2 Working with text data

This chapter covers

- Preparing text for large language model training
- Splitting text into word and subword tokens
- Byte pair encoding as a more advanced way of tokenizing text
- Sampling training examples with a sliding window approach
- Converting tokens into vectors that feed into a large language model

So far, we've covered the general structure of large language models (LLMs) and learned that they are pretrained on vast amounts of text. Specifically, our focus was on decoder-only LLMs based on the transformer architecture, which underlies the models used in ChatGPT and other popular GPT-like LLMs.

During the pretraining stage, LLMs process text one word at a time. Training LLMs with millions to billions of parameters using a next-word prediction task yields models with impressive capabilities. These models can then be further fine-tuned to follow general instructions or perform specific target tasks. But before we can implement and train LLMs, we need to prepare the training dataset, as illustrated in figure 2.1.

2 与文本数据工作

本章涵盖

- 准备文本用于大型语言模型训练
- 将文本分割成单词和子词标记
- 字节对编码作为更高级的文本分词方式
- 使用滑动窗口方法采样训练示例
- 将标记转换为输入大型语言模型的向量

截至目前，我们已经介绍了大型语言模型（LLMs）的一般结构，并了解到它们是在大量文本上预训练的。具体来说，我们的重点是仅基于 Transformer 架构的解码器（LLMs），这是 ChatGPT 和其他流行 GPT-like 模型（LLMs）所使用的模型的基础。

在预训练阶段，LLMs逐词处理文本。使用数百万到数十亿参数进行下一词预测任务训练LLMs，可以得到具有令人印象深刻能力的模型。然后，这些模型可以进一步微调以遵循一般指令或执行特定目标任务。但在我们能够实现和训练LLMs之前，我们需要准备训练数据集，如图 2.1 所示。

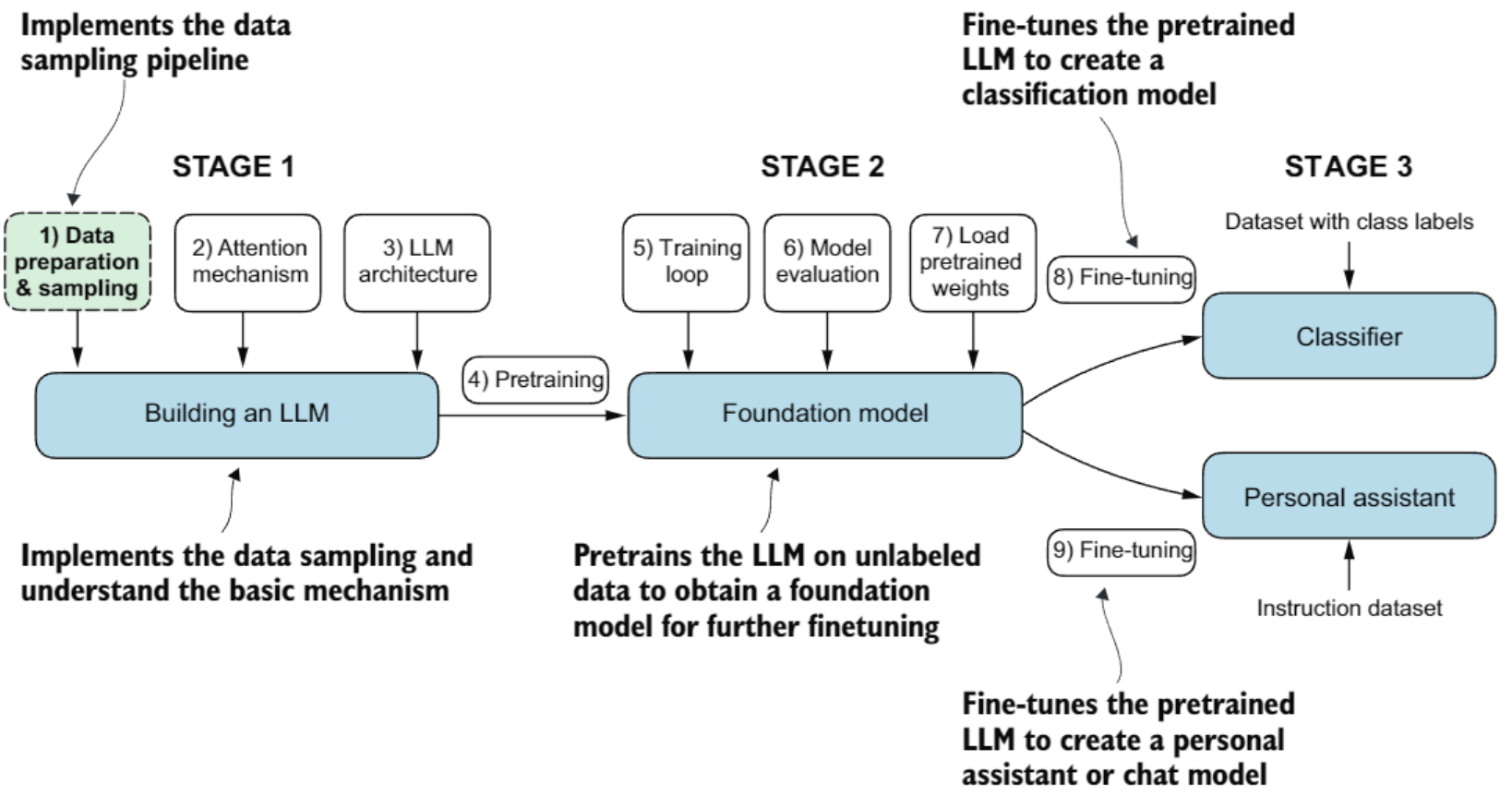

Figure 2.1 The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.

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.

2.1 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.

> **NOTE** Readers unfamiliar with vectors and tensors in a computational context can learn more in appendix A, section A.2.2.

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, as illustrated in figure 2.2. 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.

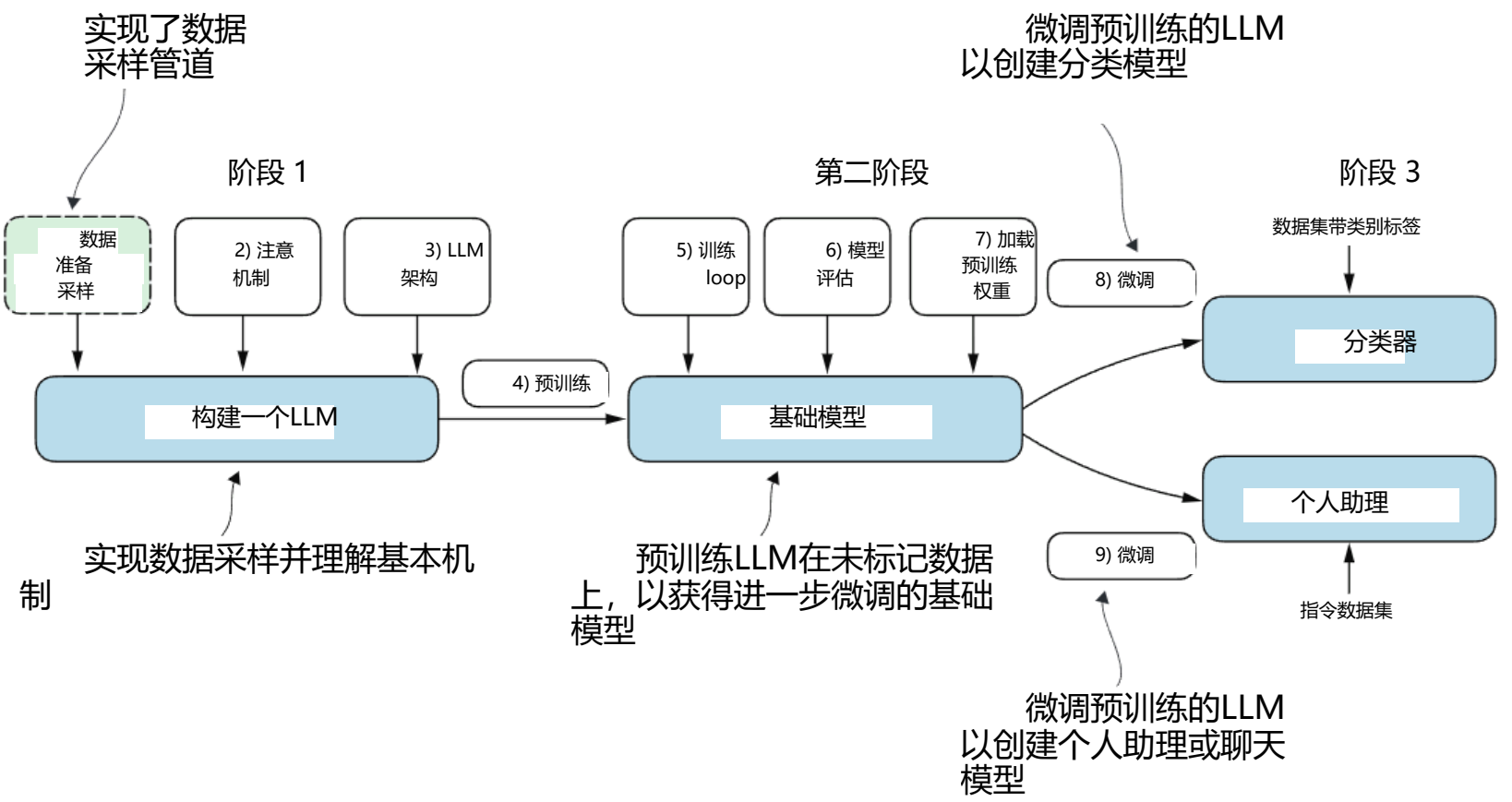

图 2.1 LLM 编码的三个主要阶段。本章重点介绍第一阶段步骤 1：实现数据样本管道。

您将学习如何为训练准备输入文本LLMs。这涉及到将文本拆分为单个单词和子词标记，然后可以将其编码为向量表示LLM。您还将了解高级标记化方案，如字节对编码，这在流行的LLMs（如 GPT）中得到了应用。最后，我们将实施采样和数据加载策略，以生成训练所需的输入-输出对LLMs。

2.1 理解词嵌入

深度神经网络模型，包括LLMs，不能直接处理原始文本。由于文本是分类的，它不兼容用于实现和训练神经网络的数学运算。因此，我们需要一种方法将单词表示为连续值向量。

注意：不熟悉计算环境中向量和张量的读者可以在附录 A 的第 A.2.2 节中了解更多信息。

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另一个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。

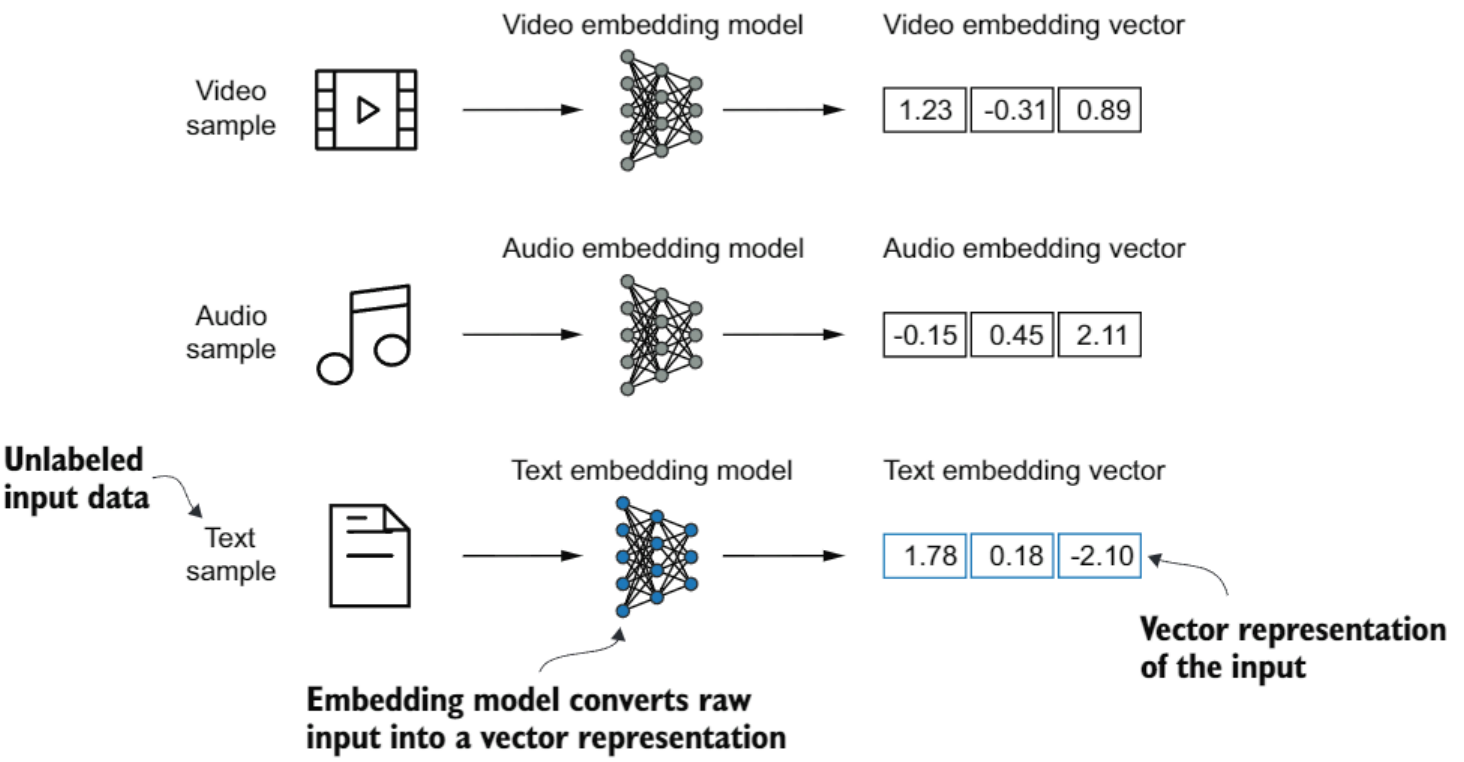

Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates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

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 nonnumeric 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, which is a technique that is beyond the scope of this book. Since our goal is to train GPT-like LLMs, which learn to generate text one word at a time, we will focus on word embeddings.

Several algorithms and frameworks have been developed to generate word embeddings. One of the earlier and most popular examples is the *Word2Vec* approach. Word2Vec trained neural network architecture to generate 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 two-dimensional word embeddings for visualization purposes, similar terms are clustered together, as shown in figure 2.3.

Word embeddings can have varying dimensions, from one to thousands. A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

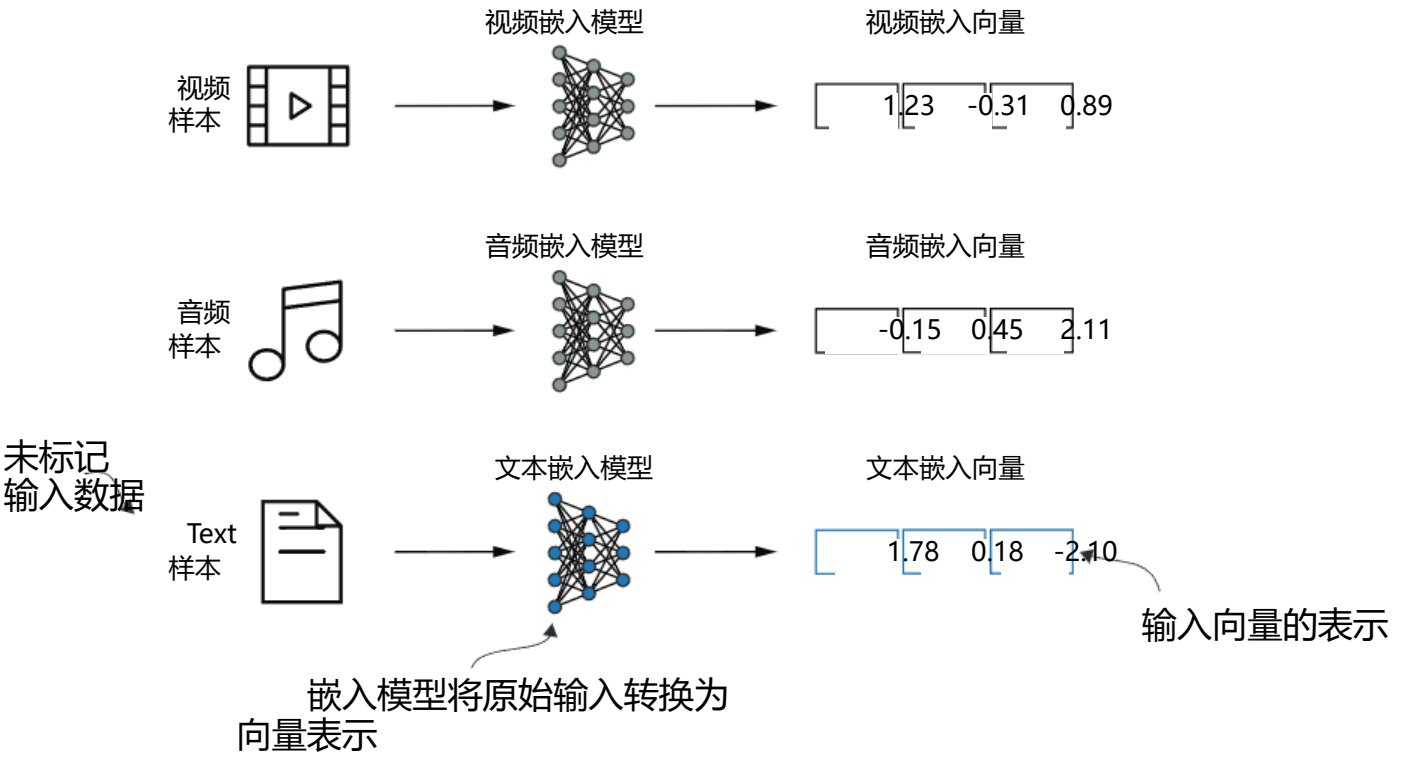

图 2.2 深度学习模型无法直接处理视频、音频和文本等原始数据格式。因此，我们使用嵌入模型将原始数据转换为深度学习架构可以轻松理解和处理的密集向量表示。具体来说，此图说明了将原始数据转换为三维数值向量的过程。

