

前言

到目前为止，我们已经介绍了大型语言模型 (LLM) 的一般结构，并了解到它们是在海量文本上进行预训练的。具体来说，我们重点关注基于 Transformer 架构的仅解码器 LLM，该架构是 ChatGPT 和其他流行的类 GPT LLM 中使用的模型的基础。

在预训练阶段，LLM 每次处理一个单词。使用下一个单词预测任务训练拥有数百万甚至数十亿个参数的 LLM，可以生成性能卓越的模型。这些模型可以进一步微调，以遵循通用指令或执行特定的目标任务。但在实现和训练 LLM 之前，我们需要准备训练数据集。

你将学习如何准备用于训练 LLM 的输入文本。这包括将文本拆分成单个单词和子单词 token，然后将其编码为 LLM 的向量表示。你还将学习高 token 化方案，例如字节对编码，该方案在 GPT 等流行的 LLM 中得到应用。最后，我们将实现一个采样和数据加载策略，以生成训练 LLM 所需的输入输出对。

2.1 理解词嵌入

深度神经网络模型（包括 LLM）无法直接处理原始文本。由于文本具有分类属性，它与用于实现和训练神经网络的数学运算不兼容。因此，我们需要一种将单词表示为连续值向量的方法。（注：LLM 的本质依然是计算，由于文本本身没法进行计算，所以使用词嵌入来表示文本，以实现计算）

将数据转换为向量格式的概念通常被称为嵌入。

使用特定的神经网络层或其他预训练的神经网络模型，我们可以嵌入不同类型的数据，例如视频、音频和文本，如图 2.2 所示。然而，需要注意的是，不同的数据格式需要不同的嵌入模型。例如，为文本设计的嵌入模型可能不适合嵌入音频或视频数据。

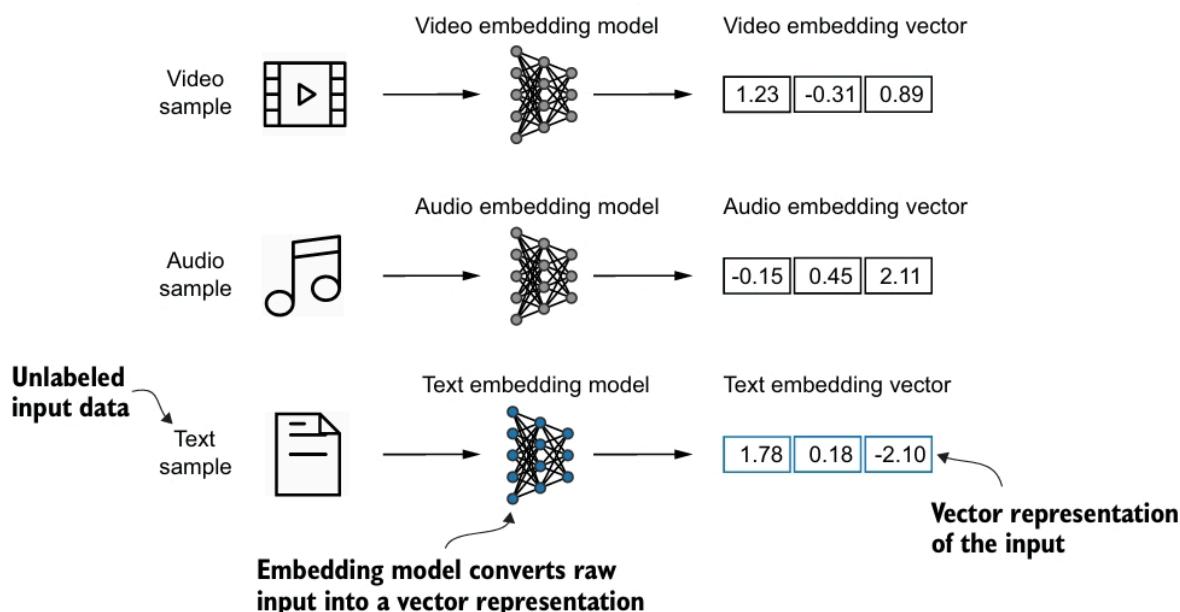


Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates the process of converting raw data into a three-dimensional numerical vector.

嵌入的核心是从离散对象（例如单词、图像甚至整个文档）到连续向量空间中的点的映射——嵌入的主要目的是将非数字数据转换为神经网络可以处理的格式。

虽然词向量是最常见的文本向量形式，但也有用于句子、段落或整篇文档的向量。句子或段落向量是检索增强生成 (RAG) 的常用选择。RAG 将生成（例如生成文本）与检索（例如搜索外部知识库）相结合，以便在生成文本时提取相关信息，由于我们的目标是训练类似 GPT 的 LLM，使其能够学习一次生成一个单词的文本，因此我们将重点关注词向量。

目前已开发出多种用于生成词向量的算法和框架。Word2Vec 方法便是其中较早且最流行的例子之一。Word2Vec 训练神经网络架构，通过预测给定目标词的上下文来生成词向量，反之亦然。Word2Vec 的核心思想是，**出现在相似上下文中的单词往往具有相似的含义**。因此，当将其投影到二维词向量中进行可视化时，**相似的术语会聚集在一起**，如图 2.3 所示。

词向量的维度可以多种多样，从一维到数千维不等。更高的维度或许能够捕捉到更细微的关系，但计算效率会有所下降。

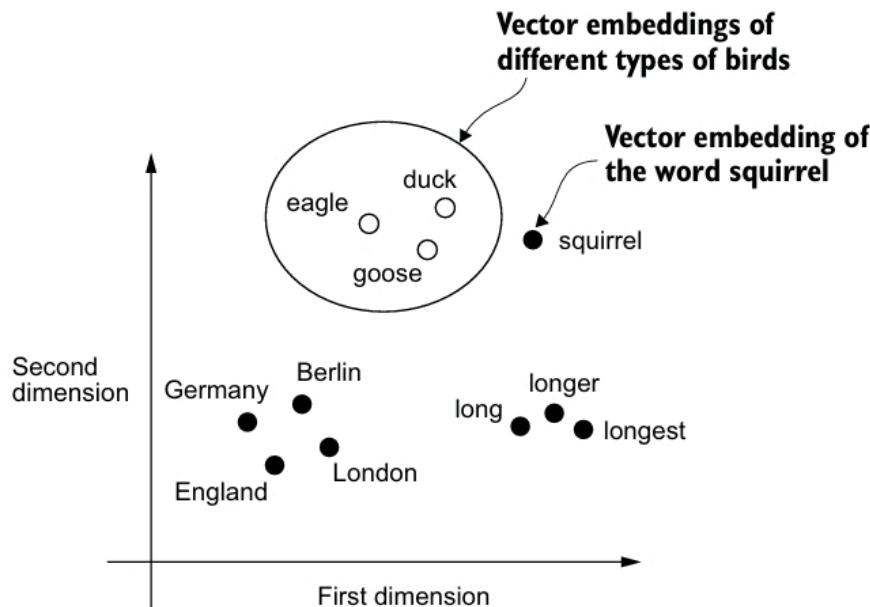


Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space than in countries and cities.

虽然我们可以使用 Word2Vec 等预训练模型来为机器学习模型生成嵌入，但 LLM 通常会生成自己的嵌入，这些嵌入是输入层的一部分，并在训练过程中进行更新。**将嵌入优化作为 LLM 训练的一部分（而不是使用 Word2Vec）** 的优势在于，这些嵌入会根据特定的任务和数据进行优化。我们将在本章后面实现此类嵌入层。（LLM 还可以创建上下文化的输出嵌入，我们将在第 3 章中讨论。）

遗憾的是，高维嵌入对可视化提出了挑战，因为我们的感官知觉和常见的图形表示本质上仅限于三维或更少，因此图 2.3 在二维散点图中展示了二维嵌入。然而，在使用 LLM 时，我们通常使用维度更高的嵌入。对于 GPT-2 和 GPT-3，嵌入大小（通常称为模型隐藏状态的维数）会根据具体的模型变体和大小而变化。这是性能和效率之间的权衡。最小的 GPT-2 模型（1.17 亿和 1.25 亿参数）使用 768 维的嵌入大小来提供具体的示例。最大的 GPT-3 模型（1.75 亿参数）使用 12,288 维的嵌入大小。

接下来，我们将介绍准备 LLM 所使用的嵌入所需的步骤，包括**将文本拆分为单词、将单词转换为token**以及**将token转换为嵌入向量**。

2.2 编码文本

让我们讨论一下如何将输入文本拆分成单个 token，这是创建 LLM 嵌入的必要预处理步骤。这些 token 可以是单个单词，也可以是特殊字符，包括标点符号，如图 2.4 所示。

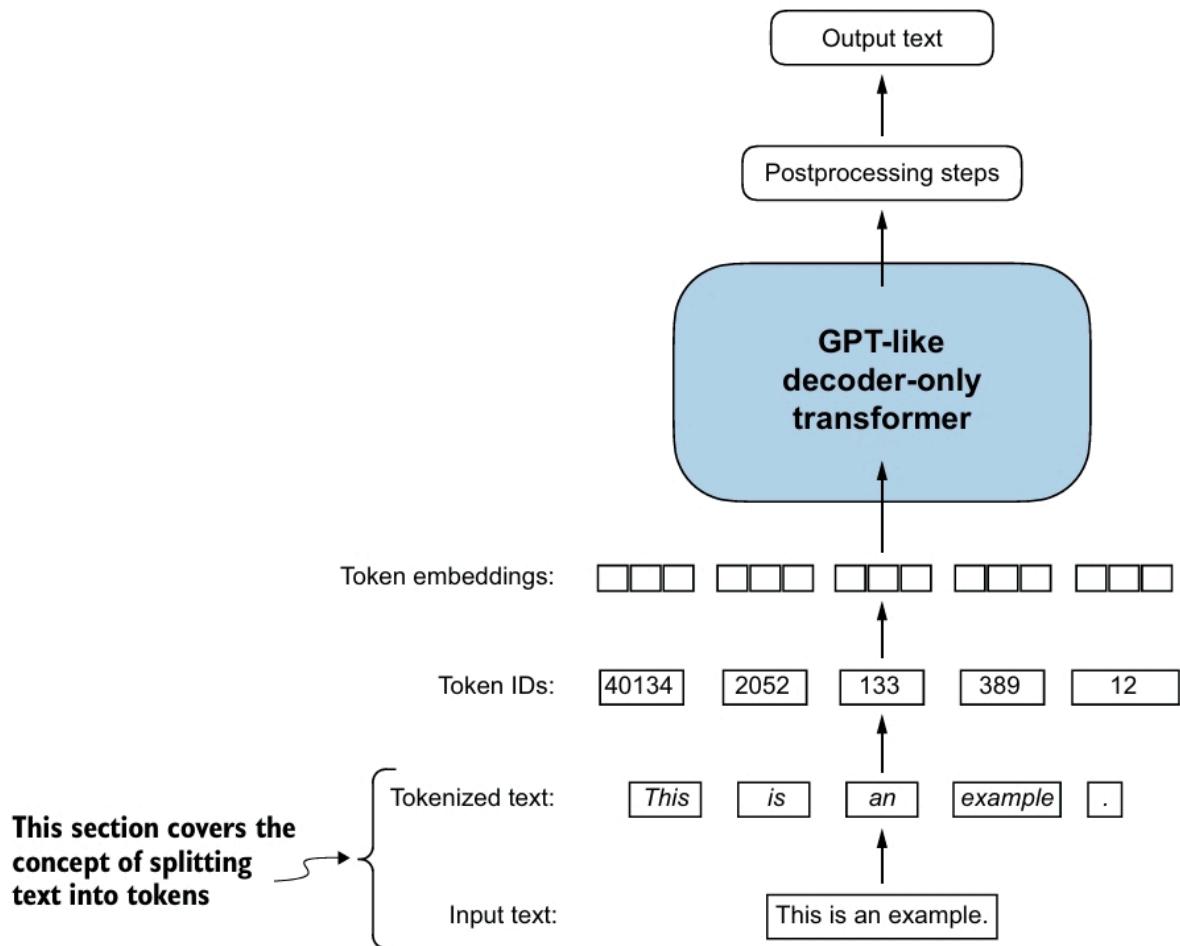


Figure 2.4 A view of the text processing steps in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters.

