Working With Text Data

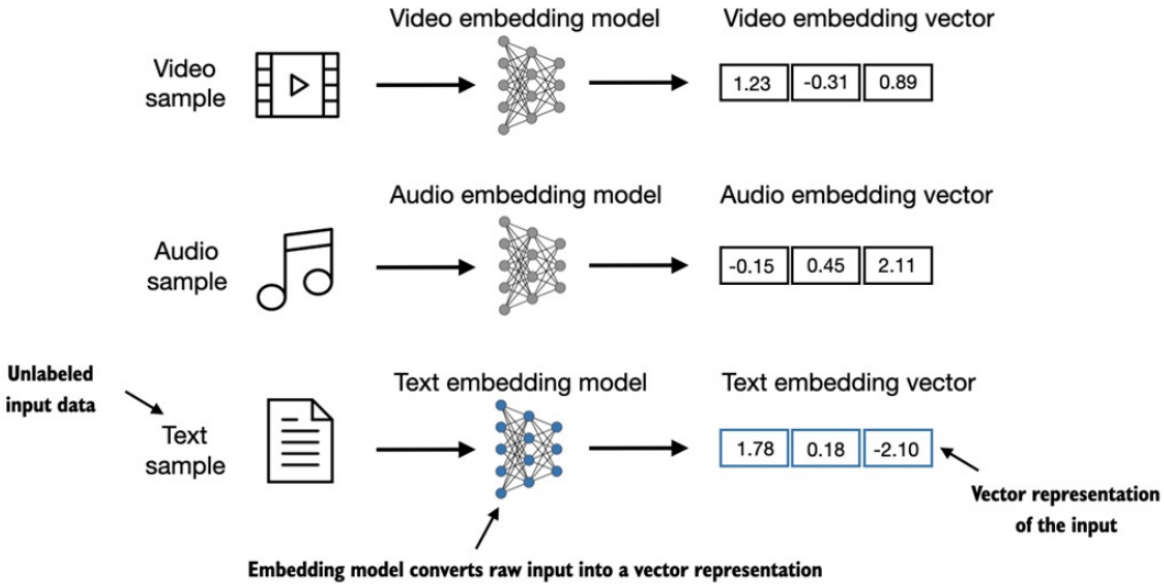
> Hamza Fatnaoui
>
> Hicham Ibn Issaghyr
>
> Ikram Arif

In this chapter, you'll learn how to prepare input text for training LLMs. This involves splitting text into individual word and subword tokens, which can then be encoded into vector representations for the LLM. You'll also learn about advanced tokenization schemes like byte pair encoding, which is utilized in popular LLMs like GPT. Lastly, we'll implement a sampling and data loading strategy to produce the input-output pairs necessary for training LLMs.

1. Understanding word embeddings

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors.

The concept of converting data into a vector format is often referred to as embedding. Using a specific neural network layer or another pretrained neural network model, we can embed different data types, for example, video, audio, and text,



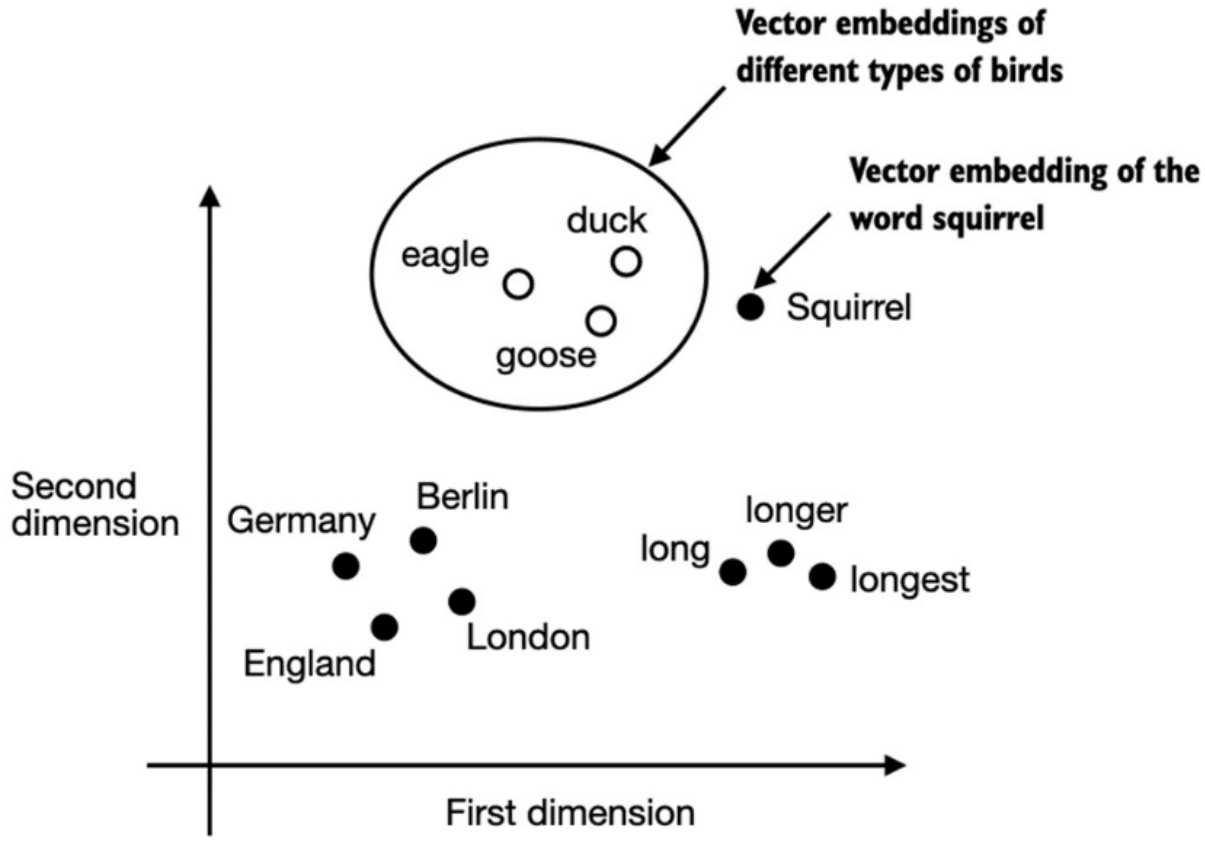
we can process various different data formats via embedding models. However, it's important to note that different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data.

