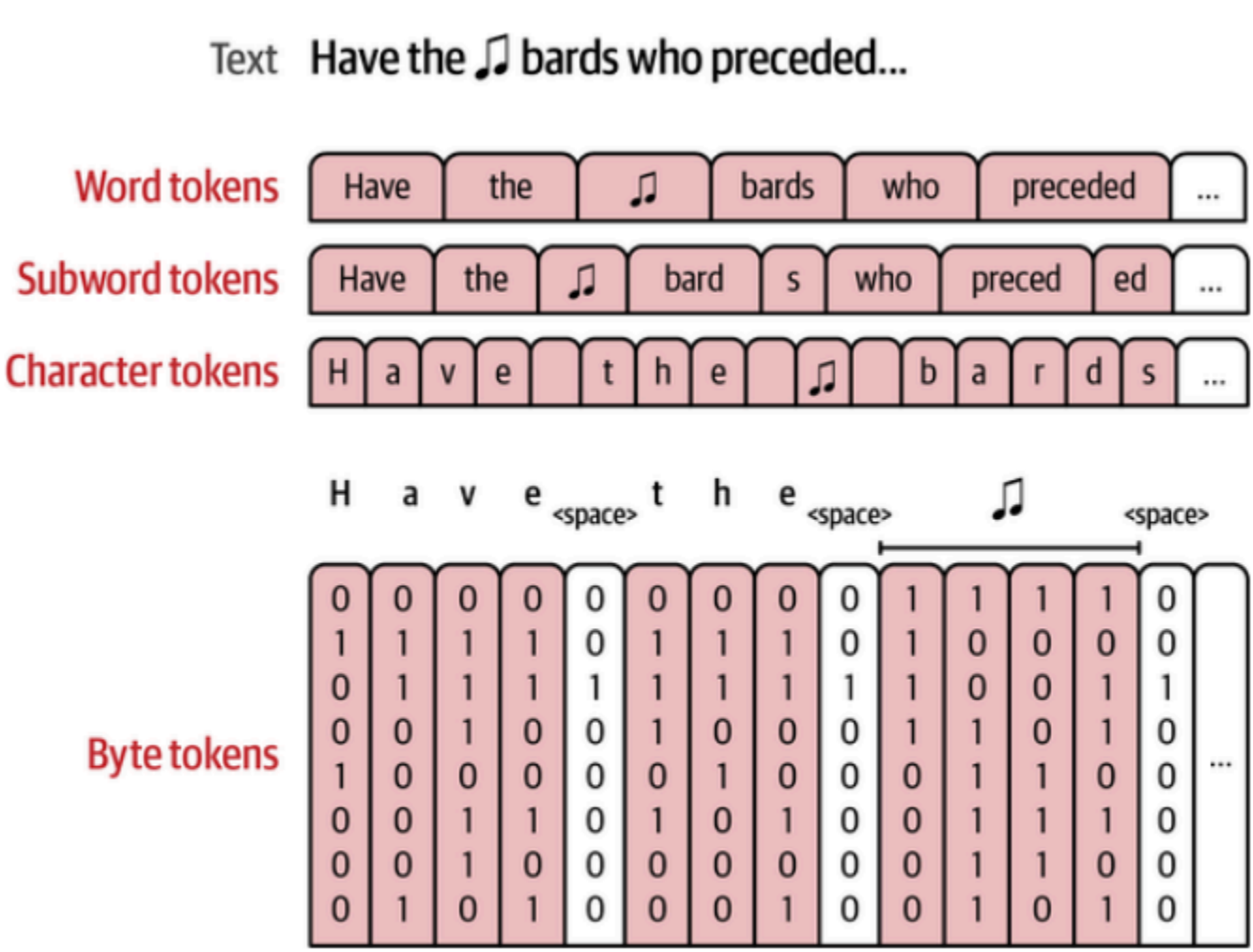


Figure 2-6. There are multiple methods of tokenization that break down the text to different sizes of components (words, subwords, characters, and bytes).