我们将用于LLM训练的文本是伊迪丝·华顿 (Edith Wharton) 的短篇小说《判决》(The Verdict)。该小说已发布到公共领域，因此允许用于LLM的训练任务。该文本可在维基文库(Wikisource)上找到，网址为 https://en.wikisource.org/wiki/The_Verdict，您可以将其复制粘贴到一个文本文件中，我将其复制成了一个文本文件“the-verdict.txt”。或者，您也可以在本书的 GitHub 代码库 <https://mng.bz/Adng> 中找到这个“the-verdict.txt”文件。您可以使用以下 Python 代码下载该文件：

```

1 import urllib.request
2
3 url = ("https://raw.githubusercontent.com/rasbt/"
4
5 "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
6
7 "the-verdict.txt")
8
9 file_path = "the-verdict.txt"
10
11 urllib.request.urlretrieve(url, file_path)

```

```
[2]: import urllib.request

url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")

file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

```
[2]: ('the-verdict.txt', <http.client.HTTPMessage at 0x7fc353fdc5c0>)
```

接下来，我们可以使用 Python 的标准文件读取程序加载 the-verdict.txt 文件。

Listing 2.1 Reading in a short story as text sample into Python

```
1 with open("the-verdict.txt", "r", encoding="utf-8") as f:
2
3     raw_text = f.read()
4
5 print("Total number of character:", len(raw_text))
6
7 print(raw_text[:99])
```

读取TXT文件，统计文件字符数并打印前100个字符

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

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

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

我们的目标是将这个 20,479 个字符的短篇故事**编码为单个单词和特殊字符，然后将其转换为用于 LLM 训练的嵌入。**

我们如何才能最好地拆分这段文本以获取token列表？为此，我们进行了一些尝试，并使用 Python 的正则表达式库 re 进行演示。

（您无需学习或记忆任何正则表达式语法，因为我们稍后会过渡到预构建的tokenizer。）

使用一些简单的示例文本，我们可以使用 re.split 命令，其语法如下，根据空格字符拆分文本：

```
1 import re
2
3 text = "Hello, world. This, is a test."
4 result = re.split(r'(\s)', text)
5
6 print(result)
```

测试正则语法库re

```
[3]: import re

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

print(result)
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ', 'test.']}
```

结果是单个单词、空格和标点符号的列表

这种简单的分词方案主要用于将示例文本拆分成单个单词；然而，有些单词仍然与标点符号相连，我们希望将其作为单独的列表条目。我们也避免将所有文本都小写，因为**大写有助于LLM区分专有名词和普通名词，理解句子结构，并学习生成具有正确大写字母的文本。**

让我们修改空格 (\s)、逗号和句点 (.) 上的正则表达式拆分：

```
1 result = re.split(r'([.,]|\s)', text)
2
3 print(result)
```

我们可以看到，单词和标点符号现在都是单独的列表条目，正如我们想要的那样

修改空格 (\s)、逗号和句点 (.) 上的正则表达式拆分

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

print(result)
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', ' ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

剩下的一个小问题是列表中仍然包含空格字符。我们也可以安全地删除这些冗余字符，如下所示：

```
1 result = [item for item in result if item.strip()]
2
3 print(result)
```

安全删除冗余空格字符

```
[5]: result = [item for item in result if item.strip()]

print(result)
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

我们在此设计的token化方案在简单的示例文本上效果很好。让我们进一步修改它，使其能够处理其他类型的标点符号，例如问号、引号以及我们之前在伊迪丝·华顿短篇小说的前 100 个字符中看到的双破折号，以及其他特殊字符：

```

1 text = "Hello, world. Is this-- a test?"
2
3 result = re.split(r'([.,;?_!"()\\']|--|\\s)', text)
4
5 result = [item.strip() for item in result if item.strip()]
6
7 print(result)

```

处理更多类型的字符

```

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

result = re.split(r'([.,;?_!"()\\']|--|\\s)', text)

result = [item.strip() for item in result if item.strip()]

print(result)

```

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

根据图 2.5 中总结的结果，我们可以看出，我们的分词方案现在可以成功处理文本中的各种特殊字符。

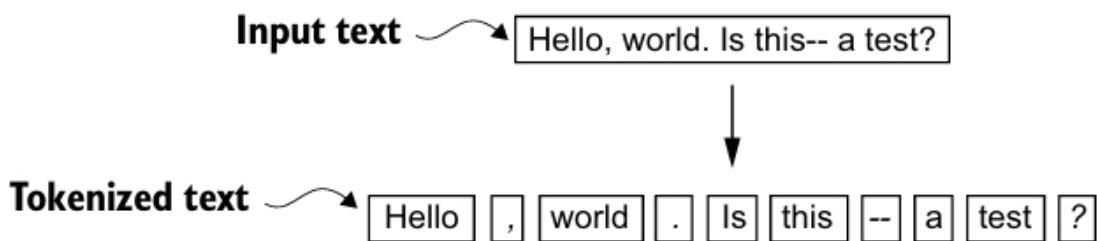


Figure 2.5 The tokenization scheme we implemented so far splits text into individual words and punctuation characters. In this specific example, the sample text gets split into 10 individual tokens.

现在我们已经有了一个基本的tokenizer，让我们将它应用到伊迪丝·华顿的整个短篇小说中：

```

1 preprocessed = re.split(r'([.,;?_!"()\\']|--|\\s)', raw_text)
2
3 preprocessed = [item.strip() for item in preprocessed if item.strip()]
4
5 print(len(preprocessed))

```

使用简单编码器编码整篇小说

```

[8]: preprocessed = re.split(r'([.,;?_!"()\\']|--|\\s)', raw_text)

preprocessed = [item.strip() for item in preprocessed if item.strip()]

print(len(preprocessed))

```

4690

这条打印语句输出 4690，这是这段文本中的token数量（不包含空格）。让我们打印前 30 个tokens，以便快速查看。

```
1 | print(preprocessed[:30])
```

打印查看前30个tokens

```
print(preprocessed[:30])
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap', 'genius', '--', 'though', 'a', 'good', 'fellow', 'enough', '--', 'so', 'it', 'was', 'no',
'great', 'surprise', 'to', 'me', 'to', 'hear', 'that', ',', 'in']
```

输出结果显示，我们的分词器似乎能很好地处理文本，因为所有单词和特殊字符都被整齐地分开了。

2.3 把tokens转化成token IDs

接下来，让我们将这些tokens从Python字符串转换为整数表示，以生成token ID。此转换是将token ID转换为嵌入向量之前的中间步骤。

为了将之前生成的tokens映射到token ID，我们必须首先构建一个词汇表。这个词汇表定义了我们如何将每个唯一的单词和特殊字符映射到唯一的整数，如图 2.6 所示。

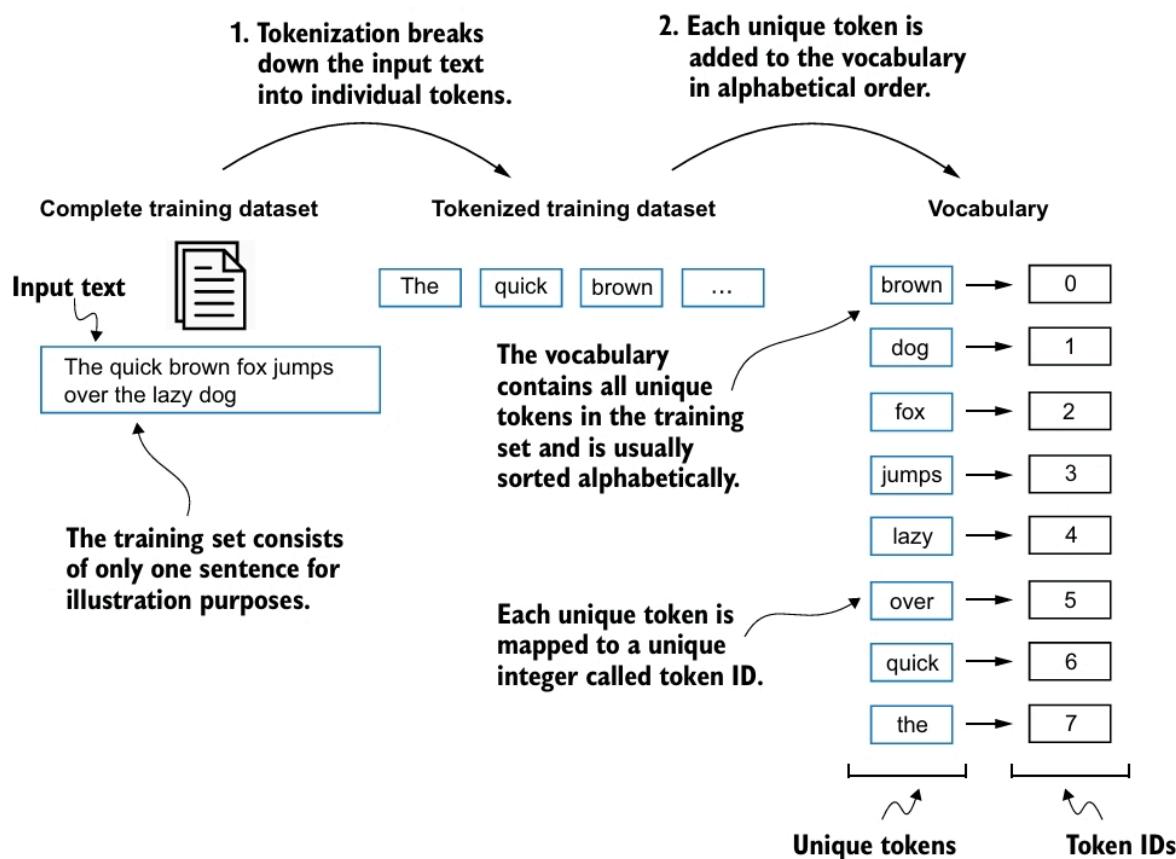


Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposefully small and contains no punctuation or special characters for simplicity.

现在我们已经对伊迪丝·华顿的短篇小说进行了token化，并将其赋值给一个名为 preprocessed 的 Python 变量，接下来让我们创建一个包含所有唯一token的列表，并按字母顺序对它们进行排序，以确定词汇量。

```
1 | all_words = sorted(set(preprocessed))
2 |
3 | vocab_size = len(all_words)
4 |
5 | print(vocab_size)
```

将编码后的tokens映射为唯一整数 (token ID)

```
[11]: all_words = sorted(set(preprocessed))

vocab_size = len(all_words)

print(vocab_size)

1130
```

通过此代码确定词汇表大小为 1,130 后，我们创建词汇表并打印其前 51 个条目以供说明。

Listing 2.2 Creating a vocabulary

```
1 vocab = {token:integer for integer,token in enumerate(all_words)}
2
3 for i, item in enumerate(vocab.items()):
4     print(item)
5     if i >= 50:
6         break
```

创建词汇表并打印前51个条目

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

for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break

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

如我们所见，该词典包含与唯一整数标签相关联的各个token。我们的下一个目标是应用该词汇表将新文本转换为token ID（图 2.7）。

Tokenization breaks down the training set into individual tokens.