At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space -- the primary purpose of embeddings is to convert non-numeric data into a format that neural networks can process.

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. Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text.

There are several algorithms and frameworks that have been developed to generate word embeddings. One of the earlier and most popular examples is the **Word2Vec** approach. Word2Vec trained neural network architecture togenerate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into 2-

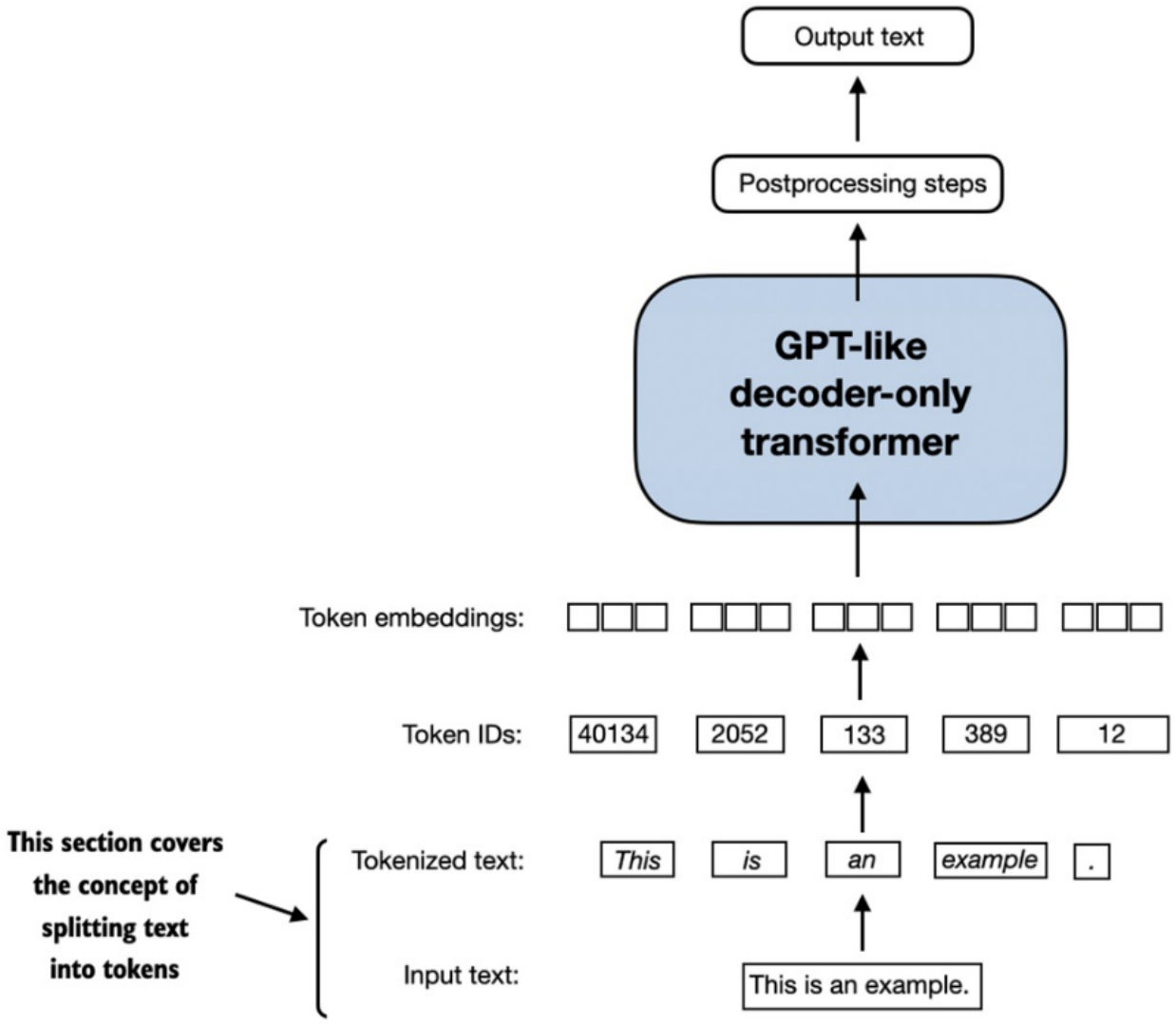
dimensional word embeddings for visualization purposes, it can be seen that similar terms cluster together:



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.

2. Tokenizing text

This section covers how we split input text into individual tokens, a required preprocessing step for creating embeddings for an LLM. These tokens are either individual words or special characters, including punctuation characters:



This is an example of tokenizing a simple file :

```
import re
```

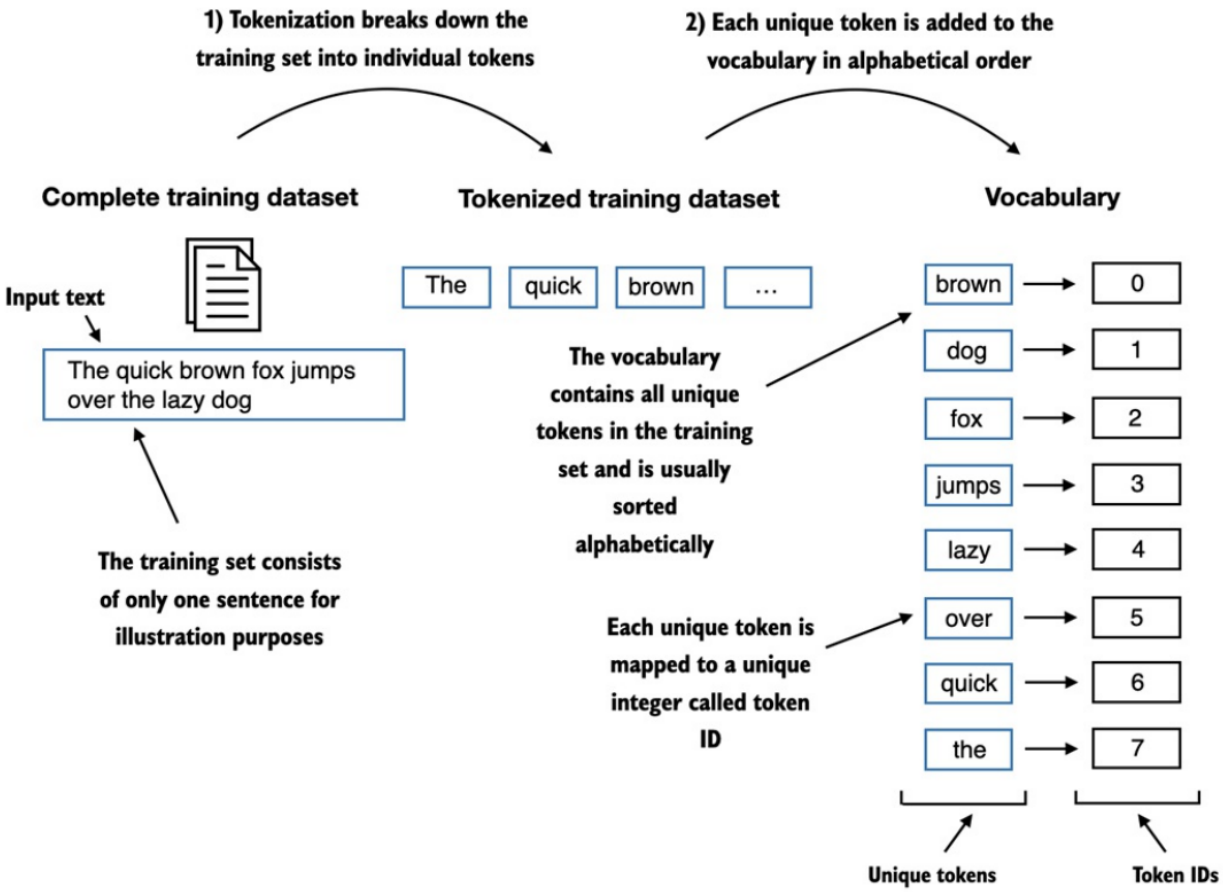
```
[1] with open("file_name", "r", encoding="utf-8") as f:
      raw_text = f.read()
```

```
[3] preprocessed = re.split(r'([,.?_!"()\']|--|\s)', raw_text)
    preprocessed = [item.strip() for item in preprocessed if item.strip()]
    # above we removed the whitespaces, not recomended if your text has python... code
    print(len(preprocessed))
```