Comparing Trained LLM Tokenizers

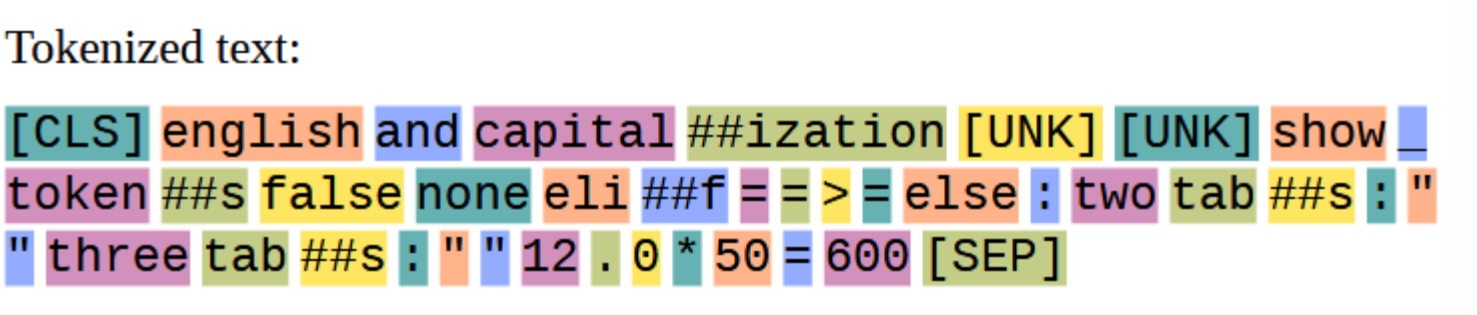
In this section, we will see using python how various tokenizers encode the following text to gain insight on their processes.

```
text = """

English and CAPITALIZATION


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

12.0*50=600

"""
```

```
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)

- **Tokenization method:** WordPiece
- **Vocabulary size**: 30,552
- **Special tokens**:
  - `unk_token [UNK]` An unknown token that the tokenizer has no specific encoding for. An example is for when a new word in the input prompt in included.
  - `sep_token [SEP]` A separator that enables certain tasks that require giving the model two texts. Example: `[CLS] What is the capital of France ? [SEP] Paris is the capital city of France . [SEP]`
  - `pad_token [PAD]` A padding token used to pad unused positions in the model's input (as the model expect a certain or minimum length of input, *context-length*)
  - `cls_token [CLS]` A special classification token for classification tasks
  - `mask_token [MASK]` A masking token used to hide tokens during the training process.



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, we see:

- The newline breaks are gone, which makes the model blind to information encoded in newlines (e.g., a chat log when each turn is a in a new line would be seen as an entire 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)

- ***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]
```

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

- Notice how `"CAPITALIZATION"` is now represented as 8 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.

GPT-2 (2019)

- ***Tokenization method***: Byte pair encoding (BPE)
- ***Vocabulary size***: 50,257
- ***Special tokens***: `<|endoftext|>`

```
English and CAP ITAL IZ ATION
Ġ Ġ Ġ Ġ Ġ Ġ
show _ t ok ens False None el if == >= else : two tabs :" "
Three tabs : " "
12 . 0 * 50 = 600
```

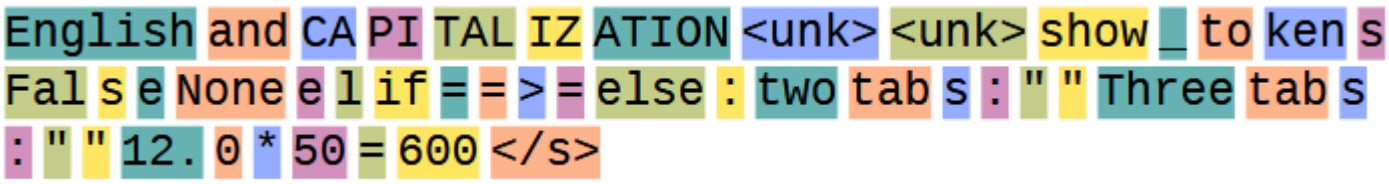
With the GPT-2 tokenizer, we notice the following:

- The newline beaks are represented in the tokenizer.
- Capitalization is preserved, and the word `"CAPITALIZATION"` is represented in four tokens.
- The two tabs are represented as 2 tokens

Flan-T5 (2022)

- ***Tokenization method:*** SentencePiece, which supports BPE and unigram language model
- ***Vocabulary size:*** 32,100
- ***Special tokens:***
  - `unk_token <unk>`
  - `pad_token <pad>`

Tokenized text:

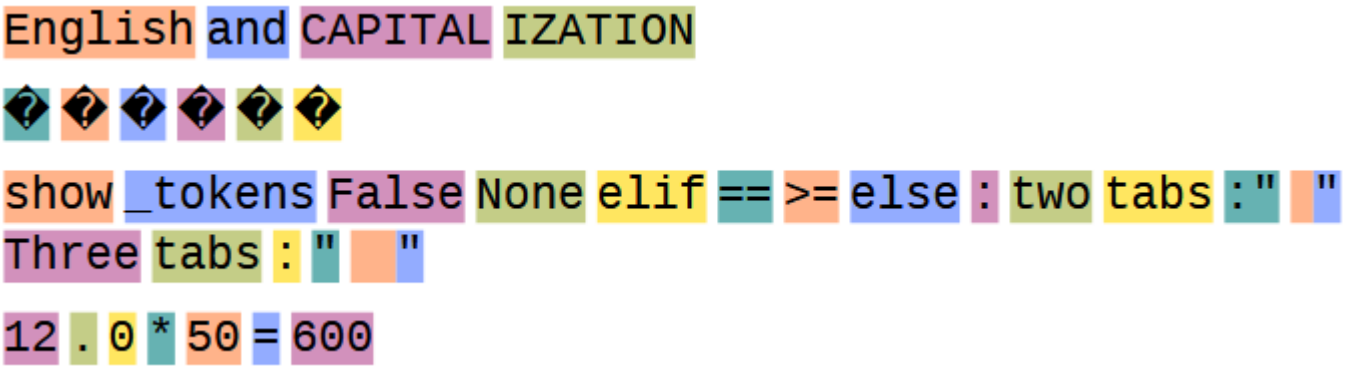


We see:

- No newline or whitespace tokens, making it unsuitable for working with code.
- The emoji and Chinese characters are both replaced by `<unk>`

GPT-4 (2023)

- ***Tokenization method:*** BPE
- ***Vocabulary size:*** A little over 100,000
- ***Special tokens:*** - `<|endoftext|>` - *Fill in the middle tokens*. These three tokens allows LLMs to generate text based not only on previous text, but also on the text after it. 1. `|<fim_prefix>|` 2. `|<fim_middle>|` 3. `|<fim_suffix>|` Tokenized text:



Important observations:

1. Spaces and a group of multiple spaces are represented as 1 token.
2. The Python keyword `elif` has its own token in GPT-4.
3. It uses fewer tokens to represent most words. Example `CAPITALIZATION` is represented with 2 tokens.

Visual Summary of tokenization methods:

BERT base model (uncased)	[CLS] english and capital ##ization [UNK] [UNK] sh ow _ token ##s false none eli ##f = = > = else : two tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SE P]
BERT base model (cased)	[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##I ON [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 Ġ Ġ Ġ Ġ Ġ Ġ show _ t ok ens False None el if == >= else : two tab s :" " 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 Ġ Ġ Ġ Ġ Ġ Ġ show _tokens False None elif == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600

Refer to the book for more comparisons

Tokenizer Properties

- ***Tokenization Methods***: BPE, WordPiece, SentencePiece are examples. More information on this here
- ***Tokenizer Parameters:***
  1. **Vocabulary size**: the number of tokens in the tokenizer's vocabulary.
  2. **Special tokens**: Examples are `[CLS]`, `[SEP]`, `[PAD]`, `<|endoftext|>`, etc. They are mostly used to indicate a structural characteristic of the input.
  3. **Capitalization**: Each tokenizer handles capital letters differently, some do not differentiate them from small letters. This could have an effect on the understanding of language of the models. Example: `Apple` (company) vs `apple` (fruit)