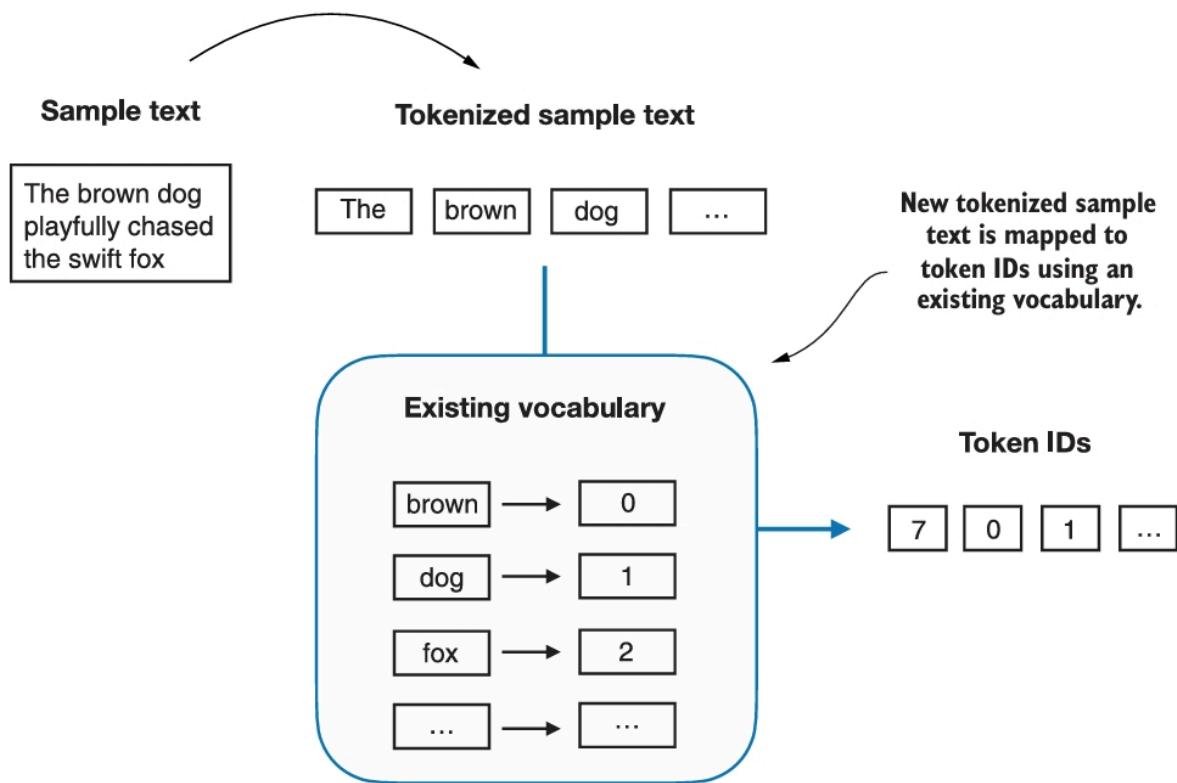


Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation or special characters for simplicity.

当我们想要将LLM的输出从数字转换回文本时，我们需要一种将 token ID 转换为文本的方法。为此，我们可以创建一个词汇表的逆版本，将 token ID 映射回相应的文本 token。

让我们用 Python 实现一个完整的分词器类，其中包含一个**encode** 方法，该方法将文本拆分成词条，并执行字符串到整数的映射，从而通过词汇表生成词条 ID。此外，我们还将实现一个**decode**方法，该方法执行反向整数到字符串的映射，从而将词条 ID 转换回文本。以下清单展示了此分词器实现的代码。

Listing 2.3 实现一个简单的文本tokenizer

```
1 class SimpleTokenizerV1:
2     def __init__(self, vocab):
3         self.str_to_int = vocab # 将词汇表存储为类属性，以便在编码和解码方法中访问
4         self.int_to_str = {i:s for s,i in vocab.items()} # 创建一个逆向词汇表,
将token ID 映射回原始文本 tokens
5
6     # 将输入文本处理成token IDs
7     def encode(self, text):
8         # 分词
9         preprocessed = re.split(r'([,.?_!"()\\]|--|\\s)', text)
10        # 处理空格
11        preprocessed = [
12            item.strip() for item in preprocessed if item.strip()
13        ]
14        # 使用词汇表将tokens映射为token IDs
15        ids = [self.str_to_int[s] for s in preprocessed]
16        return ids
17
```

```

18     # 将token IDs映射回文本
19     def decode(self, ids):
20         # 将ids中的tokenIDs通过词汇表反向映射并连接起来
21         text = " ".join([self.int_to_str[i] for i in ids])
22         # 删除指定标点符号前的空格
23         text = re.sub(r'\s+([.,?!"](\s*))', r'\1', text)
24         return text

```

实现一个简单的文本tokenizer

```

class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab # 将词汇表存储为类属性，以便在编码和解码方法中访问
        self.int_to_str = {i:s for s,i in vocab.items()} # 创建一个逆向词汇表，将token ID 映射回原始文本 tokens

    # 将输入文本处理成token IDs
    def encode(self, text):
        # 分词
        preprocessed = re.split(r'([.,?!"](\s*)|--|\s+)', text)
        # 处理空格
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        # 使用词汇表将tokens映射为token IDs
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    # 将token IDs映射回文本
    def decode(self, ids):
        # 将ids中的tokenIDs通过词汇表反向映射并连接起来
        text = " ".join([self.int_to_str[i] for i in ids])
        # 删除指定标点符号前的空格
        text = re.sub(r'\s+([.,?!"](\s*))', r'\1', text)
        return text

```

使用 SimpleTokenizerV1 Python 类，我们现在可以通过现有词汇表实例化新的tokenizer对象，然后我们可以使用这些对象对文本进行编码和解码，如图 2.8 所示。

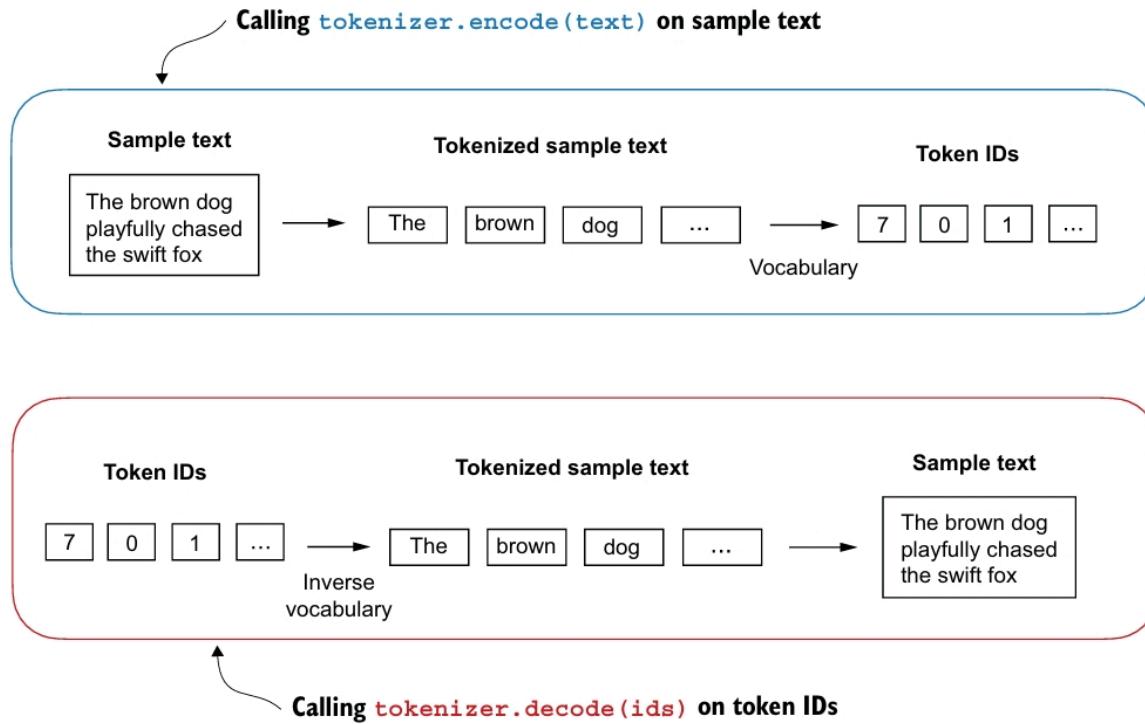


Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text tokens, and concatenates the text tokens into natural text.

让我们从 SimpleTokenizerV1 类实例化一个新的 tokenizer 对象，并对伊迪丝·华顿短篇小说中的一段进行 token 化，以便在实践中尝试一下：

实例化tokenizer对象并进行编解码

```
[15]: tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,""
         Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)

[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108, 754, 793, 7]

[16]: print(tokenizer.decode(ids))

" It' s the last he painted, you know," Mrs. Gisburn said with pardonable pride.
```

到目前为止，一切顺利。我们实现了一个分词器，它能够基于训练集中的片段对文本进行分词和去分词。现在，让我们将它应用于训练集中未包含的新文本样本：

```
1 | text = "Hello, do you like tea?"
2 | print(tokenizer.encode(text))
```

使用文中未出现过的词进行编码会报错

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))

-----
KeyError                                                 Traceback (most recent call last)
Cell In[17], line 2
      1 text = "Hello, do you like tea?"
----> 2 print(tokenizer.encode(text))

Cell In[14], line 15, in SimpleTokenizerV1.encode(self, text)
    11 preprocessed = [
    12     item.strip() for item in preprocessed if item.strip()
    13 ]
    14 # 使用词汇表将tokens映射为token IDs
-> 15 ids = [self.str_to_int[s] for s in preprocessed]
    16 return ids