在核心上，嵌入是将离散对象（如单词、图像甚至整个文档）映射到连续向量空间中的点的一种映射——嵌入的主要目的是将非数值数据转换为神经网络可以处理的形式。

虽然词嵌入是最常见的文本嵌入形式，但也有句子、段落或整个文档的嵌入。句子或段落嵌入是检索增强生成中的流行选择。检索增强生成结合生成（如产生文本）和检索（如搜索外部知识库）以在生成文本时提取相关信息，这是一种超出本书范围的技术。由于我们的目标是训练类似 GPT 的LLMs，它们学会逐词生成文本，因此我们将专注于词嵌入。

多个算法和框架已被开发用于生成词嵌入。其中一个较早且最受欢迎的例子是 Word2Vec 方法。Word2Vec 通过预测目标词或反之的上下文来训练神经网络架构以生成词嵌入。Word2Vec 背后的主要思想是出现在相似上下文中的词往往具有相似的含义。因此，当为了可视化目的投影到二维词嵌入时，相似术语会聚集在一起，如图 2.3 所示。

词嵌入的维度可以从一到数千不等。更高的维度可能能够捕捉更细微的关系，但以计算效率为代价。

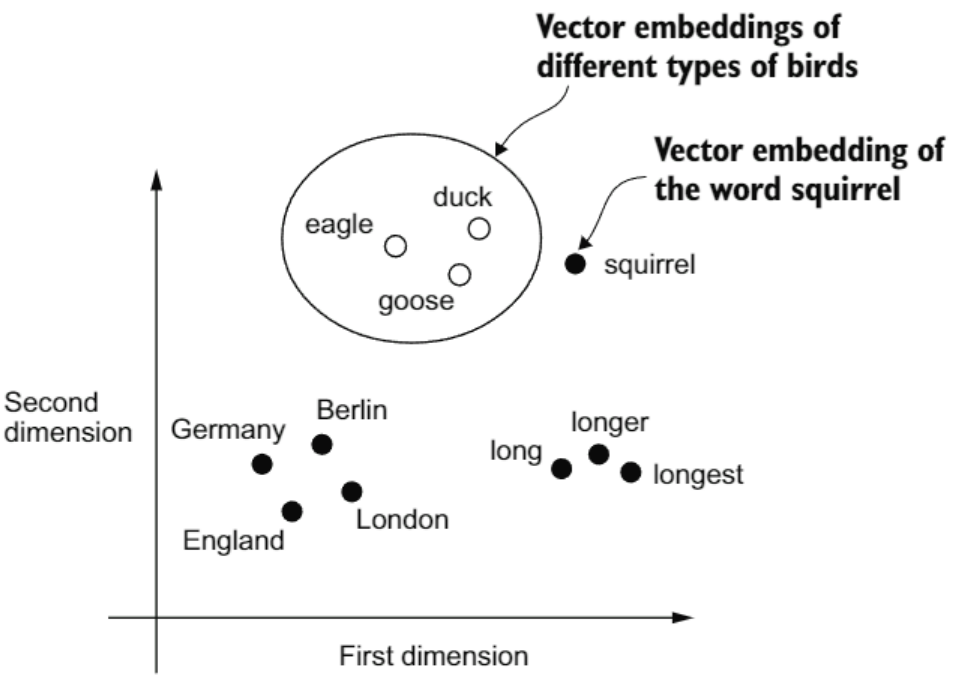

Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance,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

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

Unfortunately, high-dimensional embeddings present a challenge for visualization because our sensory perception and common graphical representations are inherently limited to three dimensions or fewer, which is why figure 2.3 shows two-dimensional embeddings in a two-dimensional scatterplot. However, when working with LLMs, we typically use embeddings with a much higher dimensionality. For both GPT-2 and GPT-3, the embedding size (often referred to as the dimensionality of the model's hidden states) varies based on the specific model variant and size. It is a tradeoff between performance and efficiency. The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

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

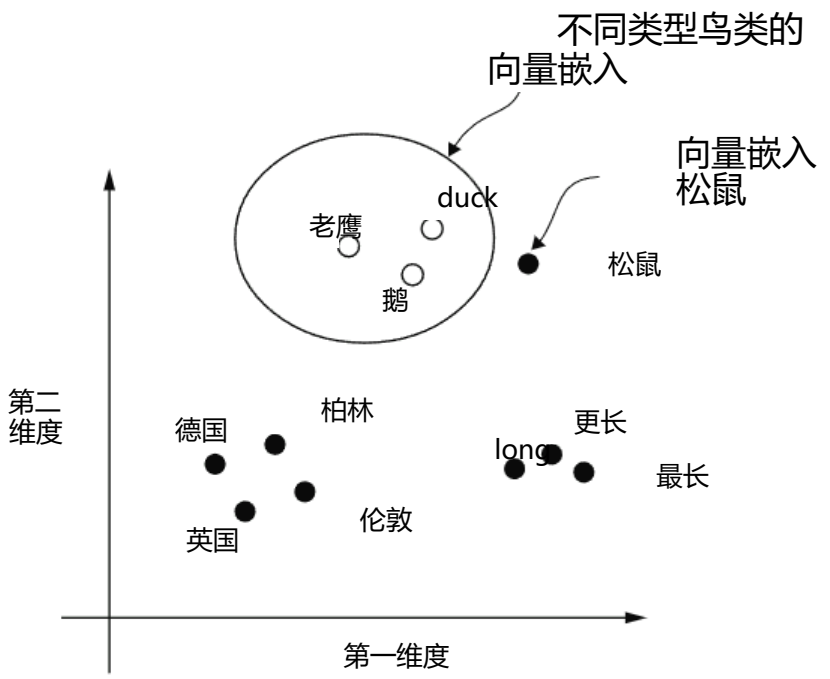

图 2.3 如果词嵌入是二维的，我们可以将它们绘制在二维散点图中进行可视化，如图所示。在使用词嵌入技术，如 Word2Vec 时，表示相似概念的词在嵌入空间中通常彼此靠近。例如，不同种类的鸟在嵌入空间中比在国家和城市中更靠近。

虽然我们可以使用预训练模型如 Word2Vec 为机器学习模型生成嵌入，但LLMs通常会产生自己的嵌入，这些嵌入是输入层的一部分，并在训练过程中更新。将嵌入作为LLM训练的一部分进行优化的优点是，嵌入被优化以适应特定任务和数据。我们将在本章后面实现这样的嵌入层。（LLMs还可以创建上下文化的输出嵌入，如我们在第 3 章中讨论的。）不幸的是，高维嵌入对可视化构成了挑战，因为我们的感官感知和常见的图形表示本质上限于三维或更少，这就是为什么图 2.3 在二维散点图中显示了二维嵌入。然而，当我们处理LLMs时，我们通常使用具有更高维度的嵌入。对于 GPT-2 和 GPT-3，嵌入大小（通常被称为模型隐藏状态的维度）根据特定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（117M 和 125M 参数）使用 768 维的嵌入大小来提供具体示例。最大的 GPT-3 模型（175B 参数）使用 12,288 维的嵌入大小。

接下来，我们将介绍用于LLM的嵌入准备所需的步骤，包括将文本拆分为单词、将单词转换为标记以及将标记转换为嵌入向量。

2.2 Tokenizing text

Let's discuss 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, as shown in figure 2.4.

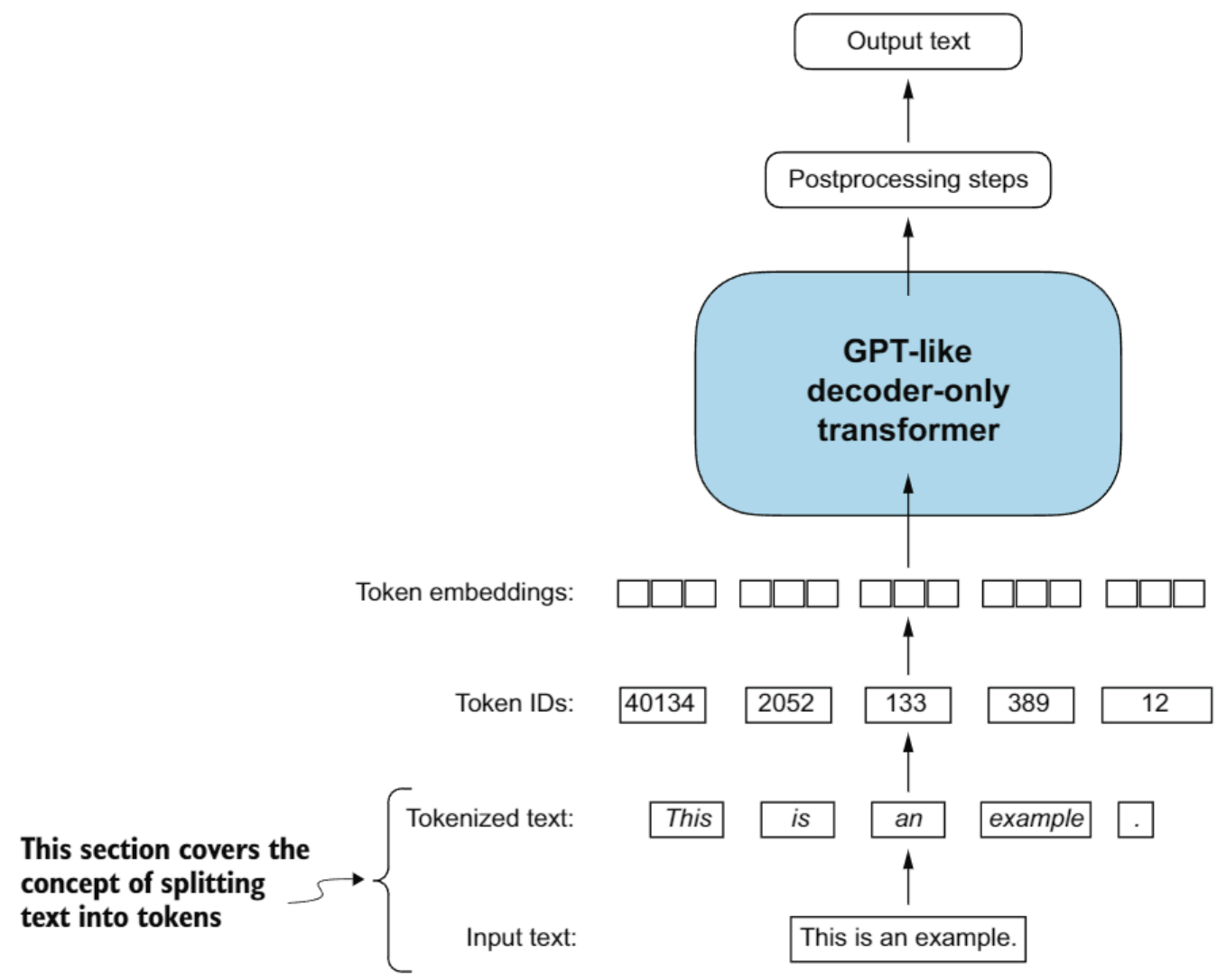

Figure 2.4 A view of the text processing steps in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters.

The text we will tokenize for LLM training is "The Verdict," a short story by Edith Wharton, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict, and you can copy and paste it into a text file, which I copied into a text file `"the-verdict.txt"`.

Alternatively, you can find this `"the-verdict.txt"` file in this book's GitHub repository at https://mng.bz/Adng. You can download the file with the following Python code:

2.2 分词文本

让我们讨论如何将输入文本分割成单个标记，这是为LLM创建嵌入所需的预处理步骤。这些标记可以是单个单词或特殊字符，包括标点符号，如图 2.4 所示。

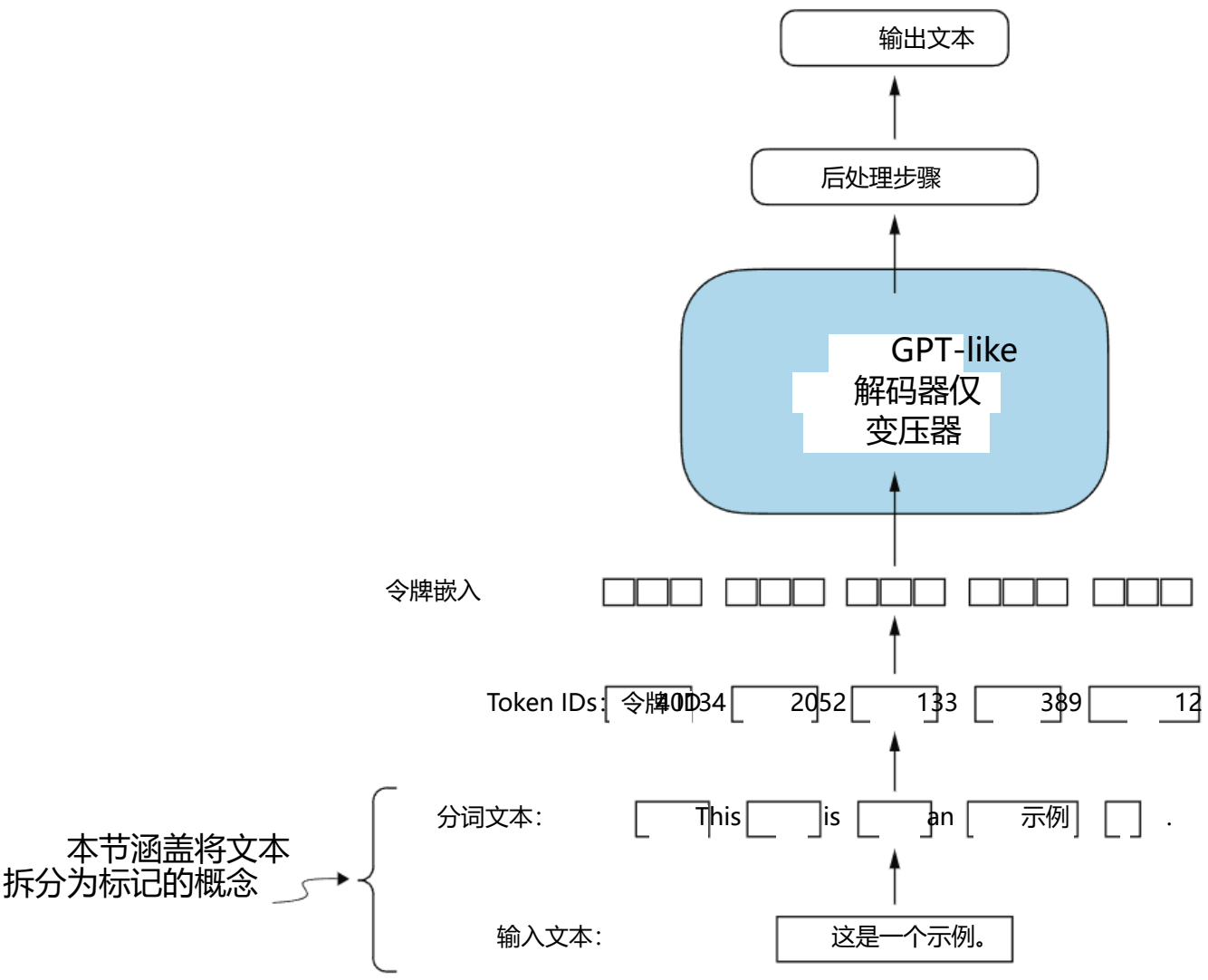

图 2.4 在LLM的上下文中，对文本处理步骤的视图。在此，我们将输入文本分割成单个标记，这些标记可以是单词或特殊字符，例如标点符号。

我们将对用于LLM训练的文本进行分词，“The Verdict”，这是伊迪丝·华顿的一篇短篇小说，已被发布到公有领域，因此可以用于LLM训练任务。该文本可在维基源上找到，网址为 https://en.wikisource.org/wiki/The_Verdict，您可以将其复制并粘贴到文本文件中，我已经复制了

将文本文件“the-verdict.txt”中。

另外，您可以在本书的 GitHub 仓库中找到此 "the-verdict.txt" 文件，网址为 https://mng.bz/Adng。您可以使用以下 Python 代码下载该文件：

```
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)
```

Next, we can load the `the-verdict.txt` file using Python's standard file reading utilities.

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

```
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])
```

The print command prints the total number of characters followed by the first 100 characters of this file for illustration purposes:

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

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

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

How can we best split this text to obtain a list of tokens? For this, we go on a small excursion and use Python's regular expression library `re` for illustration purposes. (You don't have to learn or memorize any regular expression syntax since we will later transition to a prebuilt tokenizer.)

Using some simple example text, we can use the `re.split` command with the following syntax to split a text on whitespace characters:

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

The result is a list of individual words, whitespaces, and punctuation characters:

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ', 'test.']
```

```
    import urllib.request
url (″https://raw.githubusercontent.com/rasbt/″ ″LLMs-from-
scratch/main/ch02/01_main-chapter-code/″ ″the-verdict.txt″) 文件路径 =
″the-verdict.txt″ urllib.request.urlretrieve(url, 文件路径)
```

接下来，我们可以使用 Python 的标准文件读取实用工具来加载 the-verdict.txt 文件。

列表 2.1 读取短篇故事作为文本样本到 Python

```
    with open("the-verdict.txt", "r", encoding="utf-8") as f:
打开文件 "the-verdict.txt" 以只读模式读取，编码为 utf-8，并将文件对象赋值给变量 f
    将一行文本作为纯文本输入，并将其翻译为简体中文，仅输出
翻译。如果某些内容无需翻译（如专有名词、代码等），则保持原文
不变。不要解释，输入文本:
```

打印命令打印出字符总数，之后为了说明目的，打印出该文件的前 100 个字符：

```
    总字符数：20479 我一直认为杰克·吉斯伯恩相当便宜的天才——尽管是个好人

        足够了--所以 it  was  no
```

我们的目标是把这个 20,479 个字符的短篇故事分解成单个单词和特殊字符，然后将其转换为LLM训练的嵌入。

注意：在使用LLMs时，处理数百万篇文章和数十万本书——数 GB 的文本是很常见的。然而，出于教育目的，使用单个书籍等较小的文本样本就足够了，以展示文本处理步骤背后的主要思想，并使其能够在消费级硬件上在合理的时间内运行。

如何最好地分割此文本以获得一个标记列表？为此，我们进行一次小旅行，并使用 Python 的正则表达式库 re 进行说明。（您不必学习或记住任何正则表达式语法，因为我们稍后将过渡到预构建的标记器。）

使用一些简单的示例文本，我们可以使用 re.split 命令，以下语法来在空白字符上拆分文本：

```
import re
你好，世界。这是，一个测试。
```

结果是一个包含单个单词、空白和标点符号的列表：

```
你好，       ’’,  世界。        ’,’这，’     ’ ’,’是’,  ’’,  ’a’,  ’ ’,  测试.]
```

This simple tokenization scheme mostly works for separating the example text into individual words; however, some words are still connected to punctuation characters that we want to have as separate list entries. We also refrain from making all text lowercase because capitalization helps LLMs distinguish between proper nouns and common nouns, understand sentence structure, and learn to generate text with proper capitalization.

Let's modify the regular expression splits on whitespaces (`\s`), commas, and periods (`[,.]`):

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

We can see that the words and punctuation characters are now separate list entries just as we wanted:

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', ' ', 'is',
' ', 'a', ' ', 'test', '.', '']
```

A small remaining problem is that the list still includes whitespace characters. Optionally, we can remove these redundant characters safely as follows:

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

The resulting whitespace-free output looks like as follows:

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

> **NOTE** When developing a simple tokenizer, whether we should encode whitespaces as separate characters or just remove them depends on our application and its requirements. Removing whitespaces reduces the memory and computing requirements. However, keeping whitespaces can be useful if we train models that are sensitive to the exact structure of the text (for example, Python code, which is sensitive to indentation and spacing). Here, we remove whitespaces for simplicity and brevity of the tokenized outputs. Later, we will switch to a tokenization scheme that includes whitespaces.