3. Converting tokens into token IDs

In this section, we will convert these tokens from a Python string to an integer representation to produce the so-called 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 so-called vocabulary first. This vocabulary defines how we map each unique word and special character to a unique integer:



Let's now create a list of all unique tokens and sort them alphabetically,

```
all_words = sorted(list(set(preprocessed)))
vocab = {token:integer for integer,token in enumerate(all_words)}
```

The code for this tokenizer implementation is:

```
class SimpleTokenizerV1:
  def __init__(self, vocab):
    self.str_to_int = vocab
    self.int_to_str = {i:s for s,i in vocab.items()}
  def encode(self, text):
    preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
    preprocessed = [item.strip() for item in preprocessed if item.strip()]
    ids = [self.str_to_int[s] for s in preprocessed]
    return ids
  def decode(self, ids):
    text = " ".join([self.int_to_str[i] for i in ids])
    text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
    return text
```

Here is an example of using it

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know," Mrs. Gisburn said with pardo
ids = tokenizer.encode(text)
print(ids)
```

Here, you will get an error because the word pardo is not present in the vocab. So in the next section, we will test the tokenizer further on text that contains unknown words, and we will also discuss additional special tokens that can be used to provide further context for an LLM during training.

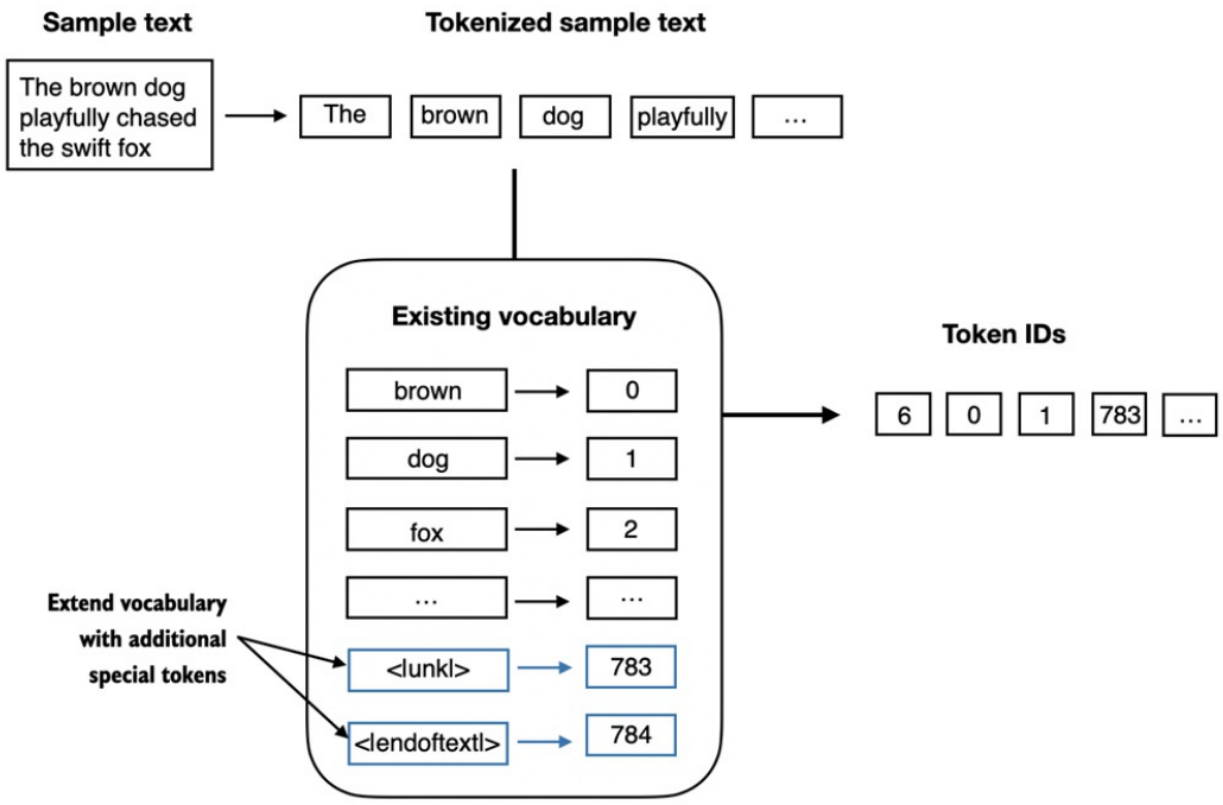
4. Adding special context tokens

In the previous section, we implemented a simple tokenizer and applied it to a passage from the training set. In this section, we will modify this tokenizer to handle unknown words.

We will also discuss the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text. These special tokens can include markers for unknown words and document boundaries, for example.

In particular, we will modify the vocabulary and tokenizer we implemented in the previous section, SimpleTokenizerV2, to support two new tokens, <|unk|> and

<|endoftext|>



we can modify the tokenizer to use an <|unk|> token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token between unrelated texts. For example, when training GPT-like LLMs on multiple independent documents or books, it is common to insert a token before each document or book that follows a previous text source. This helps the LLM understand that, although these text sources are concatenated for training, they are, in fact, unrelated.

Let's now modify the vocabulary to include these two special tokens, <unk> and <|endoftext|>, by adding these to the list of all unique words that we created in the previous section:

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

Next, the updated tokenizer class