KeyError: 'Hello'
```

问题在于，“Hello”这个词在短篇小说《判决》中没有出现。因此，它不包含在词汇表中。这凸显了在进行LLM工作时，需要考虑大量且多样化的训练集来扩展词汇量。

2.4 加入特殊的上下文tokens

我们需要修改分词器来处理未知词。我们还需要解决特殊上下文tokens的使用和添加问题，这些tokens可以增强模型对文本中上下文或其他相关信息的理解。这些特殊tokens可以包含例如未知词和文档边界 的token。具体来说，我们将修改词汇表和分词器 SimpleTokenizerV2，以支持两个新tokens <|unk|> 和 <|endoftext|>，如图 2.9 所示。

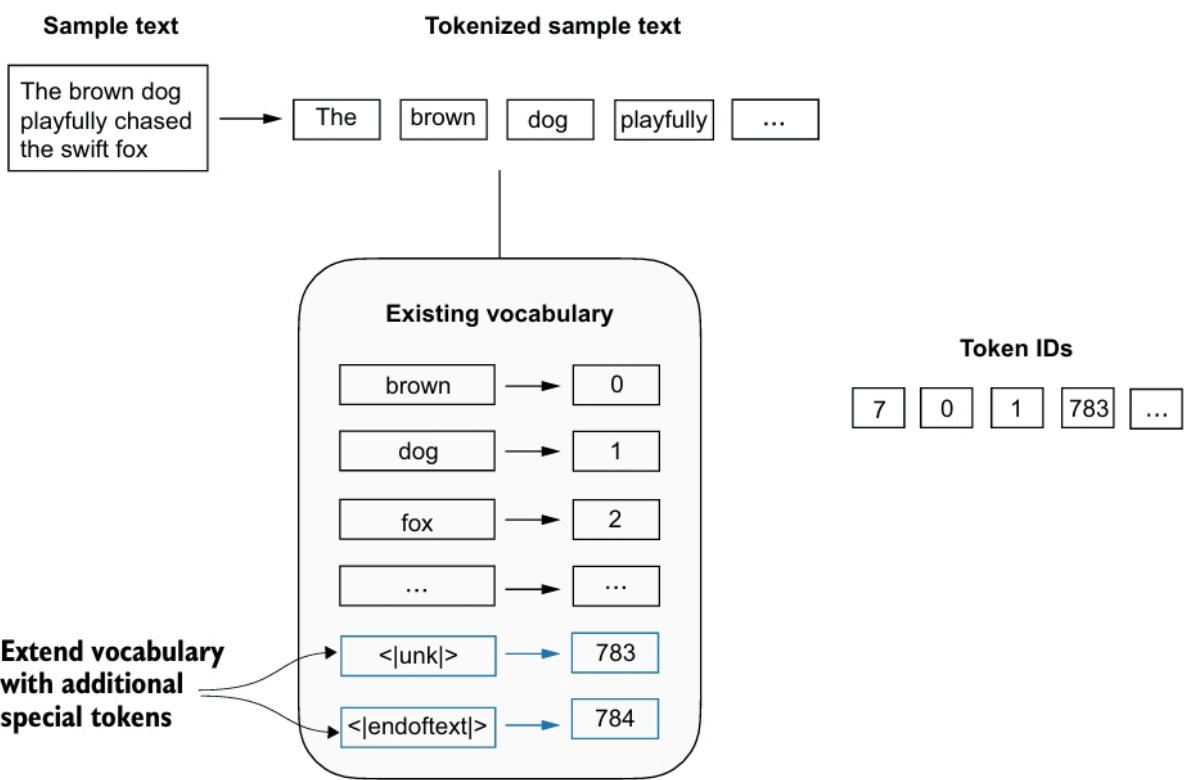


Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an `<| unk |>` token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an `<| endoftext |>` token that we can use to separate two unrelated text sources.

我们可以修改分词器，使其在遇到不属于词汇表的单词时使用`<| unk |>`分词。此外，我们还会在不相关的文本之间添加分词。

例如，在多个独立文档或书籍上训练类似 GPT 的 LLM 时，通常会在每个跟在前一个文本源之后的文档或书籍前插入一个分词，如图 2.10 所示。这有助于 LLM 理解，虽然这些文本源在训练时被连接在一起，但它们实际上是不相关的。

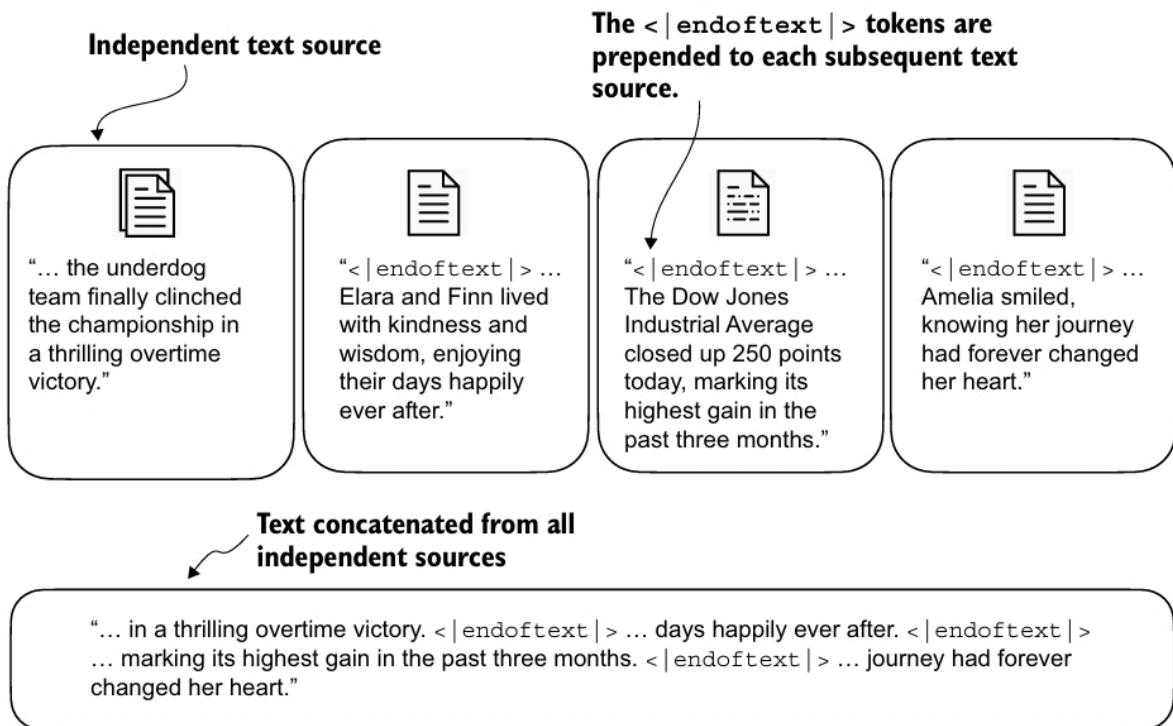


Figure 2.10 When working with multiple independent text source, we add <| endoftext |> tokens between these texts. These <| endoftext |> tokens act as markers, signaling the start or end of a particular segment, allowing for more effective processing and understanding by the LLM.

现在让我们修改词汇表，将这两个特殊token 和<|endoftext|> 添加到所有唯一单词的列表中：

```

1 all_tokens = sorted(list(set(preprocessed)))
2
3 all_tokens.extend(["<|endoftext|>", "<|unk|>"])
4
5 vocab = {token:integer for integer,token in enumerate(all_tokens)}
6
7 print(len(vocab.items()))

```

加入两个特殊标记

```

all_tokens = sorted(list(set(preprocessed)))

all_tokens.extend(["<|endoftext|>", "<|unk|>"])

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

print(len(vocab.items()))

```

1132

根据此打印语句的输出，新的词汇量为 1,132（之前的词汇量为 1,130）。为了进一步快速检查，我们打印更新后的词汇表的最后五个条目：

```

1 for i, item in enumerate(list(vocab.items())[-5:]):
2
3     print(item)

```

查看加入的两个特殊标记

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

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

根据代码输出，我们可以确认这两个新的特殊tokens确实已成功合并到词汇表中。接下来，我们相应地调整代码清单 2.3 中的tokenizer，如下清单所示。

Listing 2.4 A simple text tokenizer that handles unknown words

```
1 class SimpleTokenizerV2:
2     def __init__(self, vocab):
3         self.str_to_int = vocab # 将词汇表存储为类属性，以便在编码和解码方法中访问
4         self.int_to_str = {i:s for s,i in vocab.items()} # 创建一个逆向词汇表，
将token ID 映射回原始文本 tokens
5         # 将输入文本处理成token IDs
6         def encode(self, text):
7             # 分词
8             preprocessed = re.split(r'([.,?_!"()\\]|--|\\s)', text)
9             # 处理空格
10            preprocessed = [
11                item.strip() for item in preprocessed if item.strip()
12            ]
13            # 将未知单词替换为 <|unk|>字符
14            preprocessed = [item if item in self.str_to_int
15                           else "<|unk|>" for item in preprocessed]
16            # 使用词汇表将tokens映射为token IDs
17            ids = [self.str_to_int[s] for s in preprocessed]
18            return ids
19
20            # 将token IDs映射回文本
21            def decode(self, ids):
22                # 将ids中的tokenIDs通过词汇表反向映射并连接起来
23                text = " ".join([self.int_to_str[i] for i in ids])
24                # 删除指定标点符号前的空格
25                text = re.sub(r'\\s+([.,;?!"()\\'])', r'\\1', text)
26                return text
```

与清单 2.3 中实现的 SimpleTokenizerV1 相比，新的 SimpleTokenizerV2 会用 <|unk|> token 替换未知单词。

现在让我们在实践中尝试一下这个新的tokenizer。为此，我们使用一个简单的文本示例，该示例由两个独立且不相关的句子连接而成：

```
1 | text1 = "Hello, do you like tea?"  
2 | text2 = "In the sunlit terraces of the palace."  
3 | text = " <|endoftext|> ".join((text1, text2))  
4 | print(text)  
5 |  
6 | tokenizer = SimpleTokenizerV2(vocab)  
7 |  
8 | print(tokenizer.encode(text))
```

测试新的tokenizer

```
text1 = "Hello, do you like tea?"  
text2 = "In the sunlit terraces of the palace."  
text = " <|endoftext|> ".join((text1, text2))  
print(text)
```

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

```
tokenizer = SimpleTokenizerV2(vocab)  
print(tokenizer.encode(text))
```

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

我们可以看到，token ID 列表中包含一个 1130 token（用于 <|endoftext|> 分隔符），以及两个 1131 token（用于表示未知单词）。

让我们对文本进行解码处理，以便进行快速的完整性检查：

```
1 | print(tokenizer.decode(tokenizer.encode(text)))
```

```
print(tokenizer.decode(tokenizer.encode(text)))  
<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of the <|unk|>.
```

通过将此解码后的文本与原始输入文本进行比较，我们知道，训练数据集（伊迪丝·华顿的短篇小说《判决》）不包含“Hello”和“palace”这两个词。

根据LLM的具体情况，一些研究人员还会考虑其他特殊token，例如：

[BOS] (beginning of sequence) — 此token标记文本的开始。它向 LLM 指示一段内容的开始位置。
[EOS] (end of sequence) — 此token位于文本末尾，在连接多个不相关的文本时尤其有用，类似于 <|endoftext|>。例如，在合并两篇不同的维基百科文章或书籍时，[EOS] token指示一个文本的结束位置和下一个文本的开始位置。

[PAD] (padding) — 当训练批量大小大于 1 的 LLM 时，批量可能包含不同长度的文本。为了确保所有文本的长度相同，较短的文本会使用 [PAD] token 进行扩展或“填充”，直到达到批量中最长文本的长度。

GPT 模型使用的分词器不需要任何这些 token；为了简单起见，它只使用一个 <|endoftext|> token。<|endoftext|> 类似于 [EOS] token。

<|endoftext|> 也用于填充。然而，正如我们将在后续章节中探讨的那样，在对批量输入进行训练时，我们通常使用掩码，这意味着我们不会关注填充的 token。因此，选择用于填充的具体 token 变得无关紧要。

此外，GPT 模型使用的分词器也不会对词汇表外的单词使用 <|unk|> token。相反，GPT 模型**使用字节对编码分词器**，它将单词分解为子词单元。

2.5 字节对编码 (Byte pair encoding)

让我们看一下基于字节对编码 (BPE) 概念的更复杂的token化方案。BPE tokenizer用于训练 GPT-2、GPT-3 等 LLM，以及 ChatGPT 中使用的原始模型。

由于实现 BPE 相对复杂，我们将使用一个现有的 Python 开源库 tiktoken (<https://github.com/openai/tiktoken>)，它基于 Rust 源代码高效地实现了 BPE 算法。与其他 Python 库类似，我们可以通过 Python 的 pip 安装程序从终端安装 tiktoken 库：

```
1 | pip install tiktoken
```

我们将使用的代码基于 tiktoken 0.7.0。您可以使用以下代码检查您当前安装的版本：

```
1 | from importlib.metadata import version
2 | import tiktoken
3 | print("tiktoken version:",version("tiktoken"))
```

安装完成后，我们可以从 tiktoken 实例化 BPE tokenizer，如下所示：

```
1 | tokenizer = tiktoken.get_encoding("gpt2")
```

这个 tokenizer 的使用方法和我们之前通过 encode 方法实现的 SimpleTokenizerV2 类似：

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

然后，我们可以使用解码方法将 token ID 转换回文本，类似于我们的 SimpleTokenizerV2：

```
1 | strings = tokenizer.decode(integers)
2 | print(strings)
```

安装tiktoken包，该包基于RUST算法实现了BPE算法