The tokenization scheme we devised here works well on the simple sample text. Let's modify it a bit further so that it can also handle other types of punctuation, such as question marks, quotation marks, and the double-dashes we have seen earlier in the first 100 characters of Edith Wharton's short story, along with additional special characters:

```
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)
```

这个简单的分词方案主要用于将示例文本分割成单个单词；然而，一些单词仍然与我们需要作为单独列表条目的标点符号相连。我们还避免将所有文本转换为小写，因为大写有助于LLMs区分专有名词和普通名词，理解句子结构，并学会生成正确大小写的文本。

让我们修改基于空白符（\s）、逗号和句号（[,.]）的正则表达式分割：

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

我们可以看到，单词和标点符号现在已经是分开的列表条目，正如我们想要的：

```
    你好，这是测试。
```

一个小问题仍然存在，即列表中仍然包含空白字符。可选地，我们可以安全地删除这些冗余字符，如下所示：

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

结果的无空格输出如下所示：

```
    你好          逗号，世界          这是，一个，测试，点。
```

注意：在开发简单的分词器时，我们是否应该将空白字符编码为单独的字符或只是简单地删除它们，这取决于我们的应用程序及其需求。删除空白字符可以减少内存和计算需求。然而，保留空白字符在训练对文本精确结构敏感的模型时可能很有用（例如，对缩进和间距敏感的 Python 代码）。在这里，我们为了简化并缩短分词输出的简洁性而删除空白字符。稍后，我们将切换到包括空白字符的分词方案。

我们设计的分词方案在简单的样本文本上效果良好。让我们进一步修改它，使其也能处理其他类型的标点符号，例如问号、引号以及我们在爱迪丝·华顿短篇小说前 100 个字符中看到的破折号，以及额外的特殊字符：

```
    你好，世界。这是——一个测试？ result = re.split(r'([,.:;?_!"()\']|-
-|\s)', text) result = [item.strip() for item in result if
item.strip()] print(result)
```

The resulting output is:

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

As we can see based on the results summarized in figure 2.5, our tokenization scheme can now handle the various special characters in the text successfully.

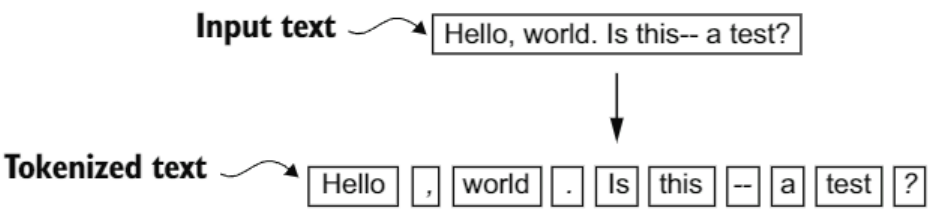

Figure 2.5 The tokenization scheme we implemented so far splits text into individual words and punctuation characters. In this specific example, the sample text gets split into 10 individual tokens.

Now that we have a basic tokenizer working, let's apply it to Edith Wharton's entire short story:

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

This print statement outputs `4690`, which is the number of tokens in this text (without whitespaces). Let's print the first 30 tokens for a quick visual check:

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

The resulting output shows that our tokenizer appears to be handling the text well since all words and special characters are neatly separated:

```
['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']
```

2.3 Converting tokens into token IDs

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

To map the previously generated tokens into token IDs, we have to build a vocabulary first. This vocabulary defines how we map each unique word and special character to a unique integer, as shown in figure 2.6.

结果输出为：

```
strip() 打印(result) ['Hello', ',', 'world这是', '。-', 'Is'] 'a', 测试，[? ]
```

如图 2.5 所示的结果所示，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。

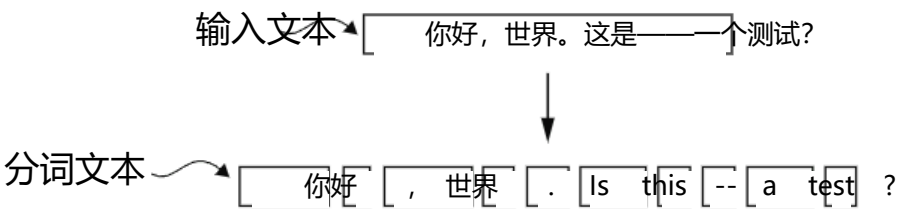

图 2.5 我们迄今为止实施的分词方案将文本分割成单个单词和标点符号。在这个特定示例中，样本文本被分割成 10 个单独的标记。

现在我们已经有一个基本的分词器在运行了，让我们将其应用于伊迪丝·华顿的整个短篇小说：

```
预处理的文本 = 使用正则表达式 re.split(r'([,.:;?_!"()\']|--|\s)', 原文本) 预处理的文本 =[去除空格的项 for 项 in 预处理的文本 if 去除空格的项] 打印(预处理的文本的长度)
```

这行打印语句输出 4690，这是文本中（不含空白符）的标记数。让我们打印前 30 个标记以进行快速视觉检查：

```
打印(preprocessed[:30])
```

结果输出显示，我们的分词器似乎处理文本得很好，因为所有单词和特殊字符都被整齐地分隔开：

```
我一直认为杰克·吉斯伯恩相当一个廉价的天才——尽管他是个足够好的家伙——所以听到他在，我并不感到惊讶，因为，在
```

2.3 将标记转换为标记 ID

接下来，我们将这些标记从 Python 字符串转换为整数表示，以生成标记 ID。此转换是在将标记 ID 转换为嵌入向量之前的中间步骤。

为了将之前生成的标记映射到标记 ID，我们首先需要构建一个词汇表。这个词汇表定义了如何将每个唯一的单词和特殊字符映射到一个唯一的整数，如图 2.6 所示。

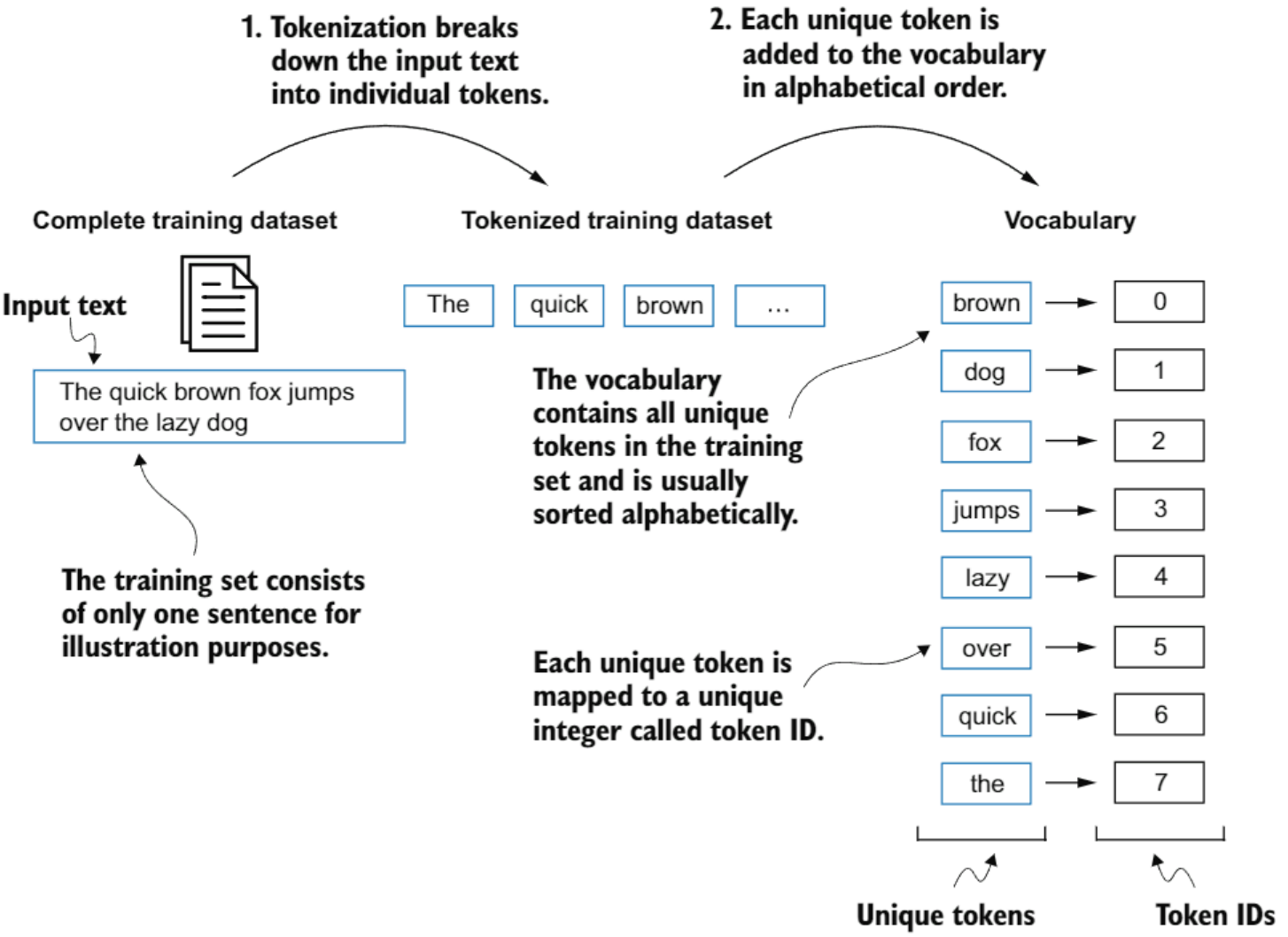

Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token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

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

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

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

Listing 2.2 Creating a vocabulary

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

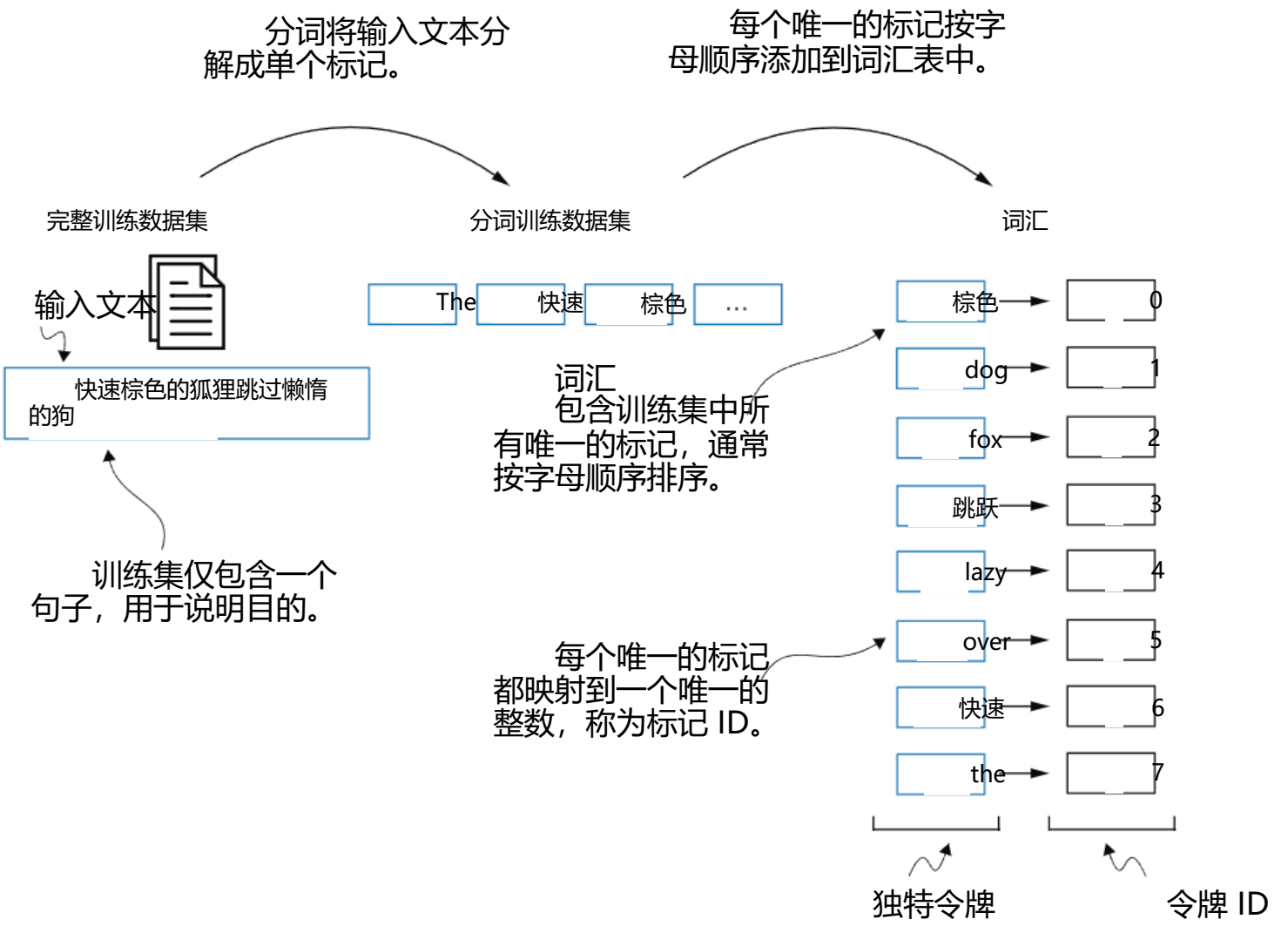

图 2.6 我们通过将训练数据集中的整个文本分词成单个标记来构建词汇表。然后，这些单个标记按字母顺序排序，并删除重复的标记。独特的标记随后汇总成一个词汇表，该词汇表定义了从每个独特标记到唯一整数值的映射。所展示的词汇表故意很小，且不包含标点符号或特殊字符，以保持简单。

现在我们已经对伊迪丝·华顿的短篇小说进行了分词，并将其分配给一个名为 preprocessed 的 Python 变量，接下来让我们创建一个包含所有唯一标记的列表，并按字母顺序排序，以确定词汇量大小：

```
    所有单词 = 对预处理的集合进行排序
vocab_size = 所有单词的长度 print(vocab_size)
```

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

列表 2.2 创建词汇表

```
    vocab = {token:整数 for 整数,token in enumerate(所有单词)} for i, 项目 in
enumerate(vocab.items()):
        打印(item)
        如果 i 大于等于 50:
            break
         中断
```

The output is

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

As we can see, the dictionary contains individual tokens associated with unique integer labels. Our next goal is to apply this vocabulary to convert new text into token IDs (figure 2.7).

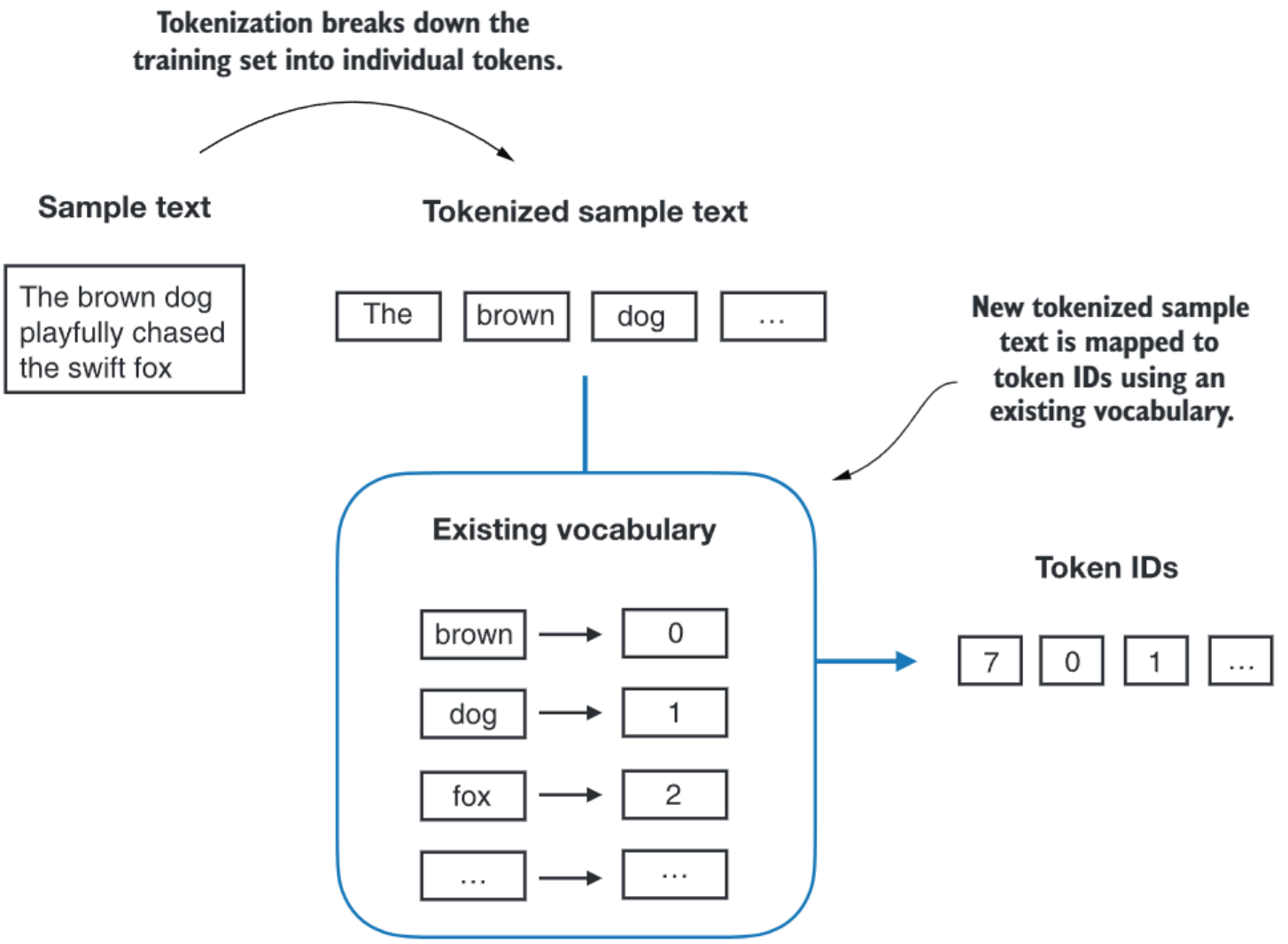

Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation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

When we want to convert the outputs of an LLM from numbers back into text, we need a way to turn token IDs into text. For this, we can create an inverse version of the vocabulary that maps token IDs back to the corresponding text tokens.