- ***The domain of data:*** Even with the same parameters and tokenization methods, tokenizers will perform differently based on the data they were trained on. For example, **text-focused models** would tokenize indentation (important in coding) like this:

```
def add_numbers(a, b):
....."""Add the two numbers `a` and `b`."""
```

Meanwhile, **code-focused models** would tokenize indentation and other words differently:

```
def add_numbers(a, b):
....."""Add the two numbers `a` and `b`."""
....return a + b
```

The main use and domain of the model influences the choices made in the tokenizer.

Token Embeddings

> **What are embeddings?**
>
> As discussed in Chapter 1, **embeddings** are the numeric representation space used to capture meanings and patterns in language.

A Language Model Holds Embeddings for the Vocabulary of Its Tokenizer

After a tokenizer is initialized and trained, it is used in the training of a language mode. **This is why pretrained language model is linked with its tokenizer**.

> The LLM maps each token to an embedding vector. Embedding values are learned/refined during the training process of the LLM.

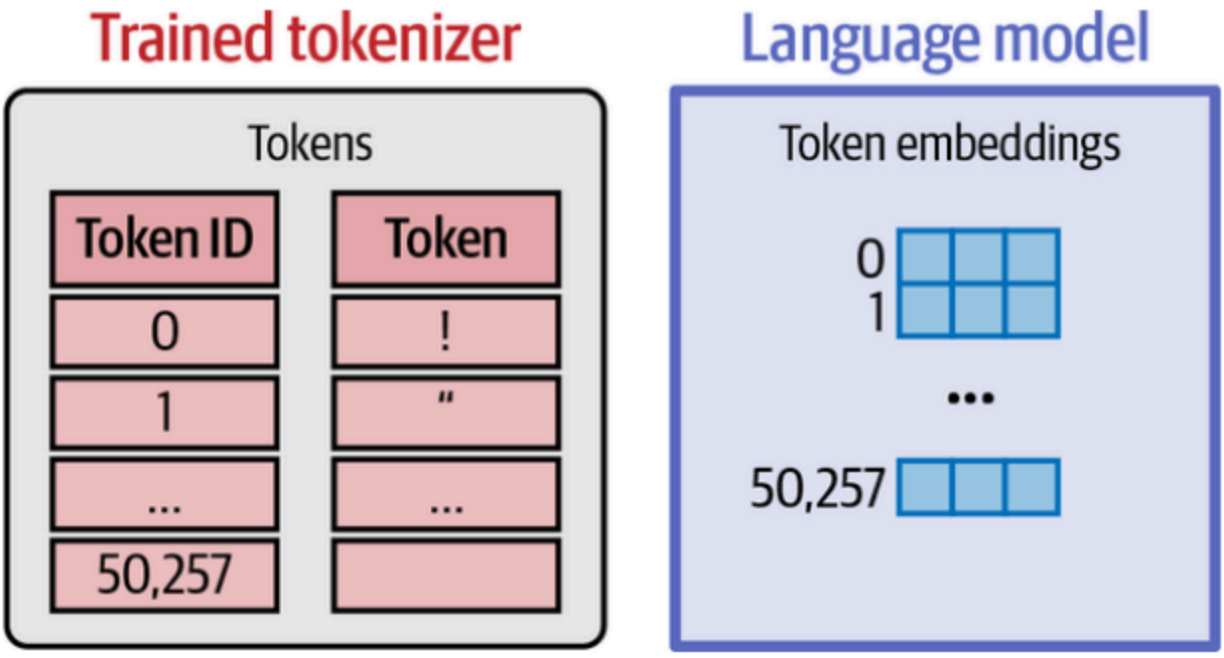


Figure 2-7. A language model holds an embedding vector associated with each token in its tokenizer.