```
class SimpleTokenizerV1:
  def __init__(self, vocab):
    self.str_to_int = vocab
    self.int_to_str = {i:s for s,i in vocab.items()}
  def encode(self, text):
    preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
    preprocessed = [item.strip() for item in preprocessed if item.strip()]
    preprocessed = [item if item in self.str_to_int else "<|unk|>" for item in preprocessed]
    ids = [self.str_to_int[s] for s in preprocessed]
    return ids
  def decode(self, ids):
    text = " ".join([self.int_to_str[i] for i in ids])
    text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
    return text
```

So far, we have discussed tokenization as an essential step in processing text as input to LLMs. 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 article ends and the next one 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.

Note that the tokenizer used for GPT models does not need any of these tokens mentioned above but only uses an <|endoftext|> token for simplicity. The <|endoftext|> is analogous to the [EOS] token mentioned above. Also, <|endoftext|> is used for padding as well. However, as we'll explore in subsequent chapters when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

Moreover, the tokenizer used for GPT models also doesn't use an <|unk|> token for out-of vocabulary words. Instead, GPT models use a byte pair encoding

tokenizer, which breaks down words into subword units, which we will discuss in the next section.

5. Byte pair encoding

We implemented a simple tokenization scheme in the previous sections for illustration purposes. This section covers a more sophisticated tokenization scheme based on a concept called **byte pair encoding (BPE)**. The BPE tokenizer covered in this section 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
```

Once installed, we can instantiate the BPE tokenizer from tiktoken as follows:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")