输出结果

```
('!', 0)
('“', 1)
（', 2）
...
（她，49）
赫尔米娅      50)
```

如我们所见，该词典包含与唯一整数标签关联的个体标记。我们的下一个目标是应用此词汇表将新文本转换为标记 ID（图 2.7）。

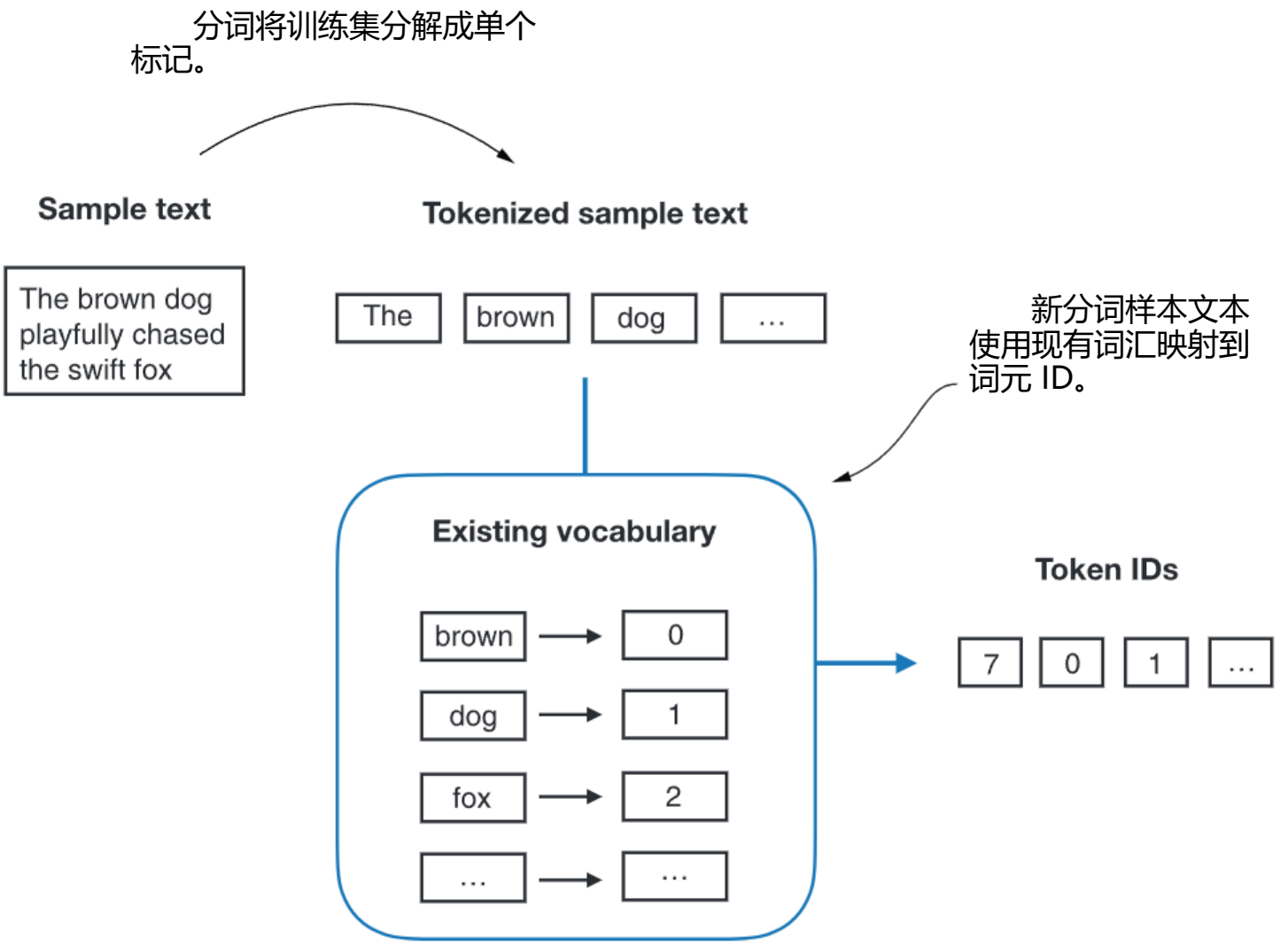

图 2.7 从一个新的文本样本开始，我们对文本进行分词，并使用词汇表将文本标记转换为标记 ID。词汇表由整个训练集构建而成，可以应用于训练集本身和任何新的文本样本。为了简化，所描述的词汇表中不包含标点符号或特殊字符。

当我们需要将LLM的输出从数字转换回文本时，我们需要一种将标记 ID 转换回文本标记的方法。为此，我们可以创建一个逆词汇表，将标记 ID 映射回相应的文本标记。

Let's implement a complete tokenizer class in Python with an `encode` method that splits text into tokens and carries out the string-to-integer mapping to produce token IDs via the vocabulary. In addition, we'll implement a `decode` method that carries out the reverse integer-to-string mapping to convert the token IDs back into text. The following listing shows the code for this tokenizer implementation.

Listing 2.3 Implementing a simple text tokenizer

```
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
```

Stores the vocabulary as a class attribute for access in the encode and decode methods

Creates an inverse vocabulary that maps token IDs back to the original text tokens

Processes input text into token IDs

Converts token IDs back into text

Removes spaces before the specified punctuation

Using the `SimpleTokenizerV1` Python class, we can now instantiate new tokenizer objects via an existing vocabulary, which we can then use to encode and decode text, as illustrated in figure 2.8.

Let's instantiate a new tokenizer object from the `SimpleTokenizerV1` class and tokenize a passage from Edith Wharton's short story to try it out in practice:

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

The preceding code prints the following token IDs:

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

Next, let's see whether we can turn these token IDs back into text using the decode method:

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

让我们在 Python 中实现一个完整的分词器类，其中包含一个 encode 方法，该方法将文本分割成标记并通过词汇表执行字符串到整数的映射以生成标记 ID。此外，我们还将实现一个 decode 方法，该方法执行反向的整数到字符串映射，将标记 ID 转换回文本。以下列表显示了此分词器实现的代码。

列表 2.3 实现一个简单的文本分词器

将词汇存储为类属性，以便在编码和解码方法中访问

创建一个反向词汇表，将标记 ID 映射回原始文本标记

```
类 SimpleTokenizerV1:
    def __init__(self, 词汇):
        self.str_to_int = 词汇表
     self.int_to_str = {i:s}
                                for s,i in vocab.items(){}

    def encode(self, text): # 将文本编码为特定格式
        预处理 = re.split(r'([,.?_!"()\']|--|\s)', 文本) 预处理 = [

        item.strip() for item in 预处理后的 if item.strip() ] ids =
     [self.str_to_int[s] for s in 预处理后的] return ids
```

进程输入文本进入令牌 IDs

将令牌 ID 转换为回到文本

```
    def decode(self, ids):
        输入文本：text = " 将列表中的整数转换为字符串并连接成一个字符串 ".join([int_to_str[s] for s in ids])
        翻译text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) 返回
        text
```

删除指定标点符号前的空格

使用 SimpleTokenizerV1 Python 类，我们现在可以通过现有的词汇表实例化新的分词对象，然后我们可以使用这些对象来编码和解码文本，如图 2.8 所示。

让我们从 SimpleTokenizerV1 类中实例化一个新的分词器对象，并从伊迪丝·华顿的短篇小说中分词一段文本，以在实践中尝试它：

```
    分词器 = SimpleTokenizerV1(vocab) 文本 =
"""It's the last he                粉刷过的，你知道的，
    Gisburn 描着可以原谅的骄傲说。``` ids =
tokenizer.encode(text) 打印(ids)
```

上一行代码打印以下令牌 ID：

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

接下来，让我们看看我们是否可以使用 decode 方法将这些令牌 ID 转换回文本：

```
    打印（分词器解码（ids））
```

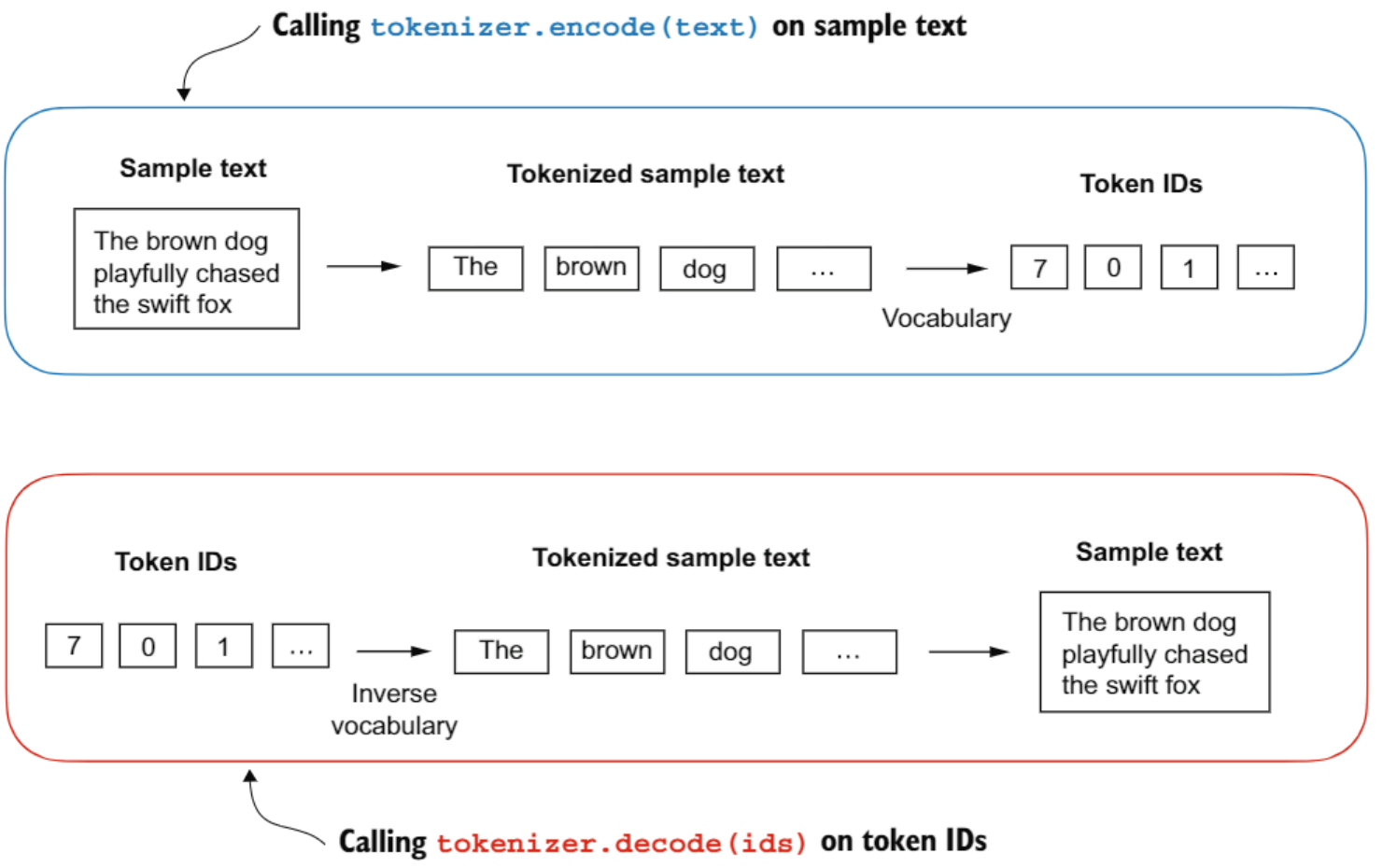

Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text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

This outputs:

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

Based on this output, we can see that the decode method successfully converted the token IDs back into the original text.

So far, so good. We implemented a tokenizer capable of tokenizing and detokenizing text based on a snippet from the training set. Let's now apply it to a new text sample not contained in the training set:

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

Executing this code will result in the following error:

```
KeyError: 'Hello'
```

The problem is that the word "Hello" was not used in the "The Verdict" short story. Hence, it is not contained in the vocabulary. This highlights the need to consider large and diverse training sets to extend the vocabulary when working on LLMs.

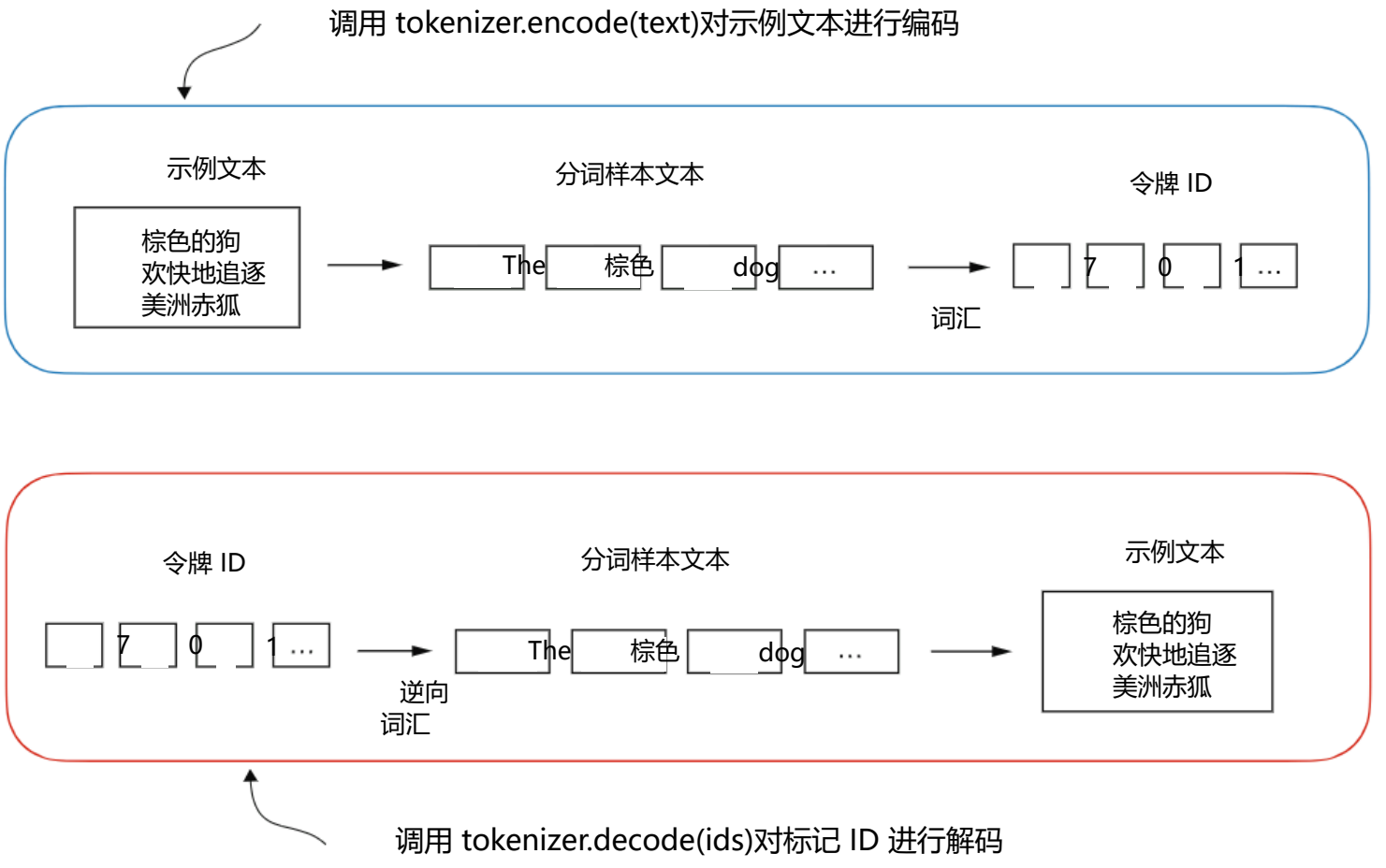

图 2.8 分词器实现共享两种常见方法：编码方法和解码方法。编码方法接收样本文本，将其分割成单个标记，并通过词汇表将标记转换为标记 ID。解码方法接收标记 ID，将它们转换回文本标记，并将文本标记连接成自然文本。

这会输出：

```
"这是他最后画的，你知道的，”吉斯伯恩夫人带着可以原谅的骄傲说。"
```

基于此输出，我们可以看到解码方法成功将令牌 ID 转换回原始文本。

到目前为止，一切顺利。我们实现了一个能够基于训练集的一个片段进行分词和去分词的标记器。现在让我们将其应用于训练集中未包含的新文本样本：

```
你好，你喜欢茶吗？
print(tokenizer.encode(text))
```

执行此代码将导致以下错误：

```
键错误: 'Hello'
```

问题在于“Hello”这个词在“The Verdict”短篇小说中没有使用。因此，它不包含在词汇表中。这突出了在处理LLMs时考虑大型和多样化的训练集以扩展词汇表的需求。

Next, we will test the tokenizer further on text that contains unknown words and discuss additional special tokens that can be used to provide further context for an LLM during training.

2.4 Adding special context tokens

We need to modify the tokenizer to handle unknown words. We also need to address 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, `SimpleTokenizerV2`, to support two new tokens, `<|unk|>` and `<|endoftext|>`, as illustrated in figure 2.9.