Creating Contextualized Word Embeddings with Language Models

> Instead of representing a single word or token with a static vector, language models create word embeddings of the words based on the entire input. This *takes context into account*.

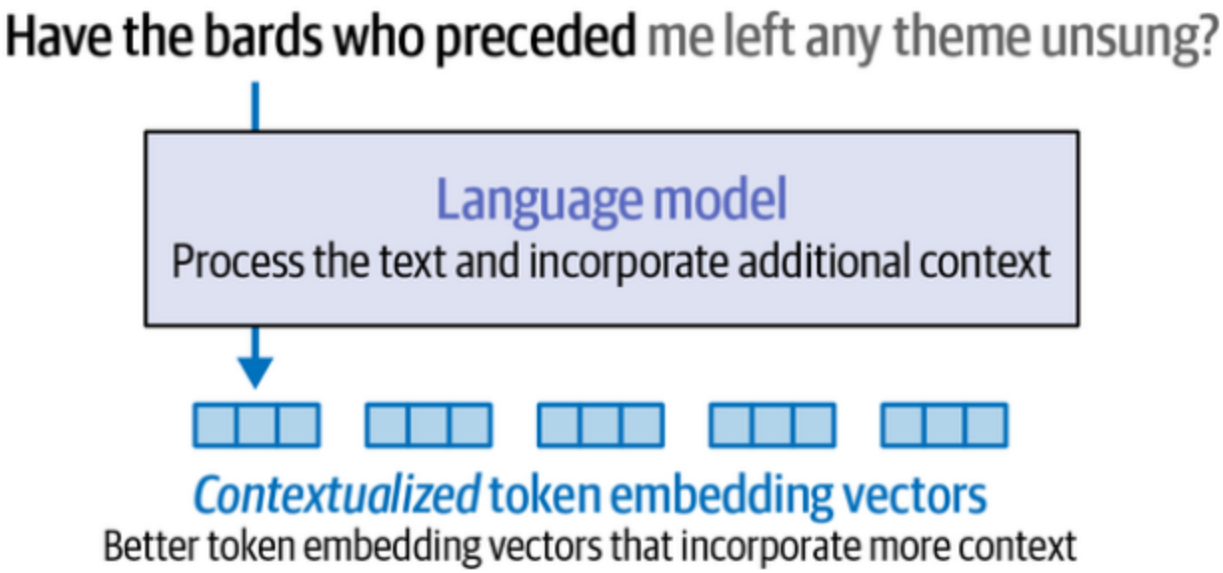


Figure 2-8. Language models produce contextualized token embeddings that improve on raw, static

Let's look at an example of how we can generate *contextualized* word embeddings:

the following code downloads a pretrained tokenizer and model, and uses them to string "Hello world":

```
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 used here is called DeBERTa v3, described in this paper.

Inspect the variable:

```
output.shape
```

This prints out:

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

Skipping the first dimension (`...Size([1, 4...`), we can read this as four tokens, each one embedded in a vector of 384 values (or dimensions)

To see the **four vectors**:

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

This prints out:

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

As you can see, this particular tokenizer and model uses `[CLS]` and `[SEP]` tokens.

The input processed by the LLM 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>)
```

> As you can see, there are 4 vectors (rows), each with 384 values. This is the **vector embedding**.