```
[1]: !pip install tiktoken
Collecting tiktoken
  Downloading tiktoken-0.9.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (6.7 kB)
Requirement already satisfied: regex>=2022.1.18 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from tiktoken) (2025.7.34)
Requirement already satisfied: requests>=2.26.0 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from tiktoken) (2.32.4)
Requirement already satisfied: charset_normalizer<4,>=2.5 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from requests>=2.26.0->tiktoken) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from requests>=2.26.0->tiktoken) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from requests>=2.26.0->tiktoken) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /home/share/anaconda/envs/zxy/lib/python3.12/site-packages (from requests>=2.26.0->tiktoken) (2025.8.3)
  Downloading tiktoken-0.9.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.2 MB)
                                         1.2/1.2 MB 6.2 MB/s eta 0:00:00:--:--
Installing collected packages: tiktoken
Successfully installed tiktoken-0.9.0

引入包，并检查版本

[3]: from importlib.metadata import version
import tiktoken
print("tiktoken version:",version("tiktoken"))

tiktoken version: 0.9.0

使用tiktoken包实例化BPE tokenizer

[4]: tokenizer = tiktoken.get_encoding("gpt2")

将文本编码为Token IDs

[5]: text = ("Hello, do you like tea? <|endoftext|> In the sunlit terraces"      "of someunknownPlace." )
integers = tokenizer.encode(text, allowed_special="(<|endoftext|>)")
print(integers)

[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812, 2114, 1659, 617, 34680, 27271, 13]

将Token IDs解码为文本

[6]: strings = tokenizer.decode(integers) ***

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

根据 token ID 和解码后的文本，我们可以得出两个值得注意的观察结果。

首先，<|endoftext|> token 被分配了一个相对较大的 token ID，即 50256。事实上，用于训练 GPT-2、GPT-3 等模型以及 ChatGPT 中使用的原始模型的 BPE tokenizer 的总词汇量为 50,257，其中，<|endoftext|> 被分配了最大的 token ID。

其次，BPE 分词器能够正确地对未知词（例如 someunknownPlace）进行编码和解码。BPE 分词器可以处理任何未知词。它是如何做到不使用 <|unk|> 分词器的呢？

BPE 的底层算法将不在其预定义词汇表中的单词分解成更小的子词单元，甚至是单个字符，从而使其能够处理词汇表之外的单词。因此，得益于 BPE 算法，如果分词器在分词过程中遇到不熟悉的单词，它可以将其表示为一系列子词token或字符，如图 2.11 所示。

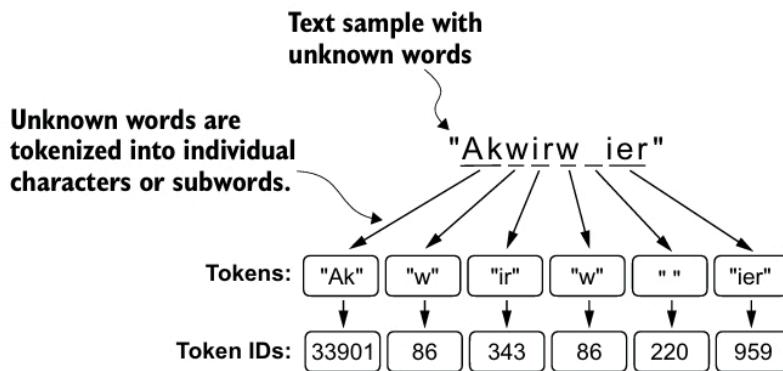


Figure 2.11 BPE tokenizers break down unknown words into subwords and individual characters. This way, a BPE tokenizer can parse any word and doesn't need to replace unknown words with special tokens, such as <|unk|>.

将未知单词分解成单个字符的能力确保了tokenizer以及用它训练的 LLM 能够处理任何文本，即使它包含训练数据中不存在的单词。

练习 2.1 未知单词的字节对编码

尝试使用 tiktoken 库中的 BPE 分词器对未知单词“Akwirw iер”进行分词，并打印各个分词 ID。然后，对列表中的每个整数调用解码函数，以重现图 2.11 所示的映射。最后，对分词 ID 调用解码方法，检查它是否可以重建原始输入“Akwirw iер”。

```

text2 = "Akwirw ier"
ids = tokenizer.encode(text2, allowed_special=["<|endoftext|>"])
print(ids)

[33901, 86, 343, 86, 220, 959]

for i in ids:
    print(tokenizer.decode([i]))

Ak
w
ir
w

ier

tokenizer.decode(ids)

'Akwirw ier'

```

BPE 的详细讨论和实现超出了本书的范围，但简而言之，它通过迭代地将高频字符合并为子词，再将高 频子词合并为词来构建词汇表。例如，BPE 首先将所有单个字符（“a”、“b”等）添加到其词汇表中。在下一阶段，它将频繁出现的字符组合合并为子词。例如，“d”和“e”可以合并为子词“de”，这在许多英语单词 中很常见，例如“define”、“depend”、“made”和“hidden”。合并由频率截止值决定。

2.6 使用滑动窗口进行数据采样

为 LLM 创建嵌入的下一步是生成训练 LLM 所需的输入-目标对。这些输入-目标对是什么样的？正如我们 已经了解的，LLM 是通过预测文本中的下一个单词来进行预训练的，如图 2.12 所示。

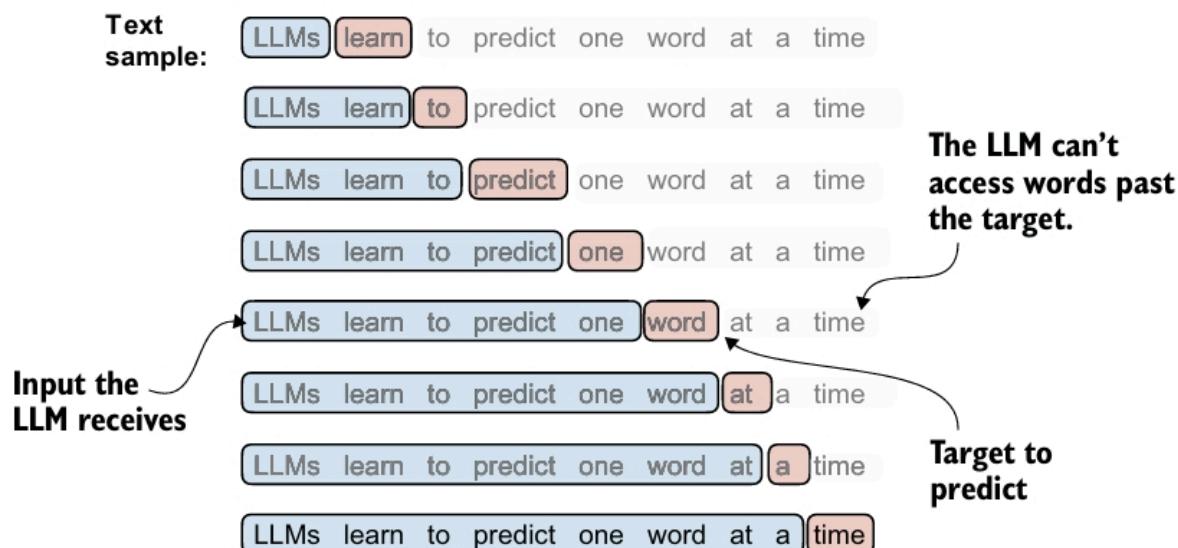


Figure 2.12 Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM’s prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure must undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.

让我们实现一个数据加载器，它使用滑动窗口方法从训练数据集中获取图 2.12 中的输入-目标对。首先， 我们将使用 BPE 分词器对整个短篇小说《判决》进行分词：

```
1 | with open("the-verdict.txt", "r", encoding="utf-8") as f:  
2 |     raw_text = f.read()  
3 |  
4 |     enc_text = tokenizer.encode(raw_text)  
5 |     print(len(enc_text))
```

使用 BPE 分词器对整个短篇小说《判决》进行分词

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:  
    raw_text = f.read()  
  
    enc_text = tokenizer.encode(raw_text)  
    print(len(enc_text))
```

5145

接下来，为了演示的目的，我们从数据集中删除了前 50 个tokens，因为这会在接下来的步骤中产生一个更有趣的文本段落(?)

```
1 | enc_sample = enc_text[50:]
```

为下一个单词预测任务创建输入-目标对的最简单、最直观的方法之一是创建两个变量 x 和 y，其中 x 包含输入token，y 包含目标（即移位 1 的输入）：

```
1 | context_size = 4  
2 | x = enc_sample[:context_size]  
3 | y = enc_sample[1:context_size+1]  
4 | print(f"x: {x}")  
5 | print(f"y: \t{y}")
```

为了演示的目的，从数据集中删除前 50 个标记

```
enc_sample = enc_text[50:]
```

创建输入-目标对

```
# context_size 决定输入中要包含多少个tokens  
context_size = 4  
x = enc_sample[:context_size]  
y = enc_sample[1:context_size+1]  
print(f"x: {x}")  
print(f"y: \t{y}")  
  
x: [290, 4920, 2241, 287]  
y:      [4920, 2241, 287, 257]
```

通过处理输入以及目标（将输入移动一个位置），我们可以创建下一个单词预测任务（见图 2.12），如下所示：

创建下一个单词预测任务

```
for i in range(1, context_size+1):
    context = enc_sample[:i] # 上下文
    desire = enc_sample[i] # 预测目标
    print(context,"---->", desire)

[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

箭头 (---->) 左侧的所有内容均表示 LLM 将接收的输入，而箭头右侧的 token ID 表示 LLM 应该预测的目标 token ID。让我们重复前面的代码，但将 token ID 转换为文本：

将token IDs转换为文本

```
for i in range(1, context_size+1):
    context = enc_sample[:i] # 上下文
    desire = enc_sample[i] # 预测目标
    print(tokenizer.decode(context),"---->", tokenizer.decode([desire]))

and ----> established
and established ----> himself
and established himself ----> in
and established himself in ----> a
```

在将token转换为嵌入之前，只剩下一项任务：实现一个高效的数据加载器，该加载器迭代输入数据集，并将输入和目标返回为 PyTorch 张量（可以将其视为多维数组）。具体来说，我们感兴趣的是返回两个张量：一个包含 **LLM 看到的文本的输入张量**，以及一个包含**LLM 需要预测的目标张量**，如图 2.13 所示。虽然该图以字符串格式显示了token，但为了便于说明，代码实现将直接对token ID 进行操作，因为 BPE tokenizer 的 encode 方法将token化和转换为token ID 合并为一个步骤。

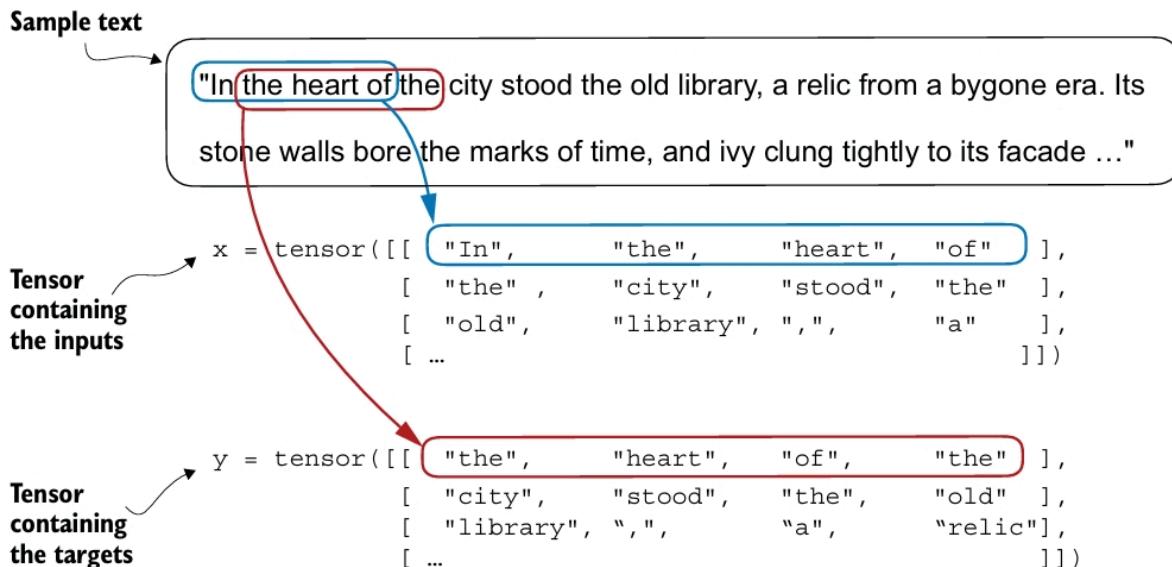


Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, `x`, where each row represents one input context. A second tensor, `y`, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.