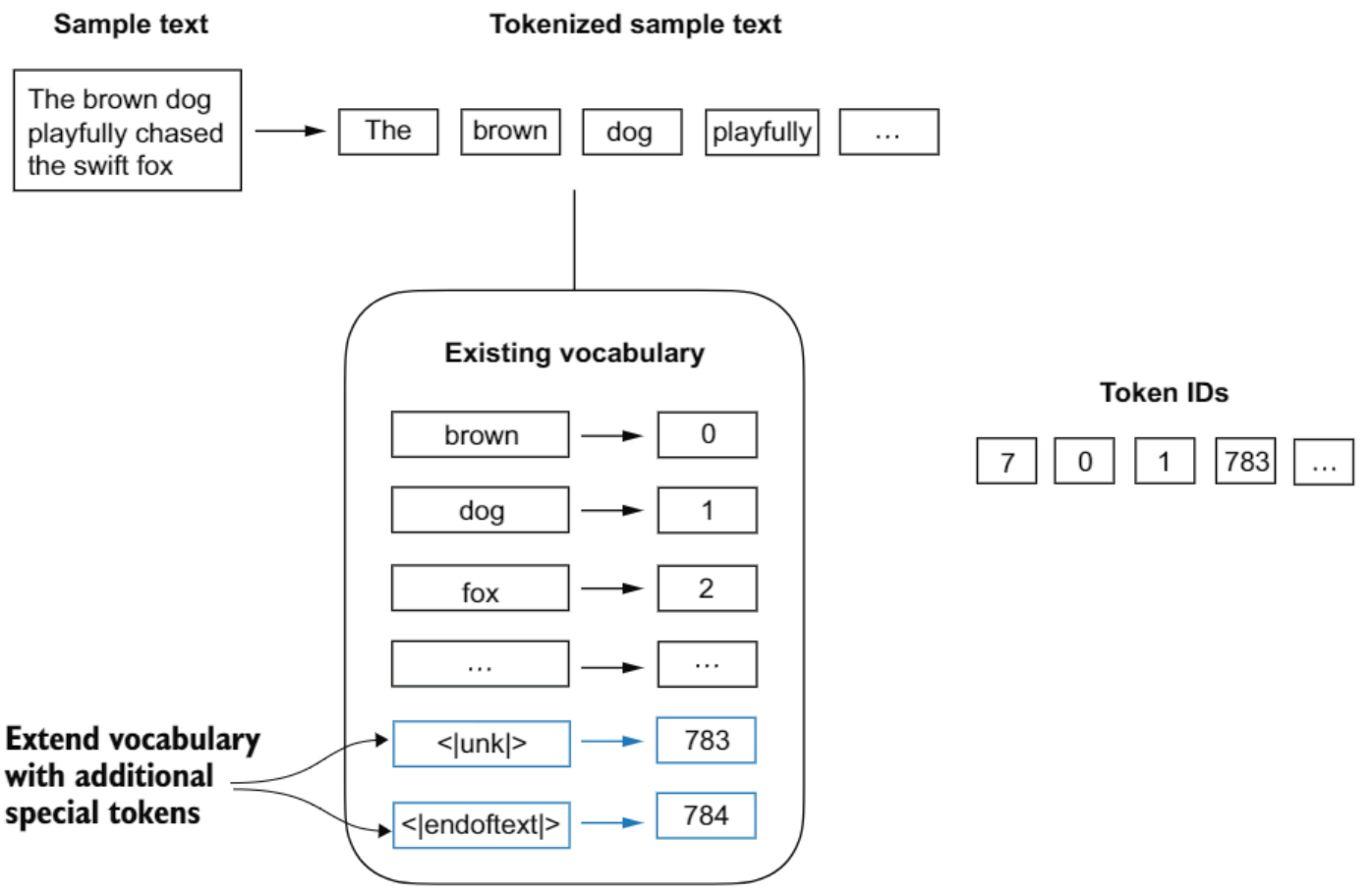

Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an `<|unk|>` token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an `<|endoftext|>` token that we can use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

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, as illustrated in figure 2.10. This helps the LLM understand that although these text sources are concatenated for training, they are, in fact, unrelated.

接下来，我们将进一步在包含未知单词的文本上测试分词器，并讨论在训练期间为LLM提供更多上下文可以使用的额外特殊标记。

2.4 添加特殊上下文标记

我们需要修改分词器以处理未知单词。我们还需要解决特殊上下文标记的使用和添加，这些标记可以增强模型对上下文或其他相关信息的理解。这些特殊标记可以包括未知单词和文档边界的标记，例如。特别是，我们将修改词汇表和分词器SimpleTokenizerV2，以支持两个新标记<|unk|>和<|end|>，如图 2.9 所示。

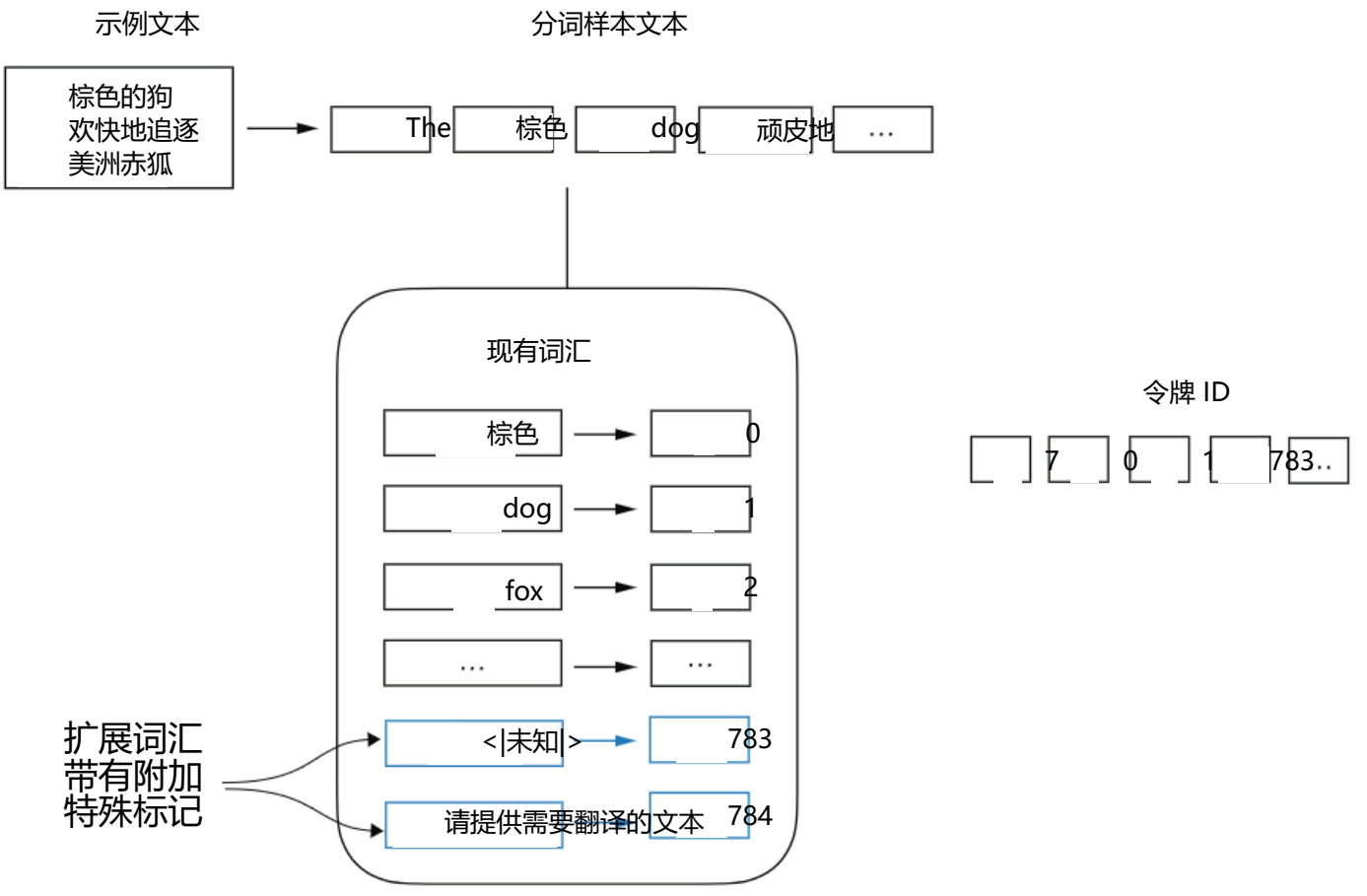

图 2.9 我们向词汇表中添加特殊标记以处理某些上下文。例如，我们添加一个<|unk|>标记来表示训练数据中不存在的未知新词，因此它们不属于现有词汇。此外，我们还添加了一个标记，可以用来分隔两个无关的文本来源。

我们可以修改分词器，在遇到不在词汇表中的单词时使用一个<|unk|>标记。此外，我们在无关文本之间添加一个标记。例如，在训练类似于 GPT 的LLMs的多个独立文档或书籍时，通常会在每个后续文档或书籍之前插入一个标记，如图 2.10 所示。这有助于LLM理解，尽管这些文本源在训练中是连接在一起的，但实际上它们是不相关的。

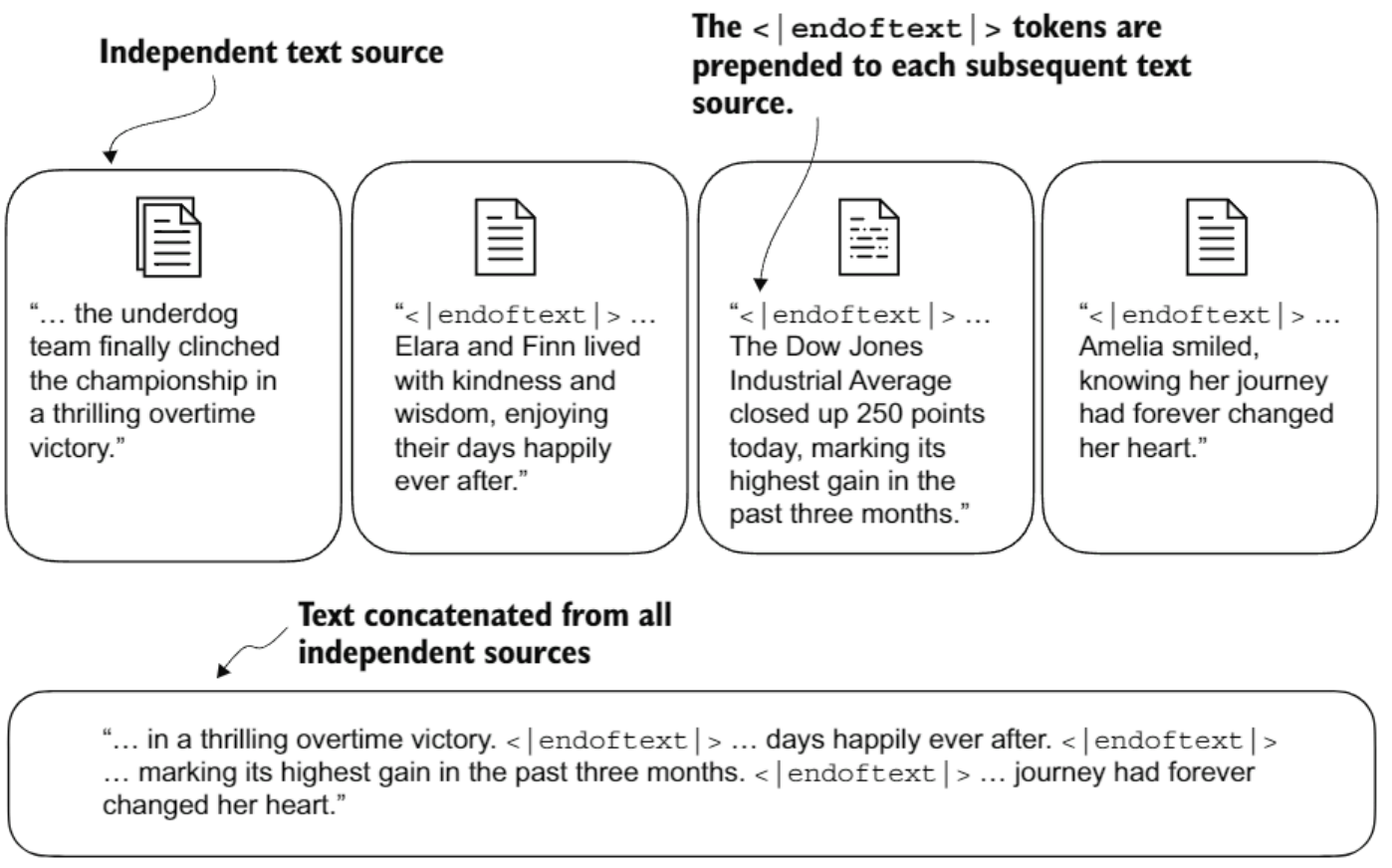

Figure 2.10 When working with multiple independent text source, we add <|endoftext|> tokens between these texts. These <|endoftext|> tokens act as markers, signaling the start or end of a particular segment, allowing for more effective processing and understanding by the LLM.

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

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

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

Based on the output of this print statement, the new vocabulary size is 1,132 (the previous vocabulary size was 1,130).

As an additional quick check, let's print the last five entries of the updated vocabulary:

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

The code prints

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

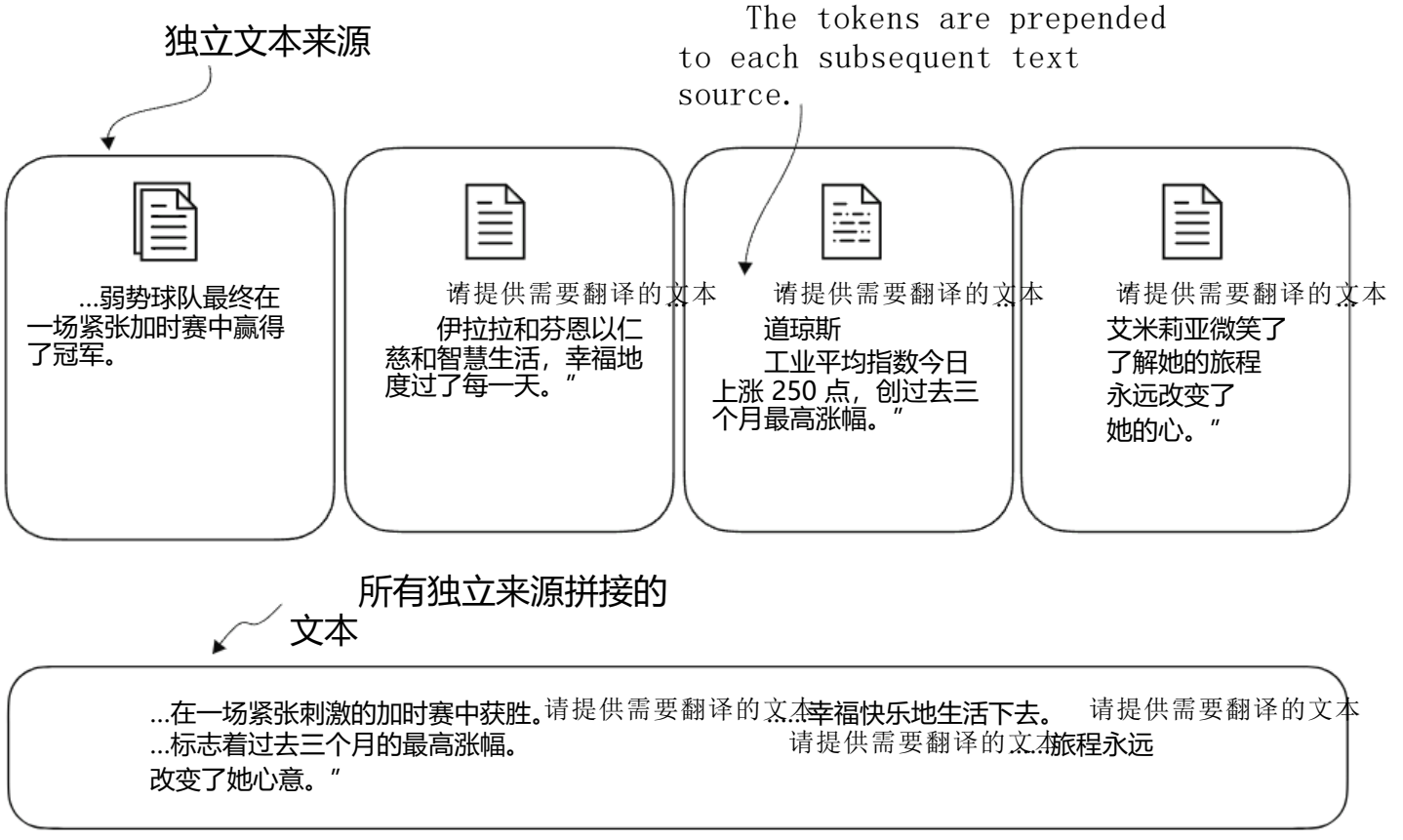

图 2.10 当处理多个独立文本源时，我们在这些文本之间添加标记。这些标记作为标记，指示特定段落的开始或结束，使处理和理解更加有效。

现在让我们修改词汇表以包括这两个特殊标记 未知内容and 请提供需要翻译的文本通过将它们添加到我们所有独特单词的列表中：

```
    所有令牌 = sorted(list(set(预处理))) 所有令
牌.extend(["", "<|unk|>"]) 词汇表 = {令牌:整数 for 整数,
令牌 in
                                                遍历(all_tokens)}

    打印(vocab.items()的长度)
```

基于此打印语句的输出，新词汇量是 1,132（之前的词汇量是 1,130）。

作为额外的快速检查，让我们打印更新词汇的最后五条条目：

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

代码打印

```
    较年轻，1127
    ('your', 1128)
    请提供需要翻译的文本
    ('（未知符号，1130）',
'（<|unk|>，1131）')
```

Based on the code output, we can confirm that the two new special tokens were indeed successfully incorporated into the vocabulary. Next, we adjust the tokenizer from code listing 2.3 accordingly as shown in the following listing.