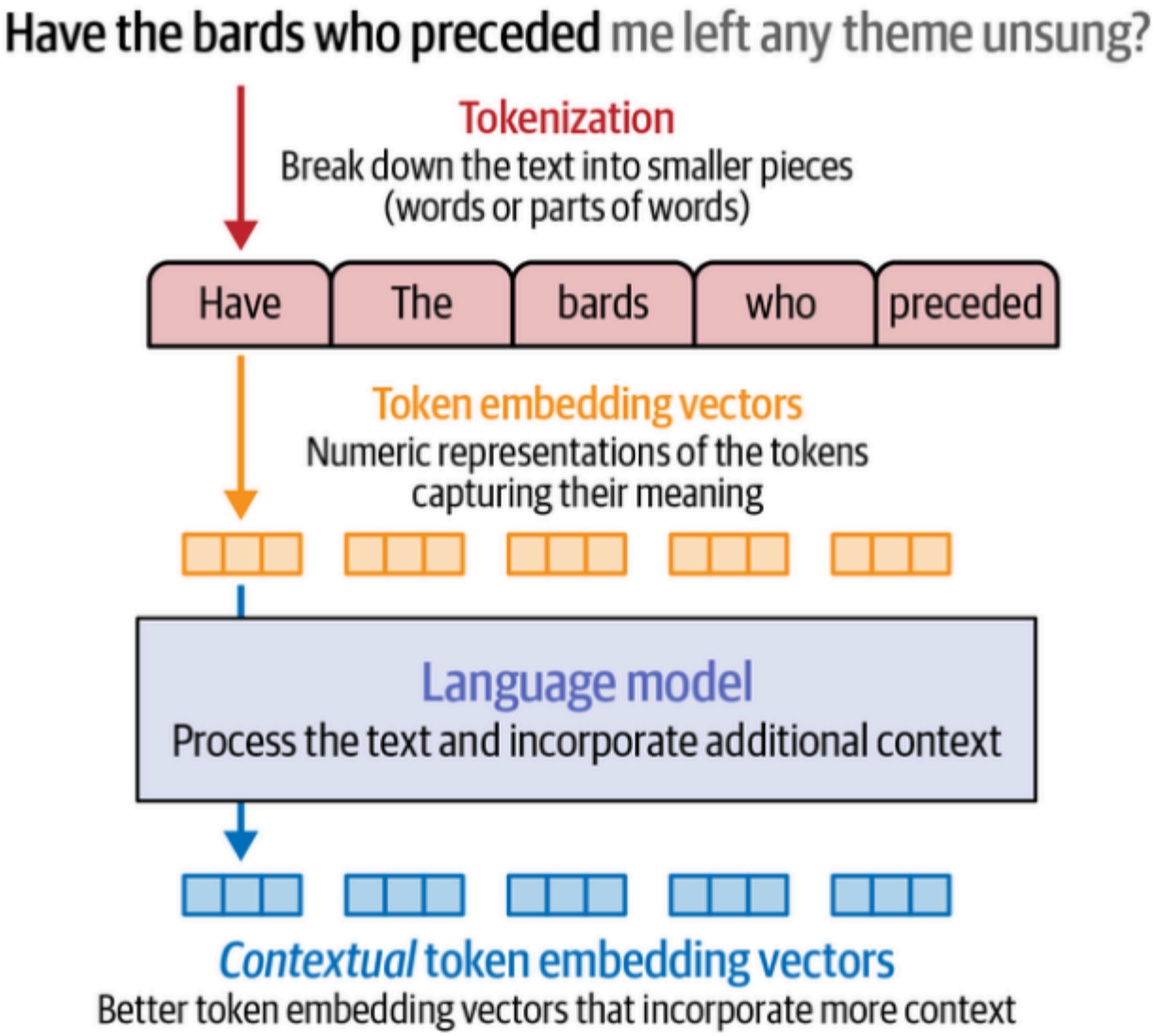


Figure 2-9. A language model operates on raw, static embeddings as its input and produces contextual text embeddings.

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.