text = "Hello, do you like tea? <|endoftext|> In the sunlit terraces of some"
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812, 2114, 286, 617]
```

We can then convert the token IDs back into text using the decode method

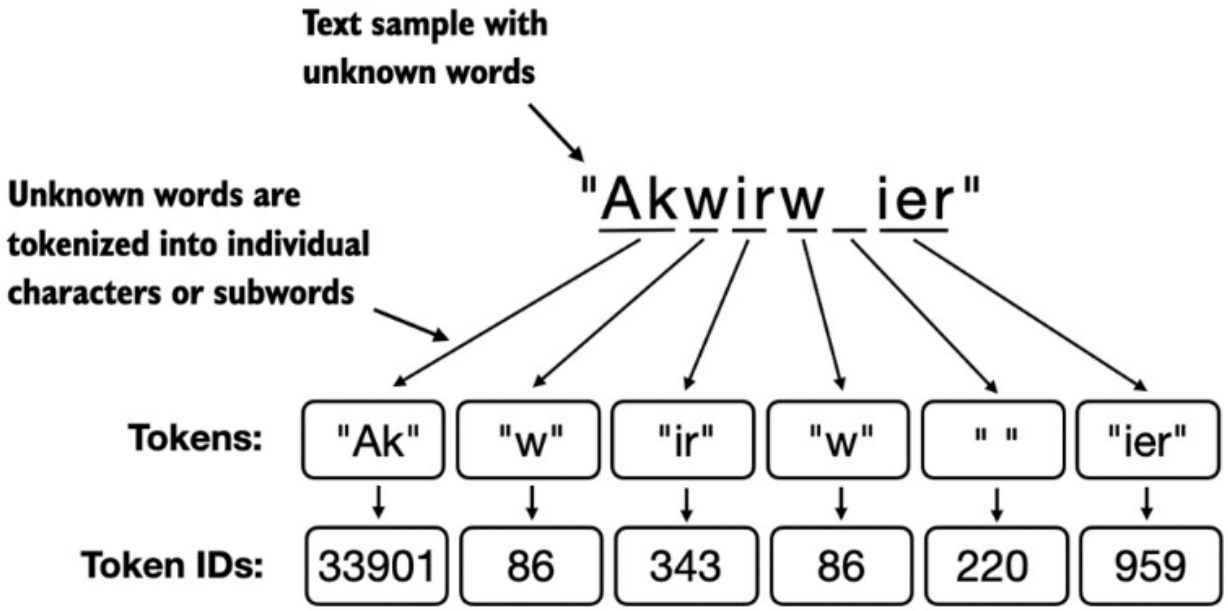
```
strings = tokenizer.decode(integers)
print(strings)
```

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

We can make two noteworthy observations based on the token IDs and decoded text above. 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 above 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?

The algorithm underlying BPE breaks down words that aren't in itspredefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters

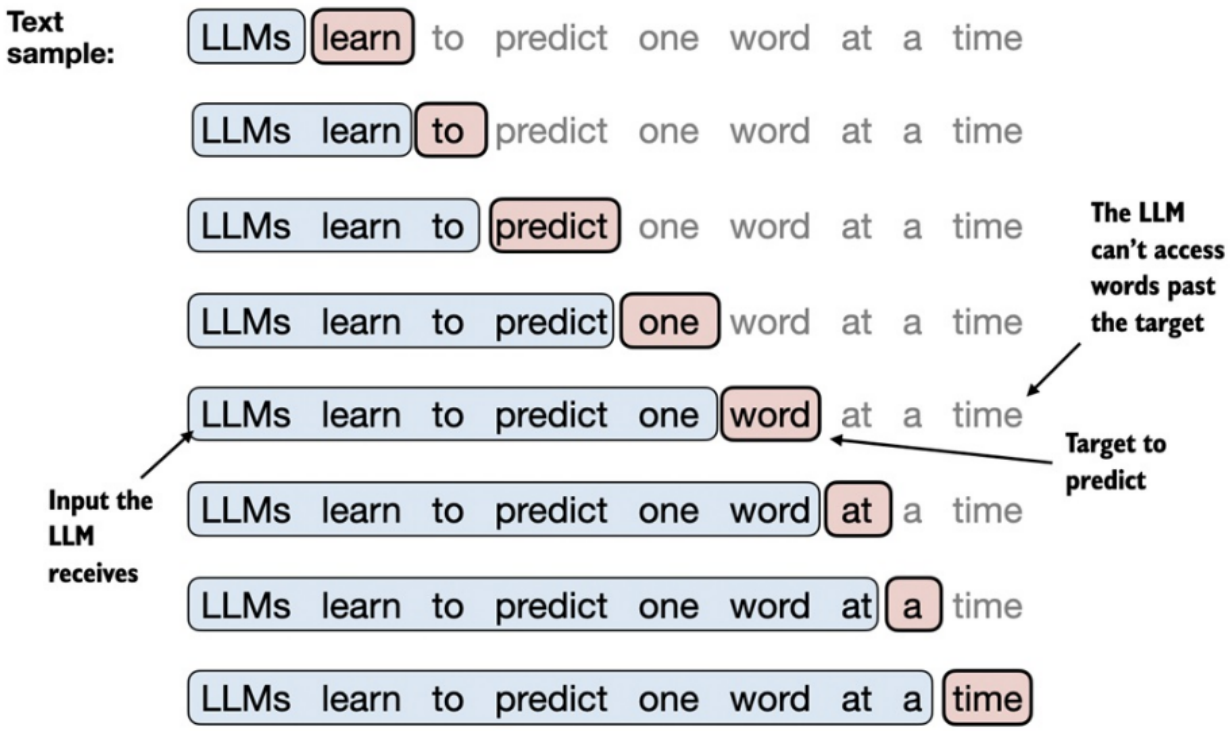


The ability to break down unknown words into individual characters ensures that the tokenizer, and consequently the LLM that is trained with it, can process any text, even if it contains words that were not present in its training data.

6. Data sampling with a sliding window

The previous section covered the tokenization steps and conversion from string tokens into integer token IDs in great detail. The next step before we can finally create the embeddings for the LLM is to generate the input-target pairs required for training an LLM.

Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM's prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure would undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.



A dataset for batched inputs and targets

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
  def __init__(self, txt, tokenizer, max_length, stride):
    self.tokenizer = tokenizer
    self.input_ids = []
    self.target_ids = []
    token_ids = tokenizer.encode(txt)
    for i in range(0, len(token_ids) - max_length, stride):
      input_chunk = token_ids[i:i + max_length]
      target_chunk = token_ids[i + 1: i + max_length + 1]
      self.input_ids.append(torch.tensor(input_chunk))
      self.target_ids.append(torch.tensor(target_chunk))
  def __len__(self):
    return len(self.input_ids)
  def __getitem__(self, idx):
    return self.input_ids[idx], self.target_ids[idx]
```

The GPTDatasetV1 class is based on the PyTorch Dataset class and defines how individual rows are fetched from the dataset, where each row consists of a number of token IDs (based on a max_length) assigned to an input_chunk tensor. The target_chunk tensor contains the corresponding targets