Listing 2.4 A simple text tokenizer that handles unknown words

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', 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
```

Replaces unknown words by <|unk|> tokens

Replaces spaces before the specified punctuations

Compared to the `SimpleTokenizerV1` we implemented in listing 2.3, the new `SimpleTokenizerV2` replaces unknown words with `<|unk|>` tokens.

Let's now try this new tokenizer out in practice. For this, we will use a simple text sample that we concatenate from two independent and unrelated sentences:

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

The output is

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

Next, let's tokenize the sample text using the `SimpleTokenizerV2` on the vocab we previously created in listing 2.2:

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

基于代码输出，我们可以确认两个新的特殊标记确实成功纳入了词汇表。接下来，我们根据以下列表相应地调整分词器。

列表 2.4 一个简单的文本分词器，用于处理未知单词

```
类 SimpleTokenizerV2:
    def __init__(self, 词汇):
        self.str_to_int = 词汇表
     self.int_to_str = {                i:s for: is,si 为in  vocab.items()

    def encode(self, text): # 将文本编码为特定格式
        预处理 = re.split(r'([,.:;?_!"()\']|--|\s)', 文本) 预处理 = [

        item.strip() for item in 预处理后的 if item.strip() ] 预处理后的
     = [item if item in self.str_to_int

                                else "<|未知|>" for    项目在预处理中]

        ids = [self.str_to_int[s] for s in 预处理] return ids


    def decode(self, ids):
        输入文本：text = "将列表中的整数转换为字符串并连接成一个字符串".join([str(i) for i in ids])

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

替换
未知词汇
通过未知实体
代币

替换指定标点
符号前的空格

与我们在 2.3 列表中实现的 SimpleTokenizerV1 相比，新的 SimpleTokenizerV2 将未知单词替换为<|unk|>标记。

现在让我们在实践中学试这个新的分词器。为此，我们将使用一个简单的文本样本，该样本由两个独立且无关的句子拼接而成：

```
你好，你喜欢茶？在宫殿的阳光露台上。
```

输出结果

```
你好，你喜欢茶吗？在宫殿阳光照耀的露台上。
```

接下来，让我们使用之前在列表 2.2 中创建的词汇表，用 SimpleTokenizerV2 对示例文本进行分词。

```
分词器 = SimpleTokenizerV2(vocab) 打印
(tokenizer.encode(text))
```

This prints the following token IDs:

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

We can see that the list of token IDs contains `1130` for the `<|endoftext|>` separator token as well as two `1131` tokens, which are used for unknown words.

Let's detokenize the text for a quick sanity check:

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

The output is

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

Based on comparing this detokenized text with the original input text, we know that the training dataset, Edith Wharton's short story "The Verdict," does not contain the words "Hello" and "palace."

Depending on the LLM, some researchers also consider additional special tokens such as the following:

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

The tokenizer used for GPT models does not need any of these tokens; it only uses an `<|endoftext|>` token for simplicity. `<|endoftext|>` is analogous to the `[EOS]` token. `<|endoftext|>` is also used for padding. 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 words down into subword units, which we will discuss next.

这打印以下令牌 ID：

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

我们可以看到，标记 ID 列表中包含 1130 个分隔符标记以及两个 1131 个标记，这些标记用于未知单词。

让我们对文本进行去标记化以进行快速检查：

```
打印（tokenizer.decode(tokenizer.encode(text)）
```

输出结果

```
<|unk|>，你喜欢茶吗？在<|unk|>的阳光露台上。
```

基于将去标记化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》，不包含单词“Hello”和“palace”。

根据LLM，一些研究人员还考虑了以下额外的特殊标记：

- [BOS]（序列开始）—此标记表示文本的开始。它向LLM指示内容开始的位置。
- [EOS]（序列结束）—此标记位于文本末尾，在连接多个无关文本时特别有用，类似于 。例如，当合并两篇不同的维基百科文章或书籍时，[EOS]标记表示一个结束和下一个开始的位置。
- [PAD]（填充）—当使用大于一的批量大小训练LLMs时，批量中可能包含不同长度的文本。为确保所有文本长度相同，较短的文本将被扩展或“填充”使用[PAD]标记，直到与批量中最长文本的长度相同。

GPT 模型使用的分词器不需要这些标记；它只使用一个标记以简化。标记与[EOS]标记类似。[EOS]也用于填充。然而，正如我们将在后续章节中探讨的，在批量输入上进行训练时，我们通常使用掩码，这意味着我们不关注填充标记。因此，选择的特定填充标记变得无关紧要。

此外，用于 GPT 模型的分词器也不会为词汇表外的单词使用 `<|unk|>` 标记。相反，GPT 模型使用字节对编码分词器，将单词分解成子词单元，我们将在下一节中讨论。

2.5 Byte pair encoding

Let's look at a more sophisticated tokenization scheme based on a concept called byte pair encoding (BPE). The BPE tokenizer was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.

Since implementing BPE can be relatively complicated, we will use an existing Python open source library called *tiktoken* (https://github.com/openai/tiktoken), which implements the BPE algorithm very efficiently based on source code in Rust. Similar to other Python libraries, we can install the tiktoken library via Python's `pip` installer from the terminal:

```
pip install tiktoken
```

The code we will use is based on tiktoken 0.7.0. You can use the following code to check the version you currently have installed:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

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

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

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

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

The code prints the following token IDs:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
 8812, 2114, 286, 617, 34680, 27271, 13]
```

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

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

The code prints

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

2.5 字节对编码

让我们看看一种基于字节对编码（BPE）概念的更复杂的分词方案。BPE 分词器被用于训练 GPT-2、GPT-3 以及 ChatGPT 中使用的原始模型。

由于实现 BPE 可能相对复杂，我们将使用一个名为 tiktoken 的现有 Python 开源库（https://github.com/openai/tiktoken），该库基于 Rust 源代码非常高效地实现了 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：

```
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来检查您当前已安装的版本：

```
   from importlib.metadata import version import
tiktoken 打印("tiktoken 版本：", version("tiktoken"))
```

一旦安装，我们可以如下实例化来自 tiktoken 的 BPE 分词器：

```
分词器       = 获取 gpt2 编码器
```

此分词器的使用方式与我们之前通过 encode 方法实现的 SimpleTokenizerV2 类似：

```
   text = (
   你好，你喜欢茶吗？在某个未知地方的阳光露台上）整数 = 分词器.encode(text, 允许特
殊字符={}) 打印(整数)
```

代码打印以下令牌 ID：

```
   [15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812,
2114, 286, 617, 34680, 27271, 13]
```

我们可以使用 decode 方法将标记 ID 转换回文本，类似于我们的 SimpleTokenizerV2：

```
strings = 分词器解码整数 print(strings)
```

代码打印

```
你好，你喜欢茶吗？在某个未知地方的阳光露台上。
```

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

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

The algorithm underlying BPE breaks down words that aren't in its predefined 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, as illustrated in figure 2.11.

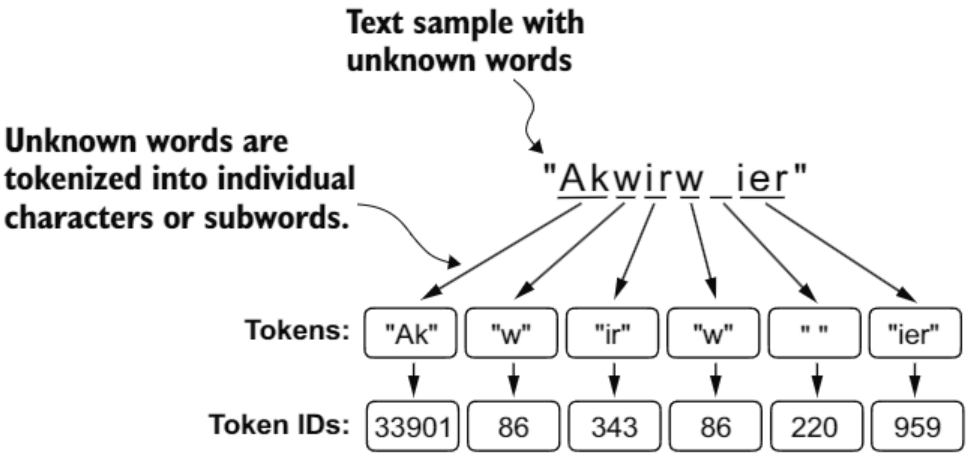

Figure 2.11 BPE tokenizers break down unknown words into subwords and individual characters. This way, a BPE tokenizer can parse any word and doesn't need to replace unknown words with special tokens, such as `<|unk|>`.

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.

Exercise 2.1 Byte pair encoding of unknown words

Try the BPE tokenizer from the tiktoken library on the unknown words "Akwirw ier" and print the individual token IDs. Then, call the `decode` function on each of the resulting integers in this list to reproduce the mapping shown in figure 2.11. Lastly, call the decode method on the token IDs to check whether it can reconstruct the original input, "Akwirw ier."

A detailed discussion and implementation of BPE is out of the scope of this book, but in short, it builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary ("a," "b," etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, "d" and "e" may be merged into the subword "de," which is common in many English

我们可以基于标记 ID 和解码文本做出两个值得注意的观察。首先，token 被分配了一个相对较大的标记 ID，即 50256。实际上，用于训练 GPT-2、GPT-3 以及 ChatGPT 中使用的原始模型的 BPE 分词器，其总词汇量为 50257，其中 token 被分配了最大的标记 ID。

其次，BPE 分词器能够正确地编码和解码未知词，例如 someunknownPlace。BPE 分词器可以处理任何未知词。它是如何在不使用<|unk|>标记的情况下实现这一点的呢？

BPE 算法背后的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或甚至单个字符，使其能够处理词汇表外的单词。因此，多亏了 BPE 算法，如果分词器在分词过程中遇到不熟悉的单词，它可以将其表示为子词标记或字符的序列，如图 2.11 所示。

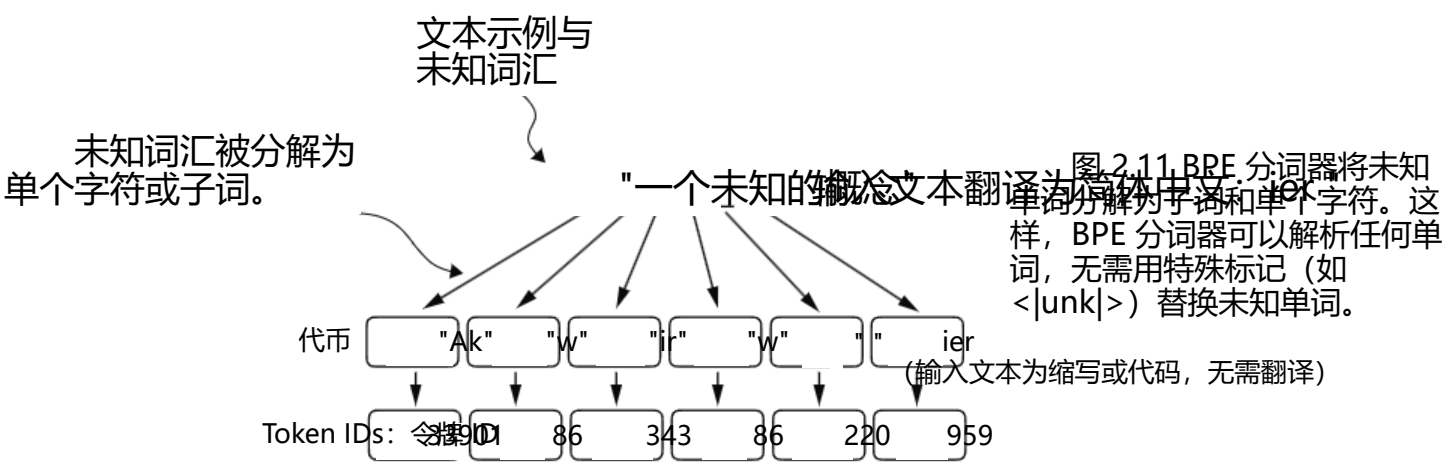

图 2.11 BPE 分词器将未知单词分解为子词和单个字符。这样，BPE 分词器可以解析任何单词，无需用特殊标记（如 <|unk|>）替换未知单词。

将未知单词分解成单个字符的能力确保了分词器以及与之训练的LLM可以处理任何文本，即使其中包含训练数据中未出现的单词。

练习 2.1 未知单词的字节对编码

尝试在 tiktoken 库的 BPE 分词器上对未知单词“Akwirw ier”进行分词，并打印出单个标记 ID。然后，对列表中的每个结果整数调用 decode 函数以重现图 2.11 所示的映射。最后，对标记 ID 调用 decode 方法以检查它是否可以重建原始输入，“Akwirw ier。”

BPE 的详细讨论和实现超出了本书的范围，但简而言之，它通过迭代地将频繁出现的字符合并为子词，将频繁出现的子词合并为单词来构建其词汇表。例如，BPE 首先将所有单个字符添加到其词汇表中（“a”、“b”等）。在下一阶段，它将经常一起出现的字符组合合并为子词。例如，“d”和“e”可能合并为子词“de”，这在许多英语单词中都很常见。

words like "define," "depend," "made," and "hidden." The merges are determined by a frequency cutoff.

2.6 Data sampling with a sliding window

The next step in creating the embeddings for the LLM is to generate the input–target pairs required for training an LLM. What do these input–target pairs look like? As we already learned, LLMs are pretrained by predicting the next word in a text, as depicted in figure 2.12.

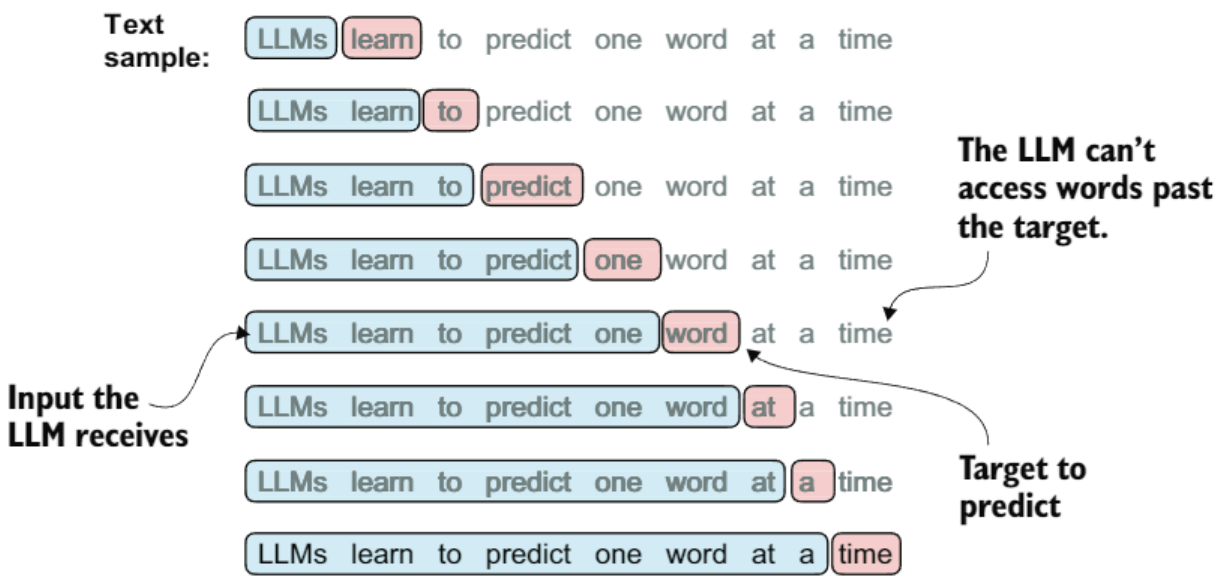

Figure 2.12 Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM's prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this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

Let's implement a data loader that fetches the input–target pairs in figure 2.12 from the training dataset using a sliding window approach. To get started, we will tokenize the whole "The Verdict" short story using the BPE tokenizer:

```
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))
```

Executing this code will return `5145`, the total number of tokens in the training set, after applying the BPE tokenizer.

Next, we remove the first 50 tokens from the dataset for demonstration purposes, as it results in a slightly more interesting text passage in the next steps:

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

这是许多英语单词中常见的，如“定义”、“依赖”、“制造”和“隐藏”。合并由频率阈值决定。

2.6 数据采样滑动窗口

下一步在创建LLM的嵌入时，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？正如我们之前所学，LLMs通过预测文本中的下一个单词进行预训练，如图 2.12 所示。

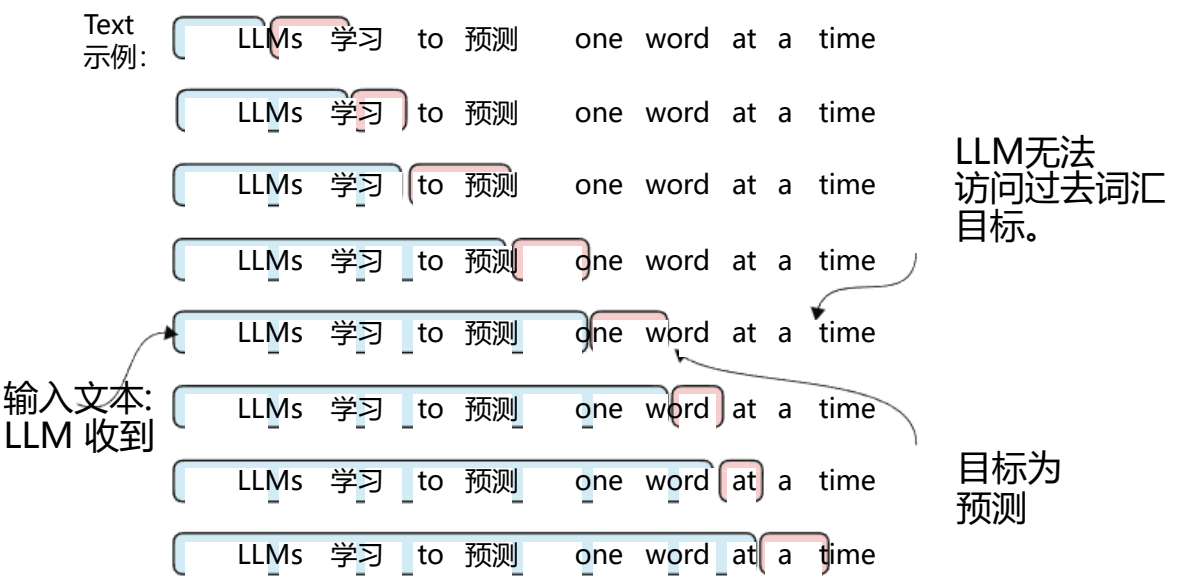

图 2.12 给定一个文本样本，提取输入块作为子样本，作为LLM和LLM在训练期间的预测任务输入，预测输入块之后的下一个单词。在训练过程中，我们屏蔽掉所有目标之后的单词。注意，图中的文本在LLM处理之前必须进行分词；然而，为了清晰起见，此图省略了分词步骤。

让我们实现一个数据加载器，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。要开始，我们将使用 BPE 标记器对整个“The Verdict”短故事进行分词：

```
    with open("the-verdict.txt", "r", encoding="utf-8") as f:
        raw_text = f.read() # 读取文件内容

    enc_text = tokenizer.encode(原始文本) 打
印(len(enc_text))
```

执行此代码将返回 5145，这是在应用 BPE 分词器后训练集中标记的总数。

接下来，为了演示目的，我们从数据集中移除前 50 个标记，这样在接下来的步骤中可以得到一个稍微更有趣的文本段落：

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

One of the easiest and most intuitive ways to create the input–target pairs for the next-word prediction task is to create two variables, `x` and `y`, where `x` contains the input tokens and `y` contains the targets, which are the inputs shifted by 1:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

The context size determines how many tokens are included in the input.