注意：为了实现高效的数据加载器，我们将使用 PyTorch 内置的 Dataset 和 DataLoader 类。

Listing 2.5 批量输入和目标的数据集

```

1 import torch
2 from torch.utils.data import Dataset, DataLoader
3
4 class GPTDatasetV1(Dataset):
5     def __init__(self, txt, tokenizer, max_length, stride):
6         self.input_ids = []
7         self.target_ids = []
8         token_ids = tokenizer.encode(txt) # 编码整个文本
9         # 循环遍历数据，使用滑动窗口将数据分成重叠的最大长度序列
10        # 假设token_ids的长度大于max_length
11        for i in range(0, len(token_ids) - max_length, stride):
12            # 输入的区域是i到i+maxLength, 左闭右开，因此最后一个token不参与输入
13            input_chunk = token_ids[i:i + max_length]
14            # 第一个token不作为预测目标
15            target_chunk = token_ids[i+1:i + max_length + 1]
16            self.input_ids.append(torch.tensor(input_chunk))
17            self.target_ids.append(torch.tensor(target_chunk))
18        # 返回数据集的行数
19        def __len__(self):
20            return len(self.input_ids)
21        # 返回数据集中单独一行数据
22        def __getitem__(self, idx):
23            return self.input_ids[idx], self.target_ids[idx]

```

GPTDatasetV1 类基于 PyTorch Dataset 类，定义了如何从数据集中提取各个行，其中每行由分配给 input_chunk 张量的多个 token ID（基于 max_length）组成。target_chunk 张量包含相应的目标。我建议继续阅读，看看当我们将数据集与 PyTorch DataLoader 结合使用时，从该数据集返回的数据是什么样子——这将带来额外的直观和清晰度。

以下代码使用 GPTDatasetV1 通过 PyTorch DataLoader 批量加载输入

Listing 2.6 数据加载器，用于生成具有输入对的批次

```

1 import tiktoken
2 def create_dataloader_v1(txt,
3                         batch_size=4,
4                         max_length=256,
5                         stride=8,
6                         shuffle=True,
7                         drop_last=True,
8                         num_workers=0):
9     tokenizer = tiktoken.get_encoding("gpt2") # 初始化tokenizer
10    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride) # 创建数据集
11    dataloader = DataLoader(
12        dataset,
13        batch_size=batch_size,
14        shuffle=shuffle,
15        drop_last=drop_last, # drop_last=True时，如果最后一个批次小于指定的
batch_size，则丢弃它，以防止训练期间出现损失峰值。
16        num_workers=num_workers # 用于预处理的 CPU 进程数
17    )
18    return dataloader

```

让我们针对上下文大小为 4 的 LLM 测试批大小为 1 的数据加载器，以直观地了解清单 2.5 中的 GPTDatasetV1 类和清单 2.6 中的 create_dataloader_v1 函数如何协同工作：

```
1 with open("the-verdict.txt", "r", encoding="utf-8") as f:  
2     raw_text = f.read()  
3  
4 dataloader = create_dataloader_v1(raw_text, batch_size=1, max_length=4,  
5 stride=1, shuffle=False)  
6  
7 data_iter = iter(dataloader) # 将数据加载器转换为 Python 迭代器，以便通过 Python 内  
8 置的 next() 函数获取下一个条目  
9 first_batch = next(data_iter)  
10 print(first_batch)
```

测试Dataset和DataLoader

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:  
    raw_text = f.read()  
  
dataloader = create_dataloader_v1(raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)  
  
data_iter = iter(dataloader) # 将数据加载器转换为 Python 迭代器，以便通过 Python 内置的 next() 函数获取下一个条目  
first_batch = next(data_iter)  
print(first_batch)  
  
[tensor([[ 40, 367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

first_batch 变量包含两个张量：第一个张量存储输入 token ID，第二个张量存储目标 token ID。由于 max_length 设置为 4，因此两个张量每个都包含 4 个 token ID。请注意，输入大小 4 非常小，仅为了简单起见而选择。通常训练 LLM 时，输入大小至少为 256。

为了理解 stride=1 的含义，让我们从该数据集中获取另一个批次：

```
1 second_batch = next(data_iter)  
2 print(second_batch)
```

如果我们比较第一个批次和第二个批次，我们可以看到第二个批次的 tokenID 移动了一个位置（例如，第一个批次输入中的第二个 ID 是 367，这是第二个批次输入的第一个 ID）。步幅设置决定了输入在各个批次之间移动的位置数，模拟了滑动窗口方法，如图 2.14 所示。

```
[tensor([[ 40, 367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

理解 stride=1 的含义

```
second_batch = next(data_iter)  
print(second_batch)  
  
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

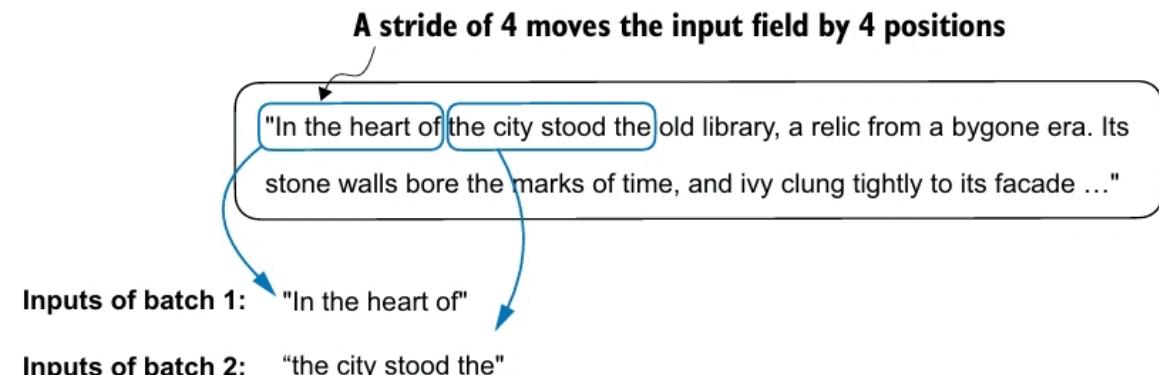
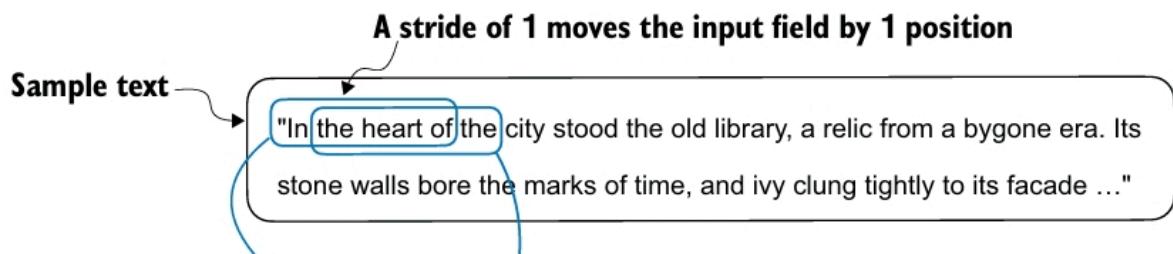


Figure 2.14 When creating multiple batches from the input dataset, we slide an input window across the text. If the stride is set to 1, we shift the input window by one position when creating the next batch. If we set the stride equal to the input window size, we can prevent overlaps between the batches.

练习 2.2 具有不同步长和上下文大小的数据加载器

为了更直观地了解数据加载器的工作原理，请尝试使用不同的设置运行它，例如 max_length=2 和 stride=2，以及 max_length=8 和 stride=2。

练习 2.2 具有不同步长和上下文大小的数据加载器 为了更直观地了解数据加载器的工作原理，请尝试使用不同的设置运行它，例如 max_length=2 和 stride=2，以及 max_length=8 和 stride=2。

```
dataloader2_2 = create_dataloader_v1(raw_text, batch_size=1, max_length=2, stride=2, shuffle=False)

data_iter2 = iter(dataloader2_2)
first_batch2 = next(data_iter2)
print(first_batch2)
second_batch2 = next(data_iter2)
print(second_batch2)

[tensor([[ 40, 367]]), tensor([[ 367, 2885]])]
[tensor([[2885, 1464]]), tensor([[1464, 1807]])]

dataloader8_2 = create_dataloader_v1(raw_text, batch_size=1, max_length=8, stride=2, shuffle=False)

data_iter3 = iter(dataloader8_2)
first_batch3 = next(data_iter3)
print(first_batch3)
second_batch3 = next(data_iter3)
print(second_batch3)

[tensor([[ 40, 367, 2885, 1464, 1807, 3619, 402, 271]]), tensor([[ 367, 2885, 1464, 1807, 3619, 402, 271, 10899]])]
[tensor([[ 2885, 1464, 1807, 3619, 402, 271, 10899, 2138]]), tensor([[ 1464, 1807, 3619, 402, 271, 10899, 2138, 257]])]
```

批次大小为 1（例如我们目前为止从数据加载器中采样的批次大小）对于演示目的很有用。如果您之前有深度学习经验，您可能知道，较小的批次大小在训练期间所需的内存较少，但会导致模型更新噪声更大。与常规深度学习一样，批次大小是一个权衡，也是训练 LLM 时需要试验的超参数。

让我们简单看一下如何使用数据加载器进行批量大小大于 1 的采样：

```
1 | dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=4,  
2 |     stride=4, shuffle=False)  
3 | data_iter = iter(dataloader) # 将数据加载器转换为 Python 迭代器，以便通过 Python 内  
4 |     置的 next() 函数获取下一个条目  
5 | inputs, targets = next(data_iter)  
6 | print("Inputs:\n", inputs)  
7 | print("\nTargets:\n", targets)
```

```
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=4, stride=4, shuffle=False)  
data_iter = iter(dataloader) # 将数据加载器转换为 Python 迭代器，以便通过 Python 内置的 next() 函数获取下一个条目  
inputs, targets = next(data_iter)  
print("Inputs:\n", inputs)  
print("\nTargets:\n", targets)  
  
Inputs:  
tensor([[ 40,   367,  2885, 1464],  
       [1807,  3619,   402,   271],  
       [10899, 2138,   257,  7026],  
       [15632,   438, 2016,   257],  
       [ 922,  5891, 1576,   438],  
       [ 568,   340,   373,   645],  
       [1049,  5975,   284,   502],  
       [ 284,  3285,   326,    11]])  
  
Targets:  
tensor([[ 367,  2885, 1464, 1807],  
       [ 3619,   402,   271, 10899],  
       [ 2138,   257,  7026, 15632],  
       [ 438, 2016,   257,   922],  
       [ 5891, 1576,   438,   568],  
       [ 340,   373,   645, 1049],  
       [ 5975,   284,   502,   284],  
       [ 3285,   326,    11,  287]])
```

请注意，我们将步长增加到 4，以充分利用数据集（我们不会跳过任何一个单词）。这避免了批次之间的任何重叠，因为过多的重叠可能会导致过拟合加剧。

2.7 创建token嵌入

准备 LLM 训练输入文本的最后一步是将 token ID 转换为嵌入向量，如图 2.15 所示。作为准备步骤，我们必须用随机值初始化这些嵌入权重。此初始化是 LLM 学习过程的起点。在第五章中，我们将在 LLM 训练过程中优化嵌入权重。