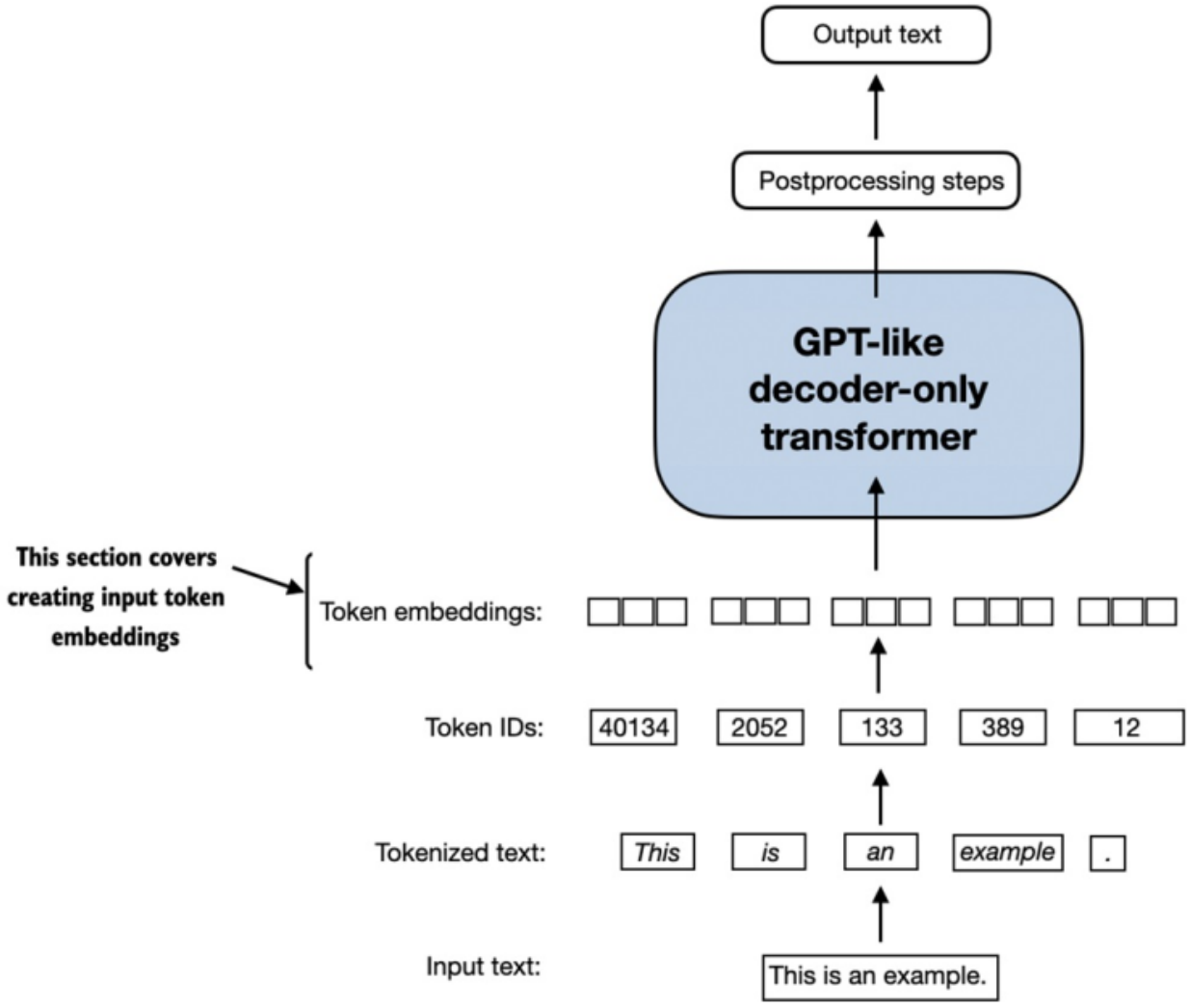
The following code will use the GPTDatasetV1 to load the inputs in batches via a PyTorch DataLoader:

```
def create_dataloader_v1(txt, batch_size=4,
  max_length=256, stride=128, shuffle=True, drop_last=True):
  tokenizer = tiktoken.get_encoding("gpt2")
  dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
  dataloader = DataLoader(
  dataset, batch_size=batch_size, shuffle=shuffle, drop_last=drop_last)
  return dataloader
```

7. Creating token embeddings

The last step for preparing the input text for LLM training is to convert the token IDs into embedding vectors

Preparing the input text for an LLM involves tokenizing text, converting text tokens to token IDs, and converting token IDs into vector embedding vectors. In this section, we consider the token IDs created in previous sections to create the token embedding vectors.



It is important to note that we initialize these embedding weights with random values as a
preliminary step. This initialization serves as the starting point for the LLM's learning process.