Running the previous code prints the following output:

```
x: [290, 4920, 2241, 287]
y:      [4920, 2241, 287, 257]
```

By processing the inputs along with the targets, which are the inputs shifted by one position, we can create the next-word prediction tasks (see figure 2.12), as follows:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

The code prints

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

Everything left of the arrow (`---->`) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict. Let's repeat the previous code but convert the token IDs into text:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

The following outputs show how the input and outputs look in text format:

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

We've now created the input–target pairs that we can use for LLM training.

There's only one more task before we can turn the tokens into embeddings: implementing an efficient data loader that iterates over the input dataset and returns the

创建输入-目标对以进行下一词预测任务的最简单和最直观的方法之一是创建两个变量 x 和 y，其中 x 包含输入标记，y 包含目标，即输入向右移动 1 个位置：

```
    context_size = 4
    x = enc_sample[:context_size] y =
enc_sample[1:context_size+1] 打印(f"x:
{x}") 打印(f"y: {y}")
```

上下文大小决定了输入中包含多少个标记。

运行上一段代码将输出以下内容：

```
    x: [290, 4920, 2241, 287]
y: [4920, 2241,]                [287, 257]
```

通过处理输入和目标，其中目标是输入向右移动一位，我们可以创建下一词预测任务（见图 2.12），如下所示：

```
    for i in 范围(1, context_size+1):
         context = enc_sample[:i]
     desired = enc_sample[i] 打印
     (context, "---->",                期望的（desired）
```

代码打印

```
    [290] ----> 4920
    [290, 4920] ----> 2241 [290,
[290] ----> 4920> 287 [290, 4920,
4920, 2241]
2241, 287]
                                    输入文本：
                                ----> 257
```

一切在箭头（---->）左侧的内容指的是LLM将接收到的输入，箭头右侧的标记 ID 代表LLM应该预测的目标标记-ID。让我们重复之前的代码，但将标记 ID 转换为文本：

```
    for i in 范围(1, context_size+1):
         context = enc_sample[:i] desired =
     enc_sample[i] 打印
     (tokenizer.decode(context),)                "---->", tokenizer.decode([desired])
```

以下输出显示了输入和输出在文本格式下的样子：

```
    并且 ----> 建立了并且建立了 ----> 他
自己并且建立了自己 ----> 在并且建立了自
己 在 ---->
                                    a
```

我们现在已经创建了可用于LLM训练的输入-目标对。

我们只需完成一个任务，就能将标记转换为嵌入：实现一个高效的数据加载器，该加载器遍历输入数据集并返回

inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure 2.13. While the figure shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

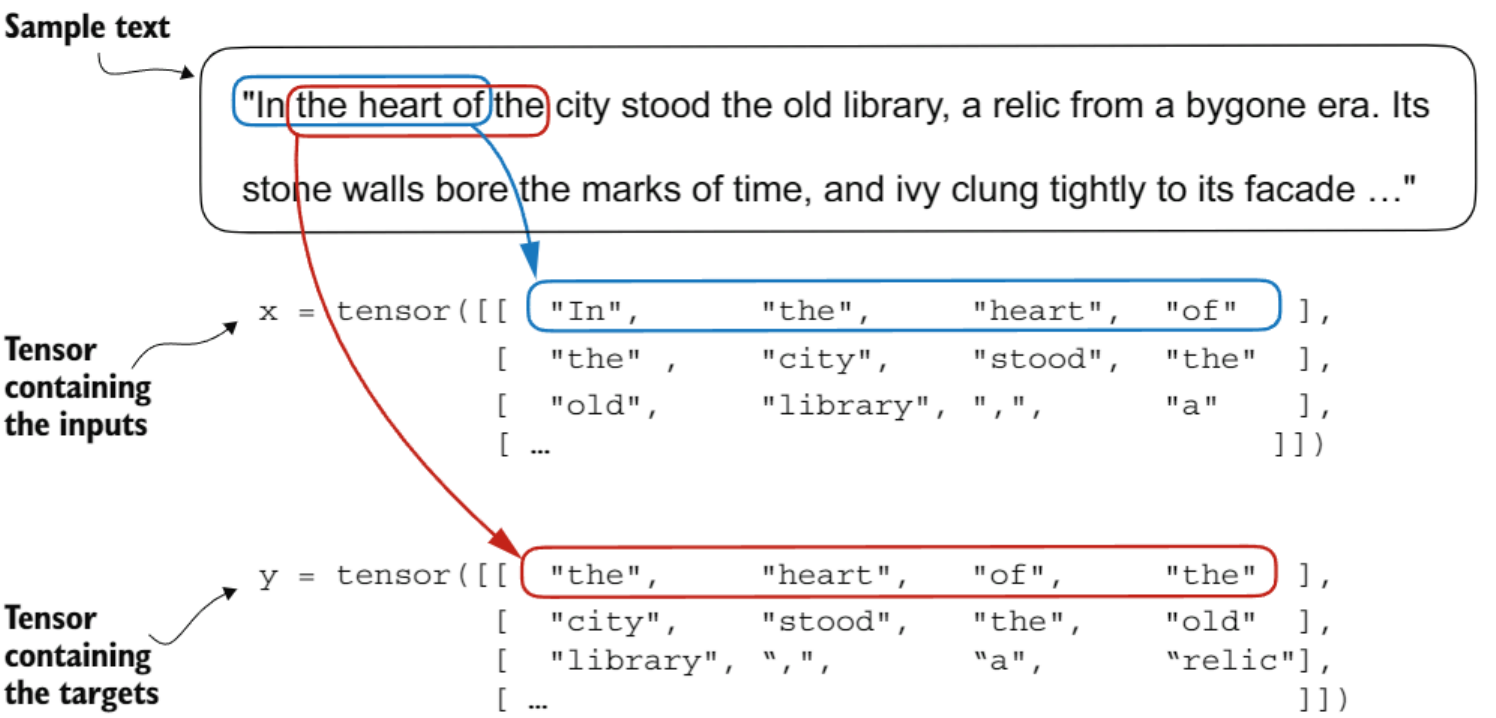

Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, `x`, where each row represents one input context. A second tensor, `y`, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.

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

The code for the dataset class is shown in the following listing.

Listing 2.5 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.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt)
```

Tokenizes the entire text

输入和目标作为 PyTorch 张量，可以将其视为多维数组。特别是，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预测的目标张量，如图 2.13 所示。虽然图示为了说明目的以字符串格式显示标记，但代码实现将直接操作标记 ID，因为 BPE 标记器的 encode 方法将标记化和转换为标记 ID 作为一个步骤执行。

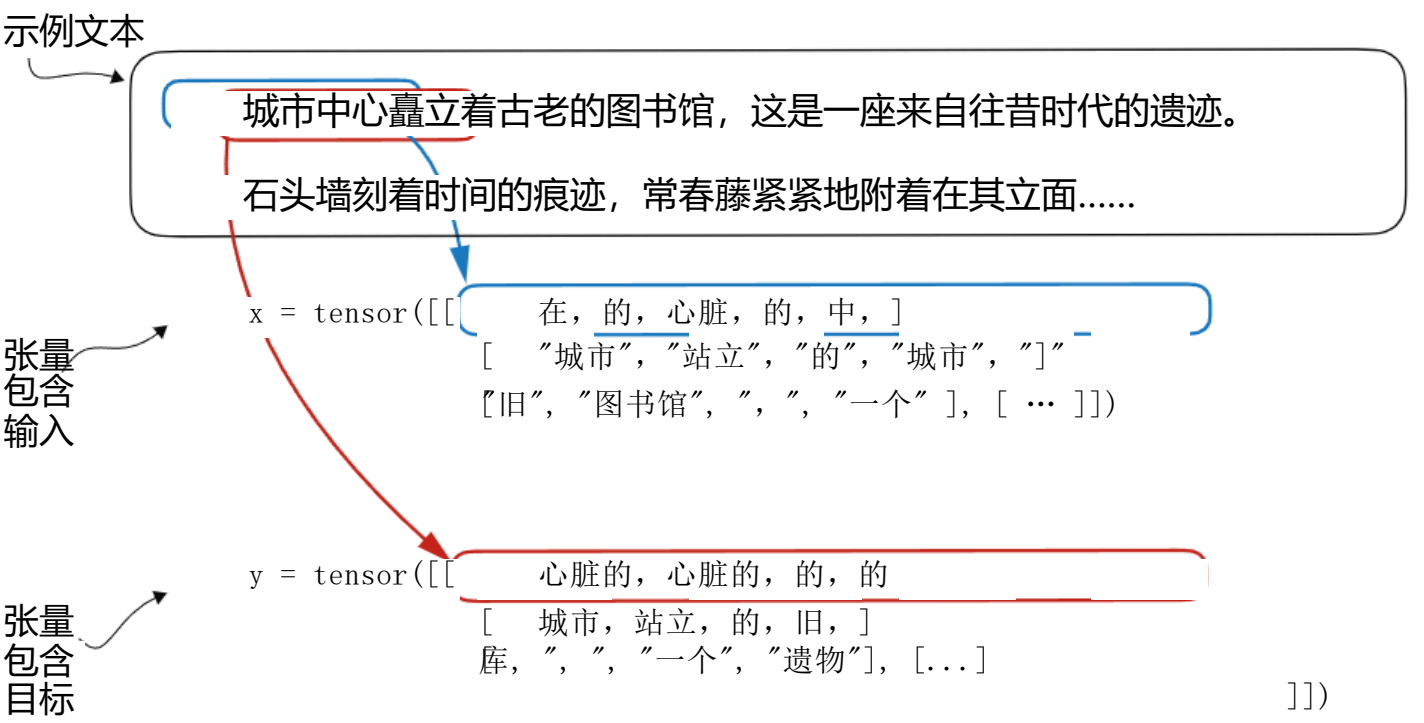

图 2.13 为了实现高效的数据加载器，我们将输入收集到一个张量 x 中，其中每一行代表一个输入上下文。第二个张量 y 包含相应的预测目标（下一个单词），这些目标是通过将输入向右移动一个位置创建的。

注意：为了高效的数据加载器实现，我们将使用 PyTorch 的内置数据加载器和数据集类。有关安装 PyTorch 的更多信息和建议，请参阅附录 A 中的 A.2.1.3 节。

数据集类的代码如下所示。

列表 2.5 批量输入和目标的数据集

```
import torch
from  torch.utils.data 导入 Dataset 和 Dataset数据加载器

class GPT 数据集 V1(Dataset)
    def __init__(self, txt, 分词器, 最大长度, 步长):
        self.input_ids = []
        self.target_ids    = []

        token_ids = tokenizer.encode(txt)
```

分词整个文本

```
            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]
```

Uses a sliding window to chunk the book into overlapping sequences of max_length

Returns a single row from the dataset

Returns the total number of rows in the dataset

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. I recommend reading on to see what the data returned from this dataset looks like when we combine the dataset with a PyTorch `DataLoader`—this will bring additional intuition and clarity.

NOTE If you are new to the structure of PyTorch `Dataset` classes, such as shown in listing 2.5, refer to section A.6 in appendix A, which explains the general structure and usage of PyTorch `Dataset` and `DataLoader` classes.

The following code uses the `GPTDatasetV1` to load the inputs in batches via a PyTorch `DataLoader`.

Listing 2.6 A data loader to generate batches with input-with pairs

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    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,
        num_workers=num_workers
    )

    return dataloader
```

Initializes the tokenizer

Creates dataset

drop_last=True drops the last batch if it is shorter than the specified batch_size to prevent loss spikes during training.

The number of CPU processes to use for preprocessing

```
        for i 在 range(0, len(token_ids) - 最大长度, 步长):
            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): # 获取长度
        返回 self.input_ids 的长度


    def __getitem__(self, 索引):
        返回 self.input_ids[idx], self.target_ids[idx]
```

返回数据集的单行

返回数据集中的总行数

使用滑动窗口进行分块书籍进入重叠的最大长度序列

GPTDatasetV1 类基于 PyTorch Dataset 类，定义了如何从数据集中获取单个行，其中每行包含一个由 max_length 分配的 token ID 数组，这些 ID 被分配给输入 chunk 张量。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 结合时返回的数据是什么样的——这将带来额外的直观性和清晰度。

注意：如果您对 PyTorch Dataset 类的结构不熟悉，例如列表 2.5 所示，请参阅附录 A 中的 A.6 节，该节解释了 PyTorch Dataset 和 DataLoader 类的通用结构和用法。

以下代码使用 GPTDatasetV1 通过 PyTorch 分批加载数据数据加载器 .

列表 2.6 数据加载器，用于生成输入-输出对批次

```
def 创建数据加载器_v1(txt, batch_size=4, max_length=256
                      stride=128, shuffle=True, drop_last=True,
                  num_workers=0): 步长=128, 乱序=True, 丢弃最后一个
                  =True, 工作进程数=0)
    分词器 = tiktoken.get_encoding("gpt2") 数据集 = GPTDatasetV1(txt,
分词器, 最大长度, 步长) 数据加载器 = DataLoader(


    数据集，批量大小为
batch_size，打乱顺序为
shuffle，丢弃最后一个为
drop_last，工作进程数为
num_workers）



    返回 数据加载器
```

初始化分词器

创建数据集

drop_last=True 在训练过程中，如果最后一个批次小于指定的 batch_size，则将其丢弃，以防止损失值波

使用的 CPU 进程数用于预处理的

Let's test the `dataloader` with a batch size of 1 for an LLM with a context size of 4 to develop an intuition of how the `GPTDatasetV1` class from listing 2.5 and the `create_dataloader_v1` function from listing 2.6 work together:

```
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)
first_batch = next(data_iter)
print(first_batch)
```

Converts dataloader into a Python iterator to fetch the next entry via Python's built-in next() function

Executing the preceding code prints the following:

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

The `first_batch` variable contains two tensors: the first tensor stores the input token IDs, and the second tensor stores the target token IDs. Since the `max_length` is set to 4, each of the two tensors contains four token IDs. Note that an input size of 4 is quite small and only chosen for simplicity. It is common to train LLMs with input sizes of at least 256.

To understand the meaning of `stride=1`, let's fetch another batch from this dataset:

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

The second batch has the following contents:

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

If we compare the first and second batches, we can see that the second batch's token IDs are shifted by one position (for example, the second ID in the first batch's input is 367, which is the first ID of the second batch's input). The `stride` setting dictates the number of positions the inputs shift across batches, emulating a sliding window approach, as demonstrated in figure 2.14.

Exercise 2.2 Data loaders with different strides and context sizes

To develop more intuition for how the data loader works, try to run it with different settings such as `max_length=2` and `stride=2`, and `max_length=8` and `stride=2`.

Batch sizes of 1, such as we have sampled from the data loader so far, are useful for illustration purposes. If you have previous experience with deep learning, you may know that small batch sizes require less memory during training but lead to more

让我们使用批大小为 1 对LLM进行测试，上下文大小为 4，以开发对 2.5 列表中的 GPTDatasetV1 类和 create_的直观理解。

数据加载器_v1 函数从 2.6 列表协同工作：

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read() # 读取文件内容

dataloader = create_dataloader_v1(
raw_text, 批处理大小=1, 最大长度=4, 步长=1, 打乱=False) 数据迭代器 =
iter(dataloader) 第一批数据 = next(data_iter) 打印(第一批数据)
```

将 dataloader 转换为 Python 迭代器，通过 Python 内置的 next()函数获取下一个条

执行前面的代码将打印以下内容：

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

第一批变量包含两个张量：第一个张量存储输入标记 ID，第二个张量存储目标标记 ID。由于最大长度设置为 4，两个张量都包含四个标记 ID。请注意，输入大小为 4 相当小，仅为了简单起见而选择。通常，使用至少 256 的输入大小来训练LLMs。

要理解 stride=1 的含义，让我们从这个数据集中获取另一个批次：

```
second_batch = 下一个批次 =
next(data_iter) 打印(second_batch)
```

第二批次包含以下内容：

```
[tensor([[ 367, 2885, 1464,  1807]]), 张量([[2885,  1464, 1807,  3619]])]
```

如果我们比较第一组和第二组，我们可以看到第二组的标记 ID 向右移动了一个位置（例如，第一组输入的第二 ID 是 367，这是第二组输入的第一个 ID）。步长设置决定了输入在组间移动的位置数，模拟滑动窗口方法，如图 2.14 所示。

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，尝试用不同的设置如 max_length=2 和 stride=2，以及 max_length=8 和 stride=2。

批大小为 1，例如我们迄今为止从数据加载器中抽取的，对于说明目的很有用。如果你有深度学习的先前经验，你可能知道小批大小在训练期间需要的内存较少，但会导致更多的

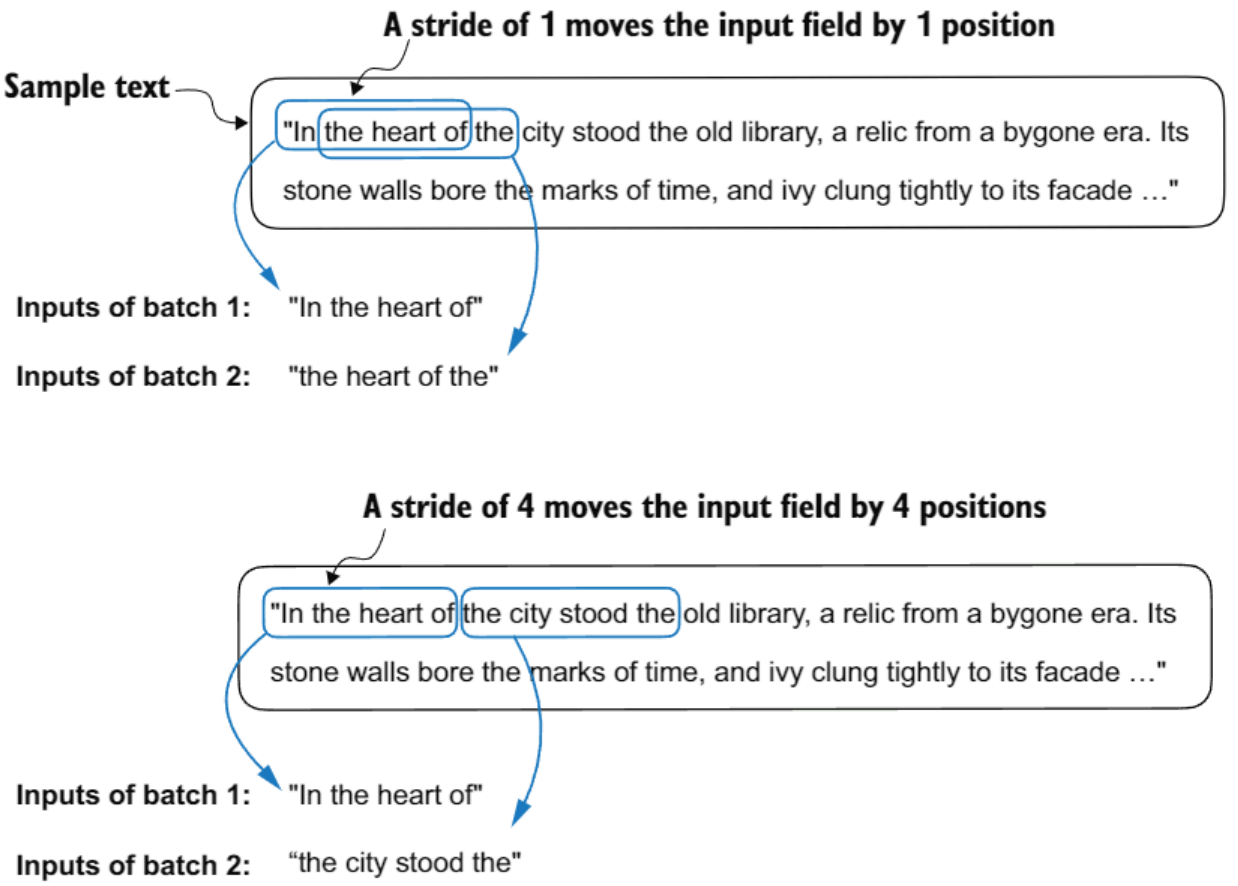

Figure 2.14 When creating multiple batches from the input dataset, we slide an input window across the text. If the stride is set to 1, we shift the input window by one position when creating the next batch. If we set the stride equal to the input window size, we can prevent overlaps between the batches.

noisy model updates. Just like in regular deep learning, the batch size is a tradeoff and a hyperparameter to experiment with when training LLMs.

Let’s look briefly at how we can use the data loader to sample with a batch size greater than 1:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

This prints