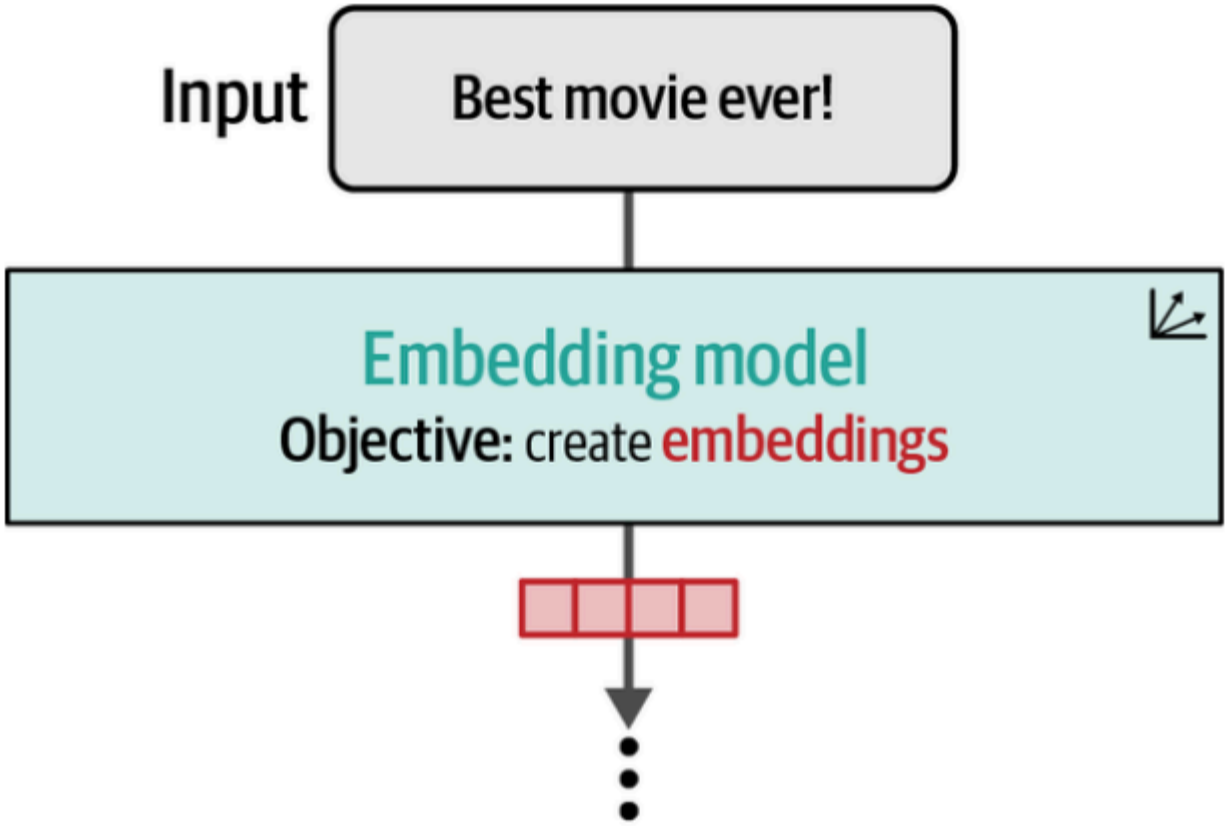


Figure 2-10. In step 1, we use the embedding model to extract the features and convert the input text to embeddings.

One of the ways text embedding vectors are produced is by averaging the values of all the token embeddings.

However, high-quality text embedding models are trained specifically for text embedding tasks.

We can produce text embeddings with `sentence-transformers`.

In the following code, we will use the `all-mpnet-base-v2` model:

```
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!")
```

To see the values and dimensions of the resulting vector embedding, use:

```
vector.shape
```

This prints out:

```
(768,)
```

As you can see, in contrast to the previous one with 4 vector embeddings, this one resulted to 1 vector embedding of **768 dimensions**.

Word Embeddings Beyond LLMs

Using pretrained Word Embeddings

```
import gensim.downloader as api

# Download embeddings (66MB, glove, trained on wikipedia, vectorsize: 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")
```

After downloading the embeddings, you can explore its 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 algorithm uses a sliding window to generate training examples. For example, a window size of 2 means we consider 1 word before and 1 word after a central word.
>
> A classification task is used to train a neural network that predicts is words commonly appear in the same context or not.