A continuous vector representation, or embedding, is necessary since GPT- like LLMs are deep neural networks trained with the backpropagation algorithm

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

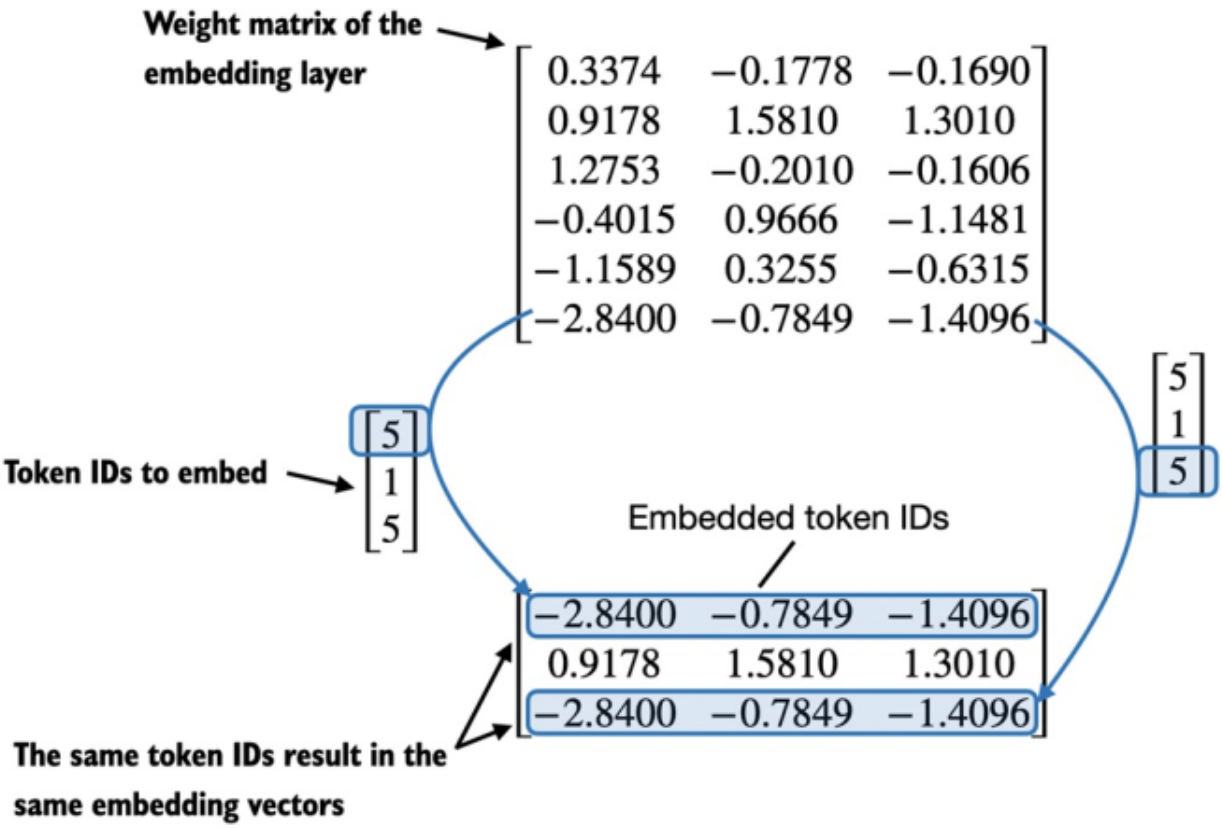
```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

8. Encoding word positions

In the previous section, we converted the token IDs into a continuous vector representation, the so-called token embeddings. In principle, this is a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism

The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence

The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or third position in the token ID input vector, will result in the same embedding vector.



In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes. However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

To achieve this, there are two broad categories of position-aware embeddings: relative positional embeddings and absolute positional embeddings.

Absolute positional embeddings are directly associated with specific positions in a sequence. For each position in the input sequence, a unique embedding is added to the token's embedding to convey its exact location. For instance, the first token will have a specific positional embedding, the
second token another distinct embedding

Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of "how far apart" rather than "at which exact position." The advantage here is that the model can generalize

better to sequences of varying lengths, even if it hasn't seen such lengths during training.

Both types of positional embeddings aim to augment the capacity of LLMs to understand the order and relationships between tokens, ensuring more accurate and context-aware predictions. The choice between them often depends on the specific application and the nature of the data being processed.

OpenAI's GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original Transformer model. This optimization process is part of the model training itself.

For a GPT model's absolute embedding approach, we just need to create another embedding layer that has the same dimension as the token_embedding_layer:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_lengthe, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

As shown in the preceding code example, the input to the pos_embeddings is usually a placeholder vector torch.arange(context_length), which contains a sequence of numbers 0, 1, ..., up to the maximum input length − 1. The context_length is a variable that represents the supported input size of the LLM. Here, we choose it similar to the maximum length of the input text. In practice, input text can be longer than the supported context length, in which case we have to truncate the text.

We can now add these directly to the token embeddings

```
input_embeddings = token_embeddings + pos_embeddings
```

9. Summary

- LLMs require textual data to be converted into numerical vectors, known as embeddings since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words orcharacters. Then, the tokens are converted into integer representations, termed token IDs.
- Special tokens, such as <|unk|> and <|endoftext|>, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between unrelated texts.
- The byte pair encoding (BPE) tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate input-target pairs for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings that are added to the token embedding vectors and are optimized during the model training.