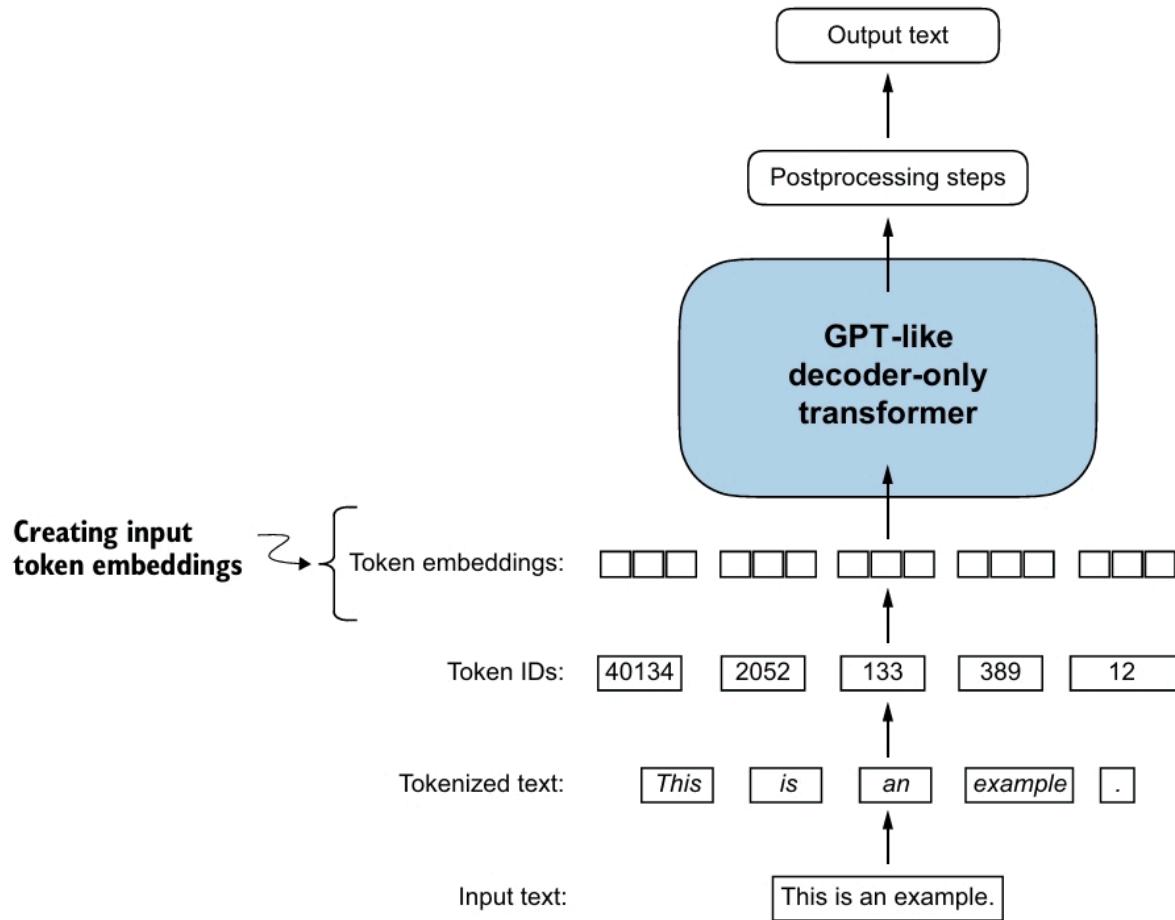


Figure 2.15 Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.

由于类似 GPT 的 LLM 是使用反向传播算法训练的深度神经网络，因此连续向量表示（或嵌入）是必需的。

让我们通过一个实际示例来了解 token ID 到嵌入向量的转换是如何进行的。假设我们有以下四个输入 token，ID 分别为 2、3、5 和 1：

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

为了简单起见，假设我们的词汇表只有**6 个单词**（而不是 BPE tokenizer 词汇表中的 50,257 个单词），并且我们想要创建大小为**3 维**的嵌入（在 GPT-3 中，嵌入大小为 12,288 维）：

```
1 | vocab_size = 6
2 |
3 | output_dim = 3
```

使用 `vocab_size` 和 `output_dim`，我们可以在 PyTorch 中实例化一个嵌入层，并将随机种子设置为 123 以提高可重复性

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

打印语句打印嵌入层的底层权重矩阵：

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

vocab_size = 6
output_dim = 3

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

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

嵌入层的权重矩阵包含较小的随机值。这些值在 LLM 训练期间会作为 LLM 优化本身的一部分进行优化。此外，我们可以看到权重矩阵有六行三列。词汇表中六个可能的tokens各占一行，三个嵌入维度各占一列。

现在，让我们将其应用到token ID 以获取嵌入向量：

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

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

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

如果我们将 token ID 为 3 的嵌入向量与前一个嵌入矩阵进行比较，我们会发现它与第四行相同（Python 从零索引开始，因此它是与索引 3 对应的行）。换句话说，**嵌入层本质上是一个查找操作，它通过 token ID 从嵌入层的权重矩阵中检索行。**

对于熟悉独热编码的人来说，本文描述的嵌入层方法本质上只是一种更高效的实现方式，即在全连接层中先进行独热编码，然后再进行矩阵乘法，其示例代码位于 GitHub 上的 <https://mng.bz/ZEB5>。由于嵌入层只是一种更高效的实现方式，相当于独热编码和矩阵乘法，因此可以将其视为一个可以通过反向传播进行优化的神经网络层。

我们已经了解了如何将单个 token ID 转换为三维嵌入向量。现在让我们将其应用于所有四个输入 ID (torch.tensor([2, 3, 5, 1]))。

```
1 | print(embedding_layer(input_ids))
```

将嵌入层应用于所有四个输入ID

```
print(embedding_layer(input_ids))

tensor([[ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-2.8400, -0.7849, -1.4096],
       [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

如图 2.16 所示，此输出矩阵中的每一行都是通过对嵌入权重矩阵进行查找操作获得的。

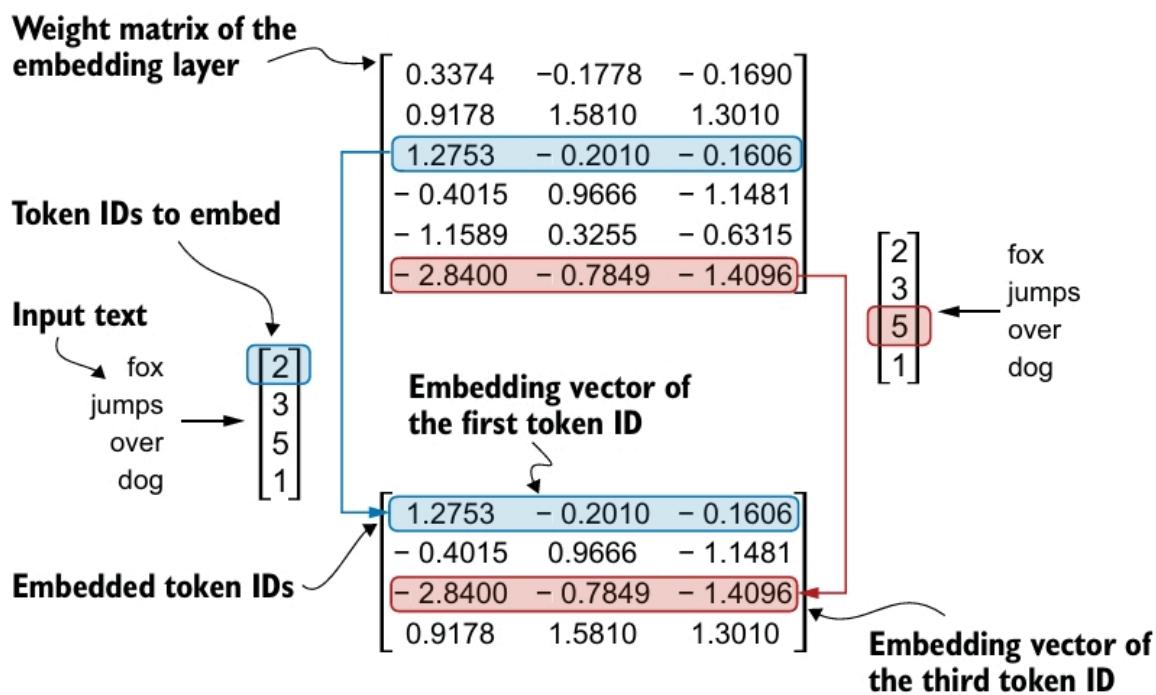


Figure 2.16 Embedding layers perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer’s weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth row because Python starts counting at 0). We assume that the token IDs were produced by the small vocabulary from section 2.3.

现在我们根据token ID 创建了嵌入向量，接下来我们将对这些嵌入向量进行一些小的修改，以便对文本中token的位置信息进行编码。

2.8 编码词位置 (Encoding word positions)

原则上，token 嵌入 (token embedding) 是 LLM 的合适输入。然而，LLM 的一个小缺点是，它们的自注意力机制（参见第 3 章）**没有序列中词法单元的位置或顺序的概念**。之前介绍的嵌入层的工作方式是，相同的词法单元 ID 总是映射到相同的向量表示，无论该词法单元 ID 在输入序列中位于何处，如图 2.17 所示。

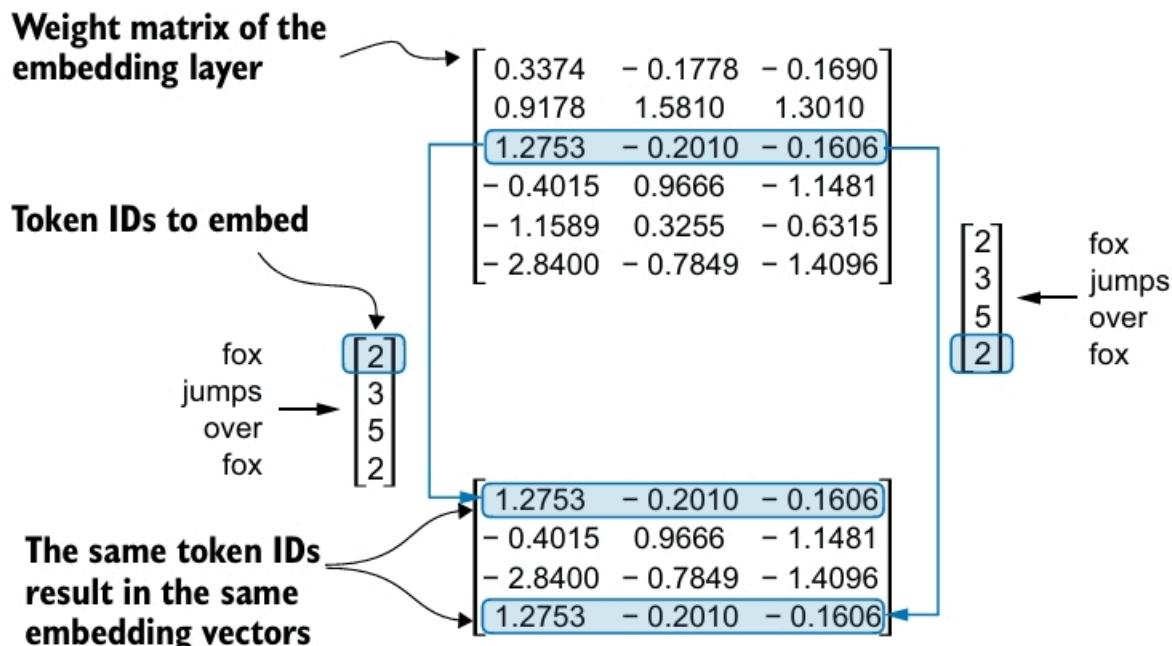


Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or fourth position in the token ID input vector, will result in the same embedding vector.

原则上，确定性且与位置无关的 token ID 嵌入有利于提高可复现性。然而，由于 LLM 本身的自注意力机制也与位置无关，因此在 LLM 中注入额外的位置信息会有所帮助。

为了实现这一点，我们可以使用两大类位置感知嵌入：相对位置嵌入和绝对位置嵌入。绝对位置嵌入与序列中的特定位置直接关联。对于输入序列中的每个位置，都会将一个唯一的位置嵌入添加到token的嵌入中，以传达其确切位置。例如，第一个token将具有特定的位置嵌入，第二个token将具有另一个不同的嵌入，依此类推，如图 2.18 所示。

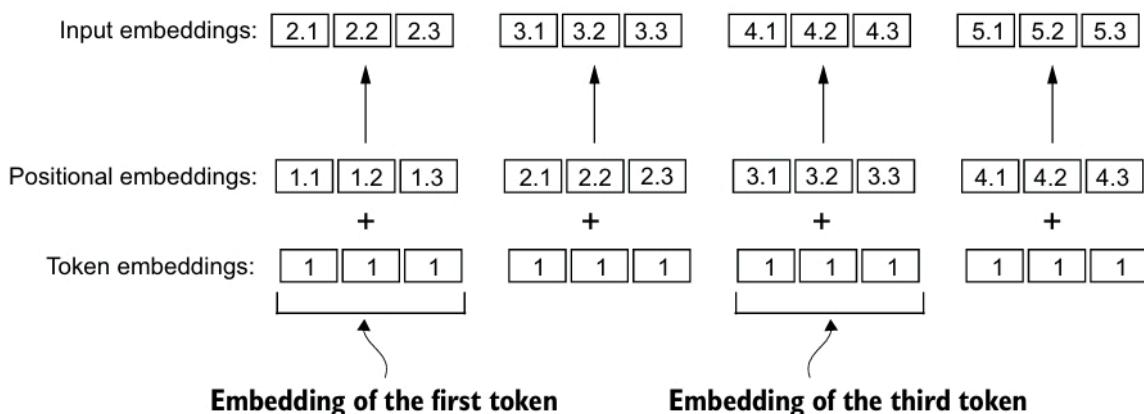


Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.

相对位置嵌入的重点并非关注token的绝对位置，而是token之间的相对位置或距离。这意味着模型学习的是“相距多远”而不是“具体在哪个位置”的关系。这样做的好处是，即使模型在训练过程中没有遇到过这种长度的序列，它也能更好地泛化到不同长度的序列。

两种类型的位置嵌入都旨在增强 LLM 理解 token 之间顺序和关系的能力，从而确保更准确、更符合上下文的预测。两者之间的选择通常取决于具体的应用和所处理数据的性质。

OpenAI 的 GPT 模型使用绝对位置嵌入，这些嵌入在训练过程中进行优化，而不是像原始 Transformer 模型中的位置编码那样固定或预定义。此优化过程是模型训练本身的一部分。现在，让我们创建初始位置嵌入来创建 LLM 输入。

之前，为了简单起见，我们专注于非常小的嵌入大小。现在，让我们考虑更现实、更实用的嵌入大小，并将输入 token 编码为 256 维向量表示，这比原始 GPT-3 模型使用的向量表示要小（在 GPT-3 中，嵌入大小为 12,288 维），但对于实验来说仍然合理。此外，我们假设 token ID 是由我们之前实现的 BPE 分词器创建的，其词汇量为 50,257：