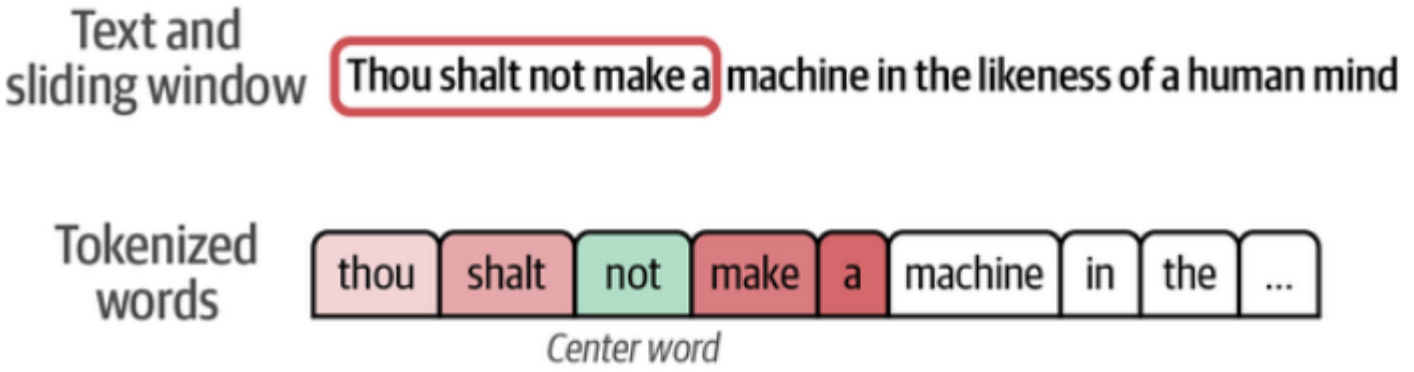


Figure 2-11. A sliding window is used to generate training examples for the word2vec algorithm to later predict if two words are neighbors or not.

In each of the produced training examples, the word in the center (e.g., `not`) is used as one input, and each of its neighbors is a distinct second input in each training example.

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.

A challenge in training word2vec is that if we only show the model positive examples (actual neighboring words), it could achieve perfect accuracy by simply always predicting "yes" - essentially gaming the system.

To address this, the training data must be enriched with negative examples - pairs of words that don't usually appear together.

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.

Simply pairing words randomly from the vocabulary works remarkably well, a technique called **negative sampling** inspired by noise-contrastive estimation.

Combined with skip-gram (the method of selecting actual neighboring words), these form word2vec's core training strategy.

Skip-gram

shalt	not	make	a	machine

input	output
make	shalt
make	not
make	a
make	machine

Negative sampling

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.

The process generates millions or billions of training examples from text, requiring initial decisions about tokenization, capitalization, and vocabulary size.

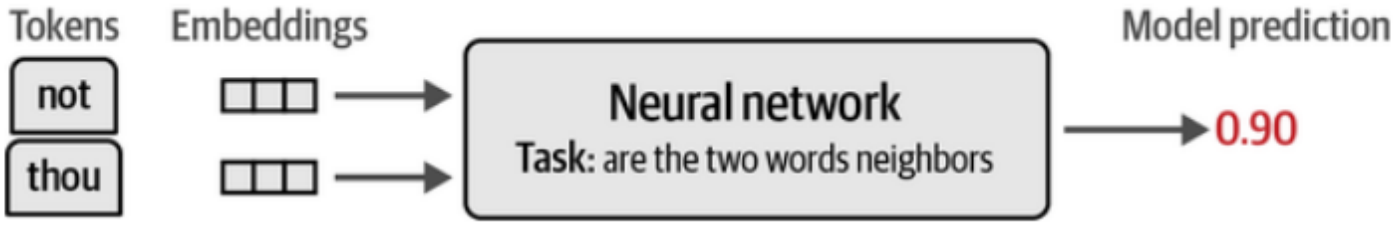


Figure 2-16. A neural network is trained to predict if two words are neighbors. It updates the embeddings in the training process to produce the final, trained embeddings.

And by the end of the training process, we have better embeddings for all the tokens in our vocabulary.