```
1 | vocab_size = 50257
2 | output_dim = 256
3 | token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

使用之前的 token_embedding_layer，如果我们从 DataLoader 中采样数据，我们会将每个批次中的每个 token 嵌入到一个 256 维向量中。如果批次大小为 8，每个批次包含 4 个 token，则结果将是一个 $8 \times 4 \times 256$ 的张量。我们首先实例化数据加载器（参见 2.6 节）：

```
1 | max_length = 4
2 | dataloader = create_dataloader_v1(
3 |     raw_text, batch_size=8, max_length=max_length,
4 |     stride=max_length, shuffle=False
5 | )
6 | data_iter = iter(dataloader)
7 | inputs, targets = next(data_iter)
8 | print("Token IDs:\n", inputs)
9 | print("\nInputs shape:\n", inputs.shape)
```

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

```
Token IDs:
tensor([[ 40,   367, 2885, 1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438, 2016,   257],
        [ 922,  5891, 1576,   438],
        [ 568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [ 284,  3285,   326,    11]])
```

```
Inputs shape:
torch.Size([8, 4])
```

我们可以看到，token ID 张量是 8×4 维的，这意味着数据批次由 8 个文本样本组成，每个样本有 4 个 token。

现在，我们使用嵌入层将这些 token ID 嵌入到 256 维向量中：

```
1 | token_embeddings = token_embedding_layer(inputs)
2 | print(token_embeddings.shape)
```

将tokenIDs嵌入到256维向量中

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)

torch.Size([8, 4, 256])
```

$8 \times 4 \times 256$ 维张量输出表明，每个 token ID 现在都被嵌入为一个 256 维向量。

对于 GPT 模型的绝对嵌入方法，我们只需要创建另一个与 token_embedding_layer 具有相同嵌入维度的嵌入层（即 position_embedding_layer）：

```
1 | context_length = max_length
2 | pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
3 | pos_embeddings = pos_embedding_layer(torch.arange(context_length))
4 | print(pos_embeddings.shape)
```

pos_embeddings 的输入通常是一个占位符向量 `torch.arange(context_length)`，它包含一个数字序列 $0, 1, \dots$ ，直到最大输入长度 -1。context_length 是一个变量，表示 LLM 支持的输入大小。在这里，我们选择它的大小与输入文本的最大长度相似。

实际上，输入文本可能比支持的上下文长度更长，在这种情况下，我们必须截断文本。

`print` 语句的输出为: `torch.Size([4, 256])`

我们看到，位置嵌入张量由四个 256 维向量组成。现在我们可以将它们直接添加到 token 嵌入中，PyTorch 会将 4×256 维的 pos_embeddings 张量添加到八个批次中每个 4×256 维的 token 嵌入张量中：

```
1 | input_embeddings = token_embeddings + pos_embeddings
2 | print(input_embeddings.shape)
```

实现绝对位置嵌入

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

torch.Size([4, 256])

input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)

torch.Size([8, 4, 256])
```

如图 2.19 所示，我们创建的 input_embeddings 是嵌入的输入示例，现在可以由主要的 LLM 模块处理，我们将在下一章开始实现这些模块。

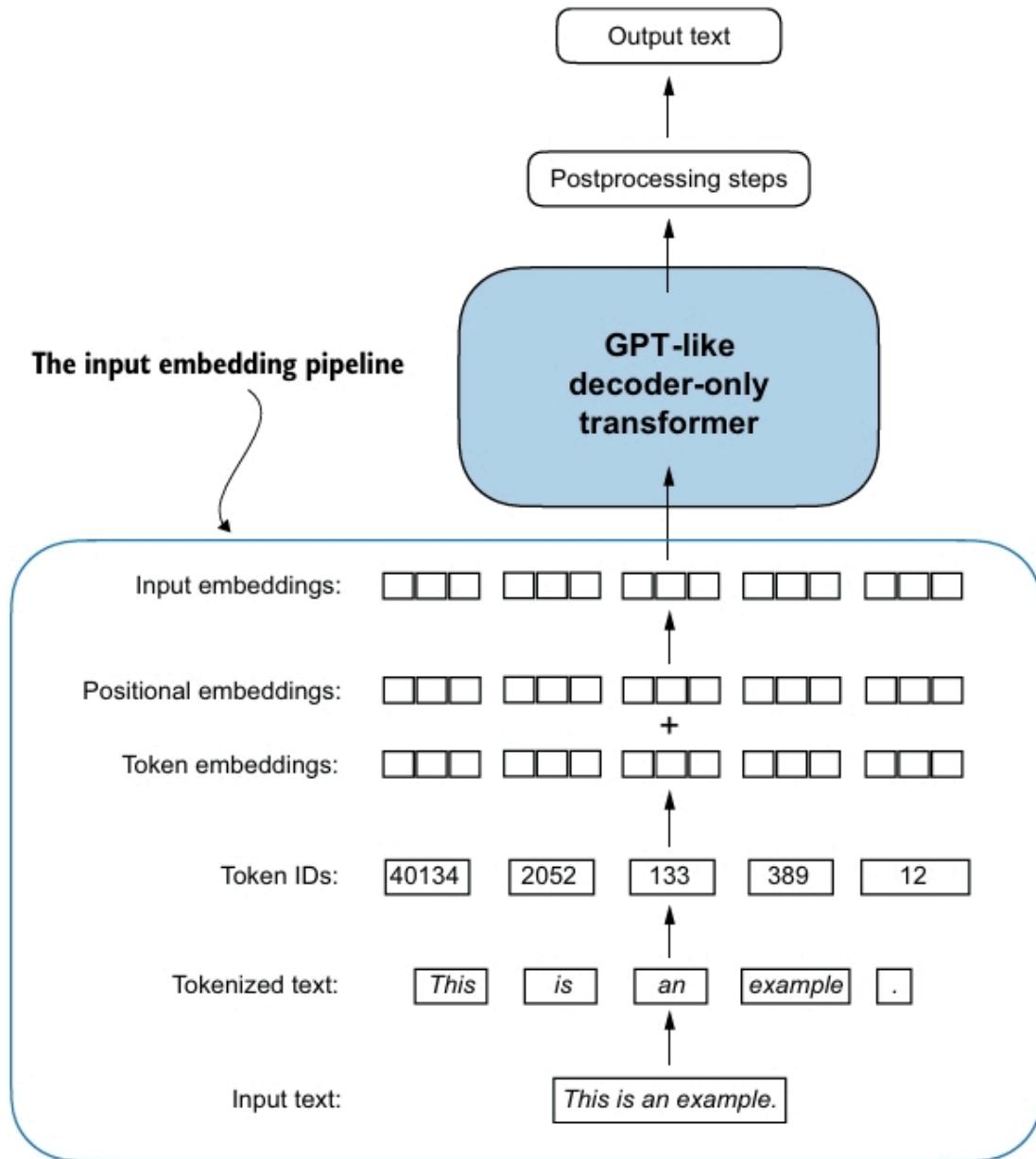


Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are used as input for the main LLM layers.

总结

- 由于 LLM 无法处理原始文本，因此它们需要将文本数据转换为数值向量，即嵌入向量。嵌入向量将离散数据（例如单词或图像）转换为连续向量空间，使其与神经网络操作兼容。
- 第一步是将原始文本分解为token，这些token可以是单词或字符。然后，这些token被转换为整数表示，称为token ID。
- 可以添加特殊token，例如`<|unk|>`和`<|endoftext|>`，以增强模型的理解能力并处理各种上下文，例如未知单词或token不相关文本之间的边界。
- 用于 GPT-2 和 GPT-3 等 LLM 的字节对编码 (BPE) tokenizer 可以通过将未知单词分解为子字单元或单个字符来有效地处理它们。
- 我们对编码后的的数据使用滑动窗口方法生成用于 LLM 训练的输入-目标对。

- PyTorch 中的**嵌入层充当查找操作，检索与token ID 对应的向量**。生成的**嵌入向量提供token的连续表示**，这对于训练 LLM 等深度学习模型至关重要。
- 虽token嵌入为每个token提供了一致的向量表示，但它们**缺乏对token在序列中位置的感知**。为了解决这个问题，存在两种主要类型的位置嵌入：绝对嵌入和相对嵌入。**OpenAI 的GPT 模型使用绝对位置嵌入，这些嵌入被添加到token嵌入向量中，并在模型训练期间进行优化**。