```
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],
```

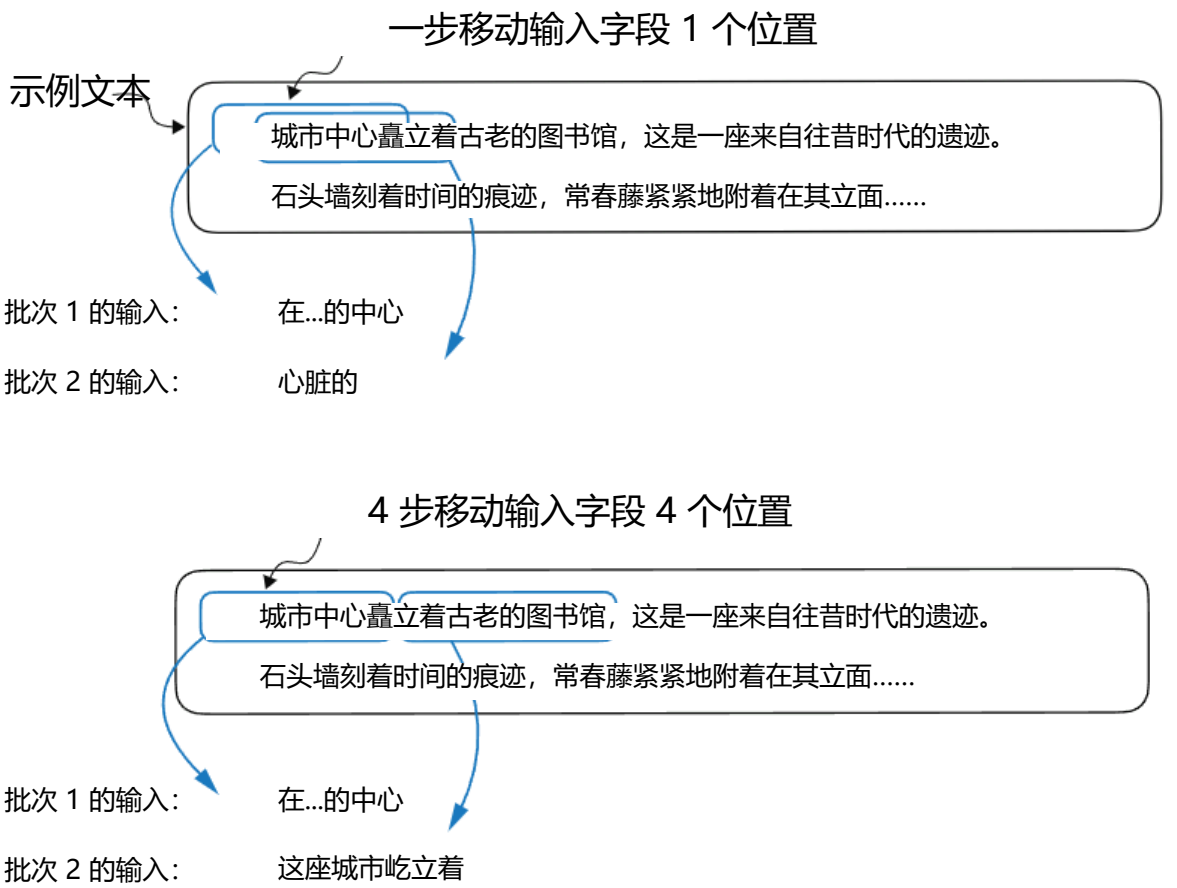

图 2.14 当从输入数据集创建多个批次时，我们在文本上滑动一个输入窗口。如果步长设置为 1，则在创建下一个批次时将输入窗口移动一个位置。如果我们设置步长等于输入窗口大小，则可以防止批次之间的重叠。

您可能知道，小批量大小在训练期间需要的内存较少，但会导致模型更新更嘈杂。就像在常规深度学习中一样，批量大小是一个权衡，并且在训练时是一个需要实验的超参数。

让我们简要看看如何使用数据加载器以大于 1 的批量大小进行采样：

```
    数据加载器 = create_dataloader_v1(
    raw_text, 批处理大小=8, 最大长度=4, 步长=4, 混洗
=False )



    data_iter = iter(dataloader) 输入，目
标 = next(data_iter) 打印("输入:\n", 输
入) 打印("\n 目标:\n", 目标)
```

这会打印

```
    输入：
     张量([[        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]])
```

Note that we increase the stride to 4 to utilize the data set fully (we don't skip a single word). This avoids any overlap between the batches since more overlap could lead to increased overfitting.

2.7 Creating token embeddings

The last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors, as shown in figure 2.15. As a preliminary step, we must initialize

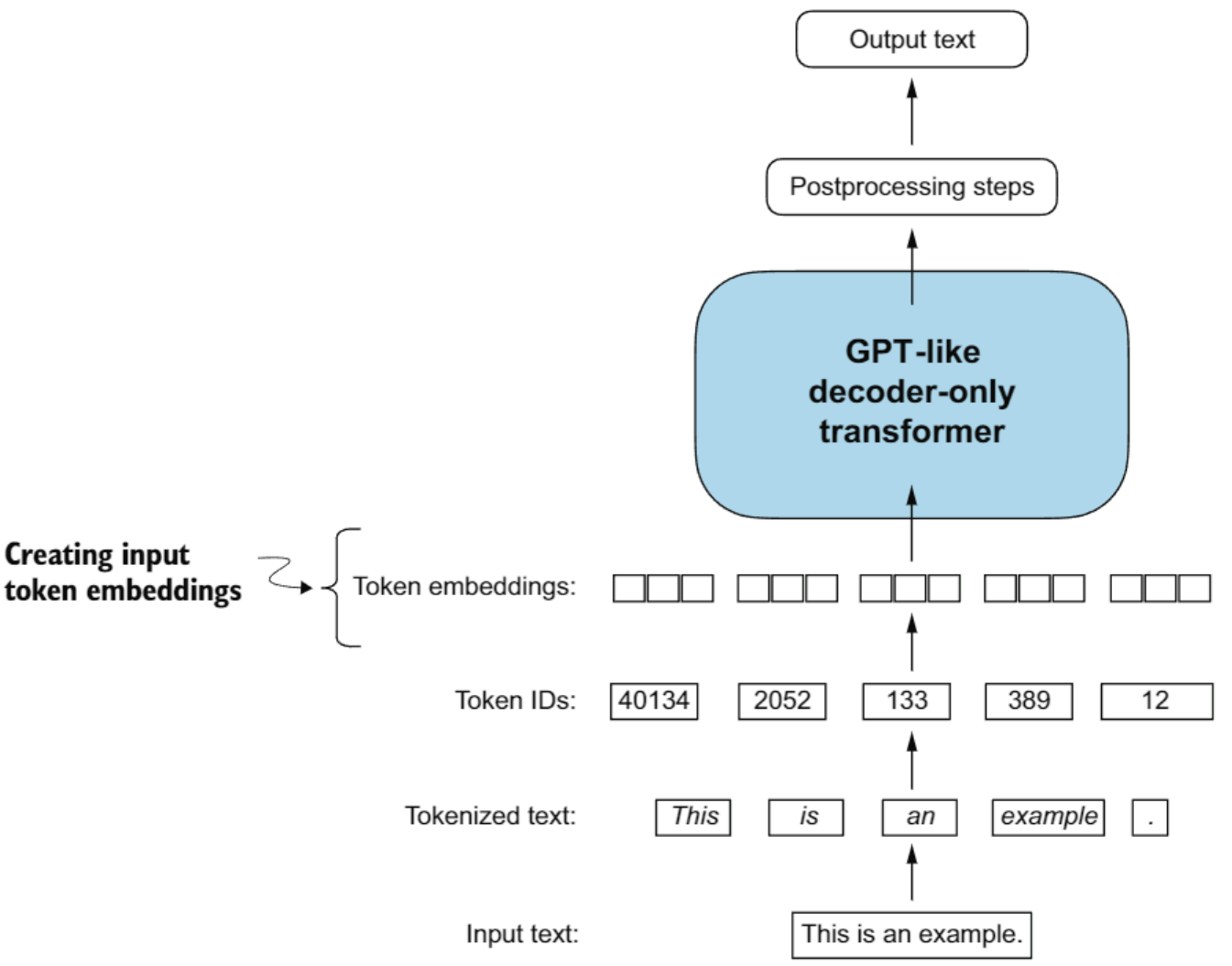

Figure 2.15 Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.

```
          [ 1049,    5975,     284,     502],
          [  284,    3285,     326,      11]])

目标：
 张量([[      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]])
```

注意我们将步长增加到 4 以充分利用数据集（我们不跳过任何单词）。这避免了批次之间的任何重叠，因为更多的重叠可能导致过拟合增加。

2.7 创建令牌嵌入

准备输入文本以进行LLM训练的最后一步是将标记 ID 转换为嵌入向量，如图 2.15 所示。作为一个初步步骤，我们必须初始化

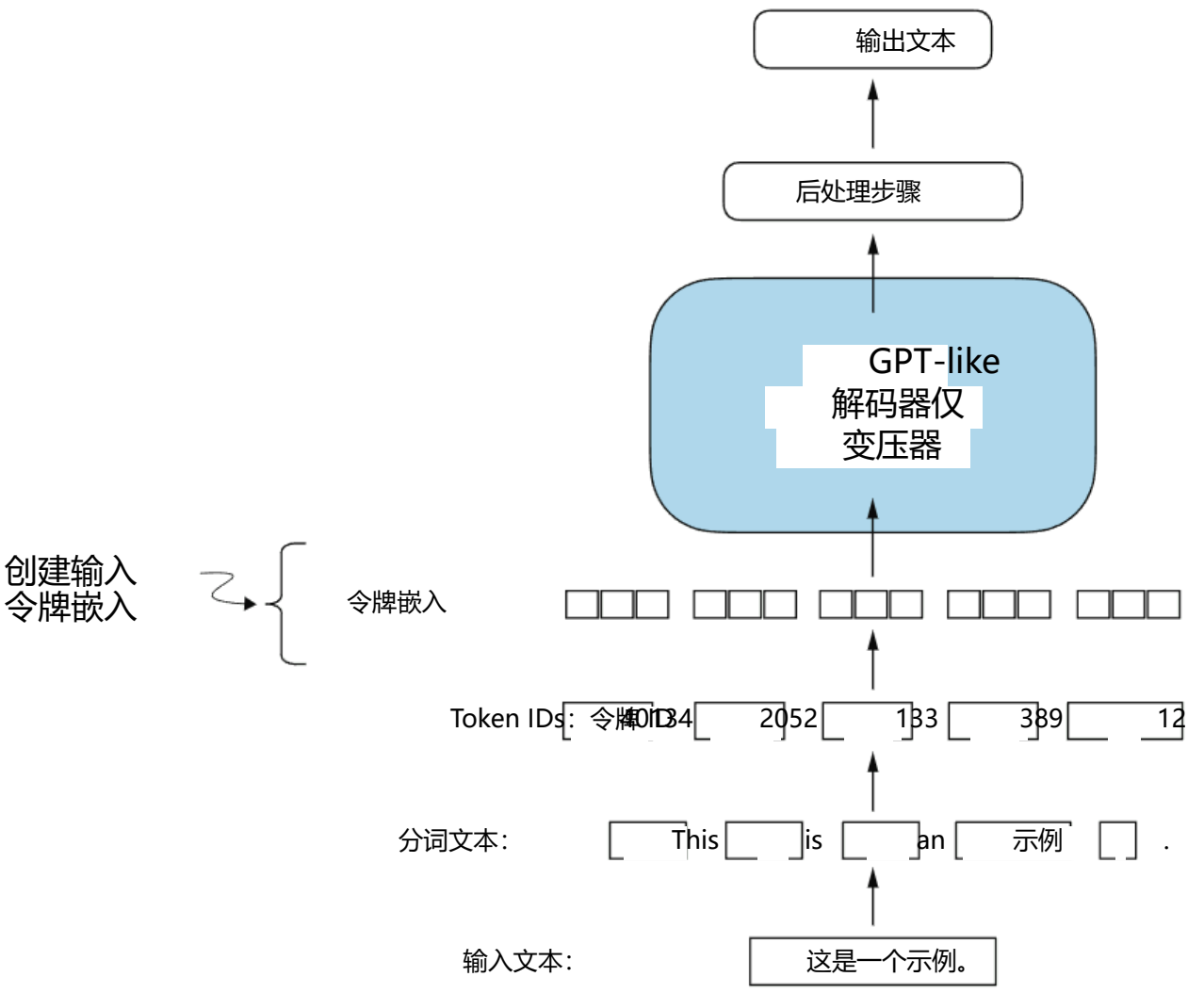

图 2.15 准备工作包括对文本进行分词、将文本分词转换为分词 ID，以及将分词 ID 转换为嵌入向量。在此，我们考虑之前创建的分词 ID 来创建分词嵌入向量。

these embedding weights with random values. This initialization serves as the starting point for the LLM's learning process. In chapter 5, we will optimize the embedding weights as part of the LLM training.

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

> **NOTE** If you are unfamiliar with how neural networks are trained with backpropagation, please read section B.4 in appendix A.

Let's see how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and 1:

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

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

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

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

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

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

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

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

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

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

这些具有随机值的嵌入权重。这种初始化作为LLM学习过程的起点。在第 5 章中，我们将优化嵌入权重作为LLM训练的一部分。

连续向量表示或嵌入是必要的，因为类似于 GPT 的LLMs是使用反向传播算法训练的深度神经网络。

注意：如果您不熟悉如何使用反向传播训练神经网络，请阅读附录 A 中的 B.4 节。

让我们通过一个实际例子来看看如何将令牌 ID 转换为嵌入向量。假设我们有以下四个输入令牌，其 ID 分别为 2、3、5 和 1：

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

为了简化，假设我们只有一个小型的词汇量，仅包含 6 个单词（而不是 BPE 分词器词汇表中的 50,257 个单词），并且我们想要创建大小为 3 的嵌入（在 GPT-3 中，嵌入大小为 12,288 维）：

```
vocab_size = 6
输出维度      = 3
```

使用 vocab_size 和 output_dim，我们可以在 PyTorch 中实例化一个嵌入层，为了可重复性，设置随机种子为 123：

```
    torch.manual_seed(123) 嵌入层 = torch.nn.Embedding(词汇大小, 输出维
度) 打印(嵌入层权重)
```

打印语句打印嵌入层的底层权重矩阵：

```
    包含参数：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小的随机值。这些值在LLM训练过程中作为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。每一行对应词汇表中的六个可能的标记之一，每一列对应三个嵌入维度之一。

现在，让我们将其应用于一个令牌 ID 以获取嵌入向量：

```
print(嵌入层(torch.tensor([3])))
```

The returned embedding vector is

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

If we compare the embedding vector for token ID 3 to the previous embedding matrix, we see that it is identical to the fourth row (Python starts with a zero index, so it's the row corresponding to index 3). In other words, the embedding layer is essentially a lookup operation that retrieves rows from the embedding layer's weight matrix via a token ID.

> **NOTE** For those who are familiar with one-hot encoding, the embedding layer approach described here is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at https://mng.bz/ZEB5. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation.

We've seen how to convert a single token ID into a three-dimensional embedding vector. Let's now apply that to all four input IDs (`torch.tensor([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

The print output reveals that this results in a 4×3 matrix:

```
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>)
```

Each row in this output matrix is obtained via a lookup operation from the embedding weight matrix, as illustrated in figure 2.16.

Having now created embedding vectors from token IDs, next we'll add a small modification to these embedding vectors to encode positional information about a token within a text.

2.8 Encoding word positions

In principle, token embeddings are a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism (see chapter 3) doesn't have a notion of position or order for the tokens within a sequence. 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, as shown in figure 2.17.

返回的嵌入向量是

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

如果我们比较 token ID 3 的嵌入向量与之前的嵌入矩阵，我们会看到它与第四行相同（Python 从零开始索引，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，通过 token ID 检索嵌入层权重矩阵中的行。

注意：对于熟悉独热编码的人来说，这里描述的嵌入层方法本质上只是实现独热编码后跟全连接层矩阵乘法的一种更有效的方式，这在 GitHub 上的补充代码中有所说明，链接为 https://mng.bz/ZEB5。因为嵌入层只是独热编码和矩阵乘法方法的一种更有效的实现，所以它可以被视为一个可以通过反向传播进行优化的神经网络层。

我们已经看到如何将单个令牌 ID 转换为三维嵌入向量。现在让我们将这个方法应用到所有四个输入 ID（torch.tensor([2, 3, 5, 1]))上：

```
打印(嵌入层(输入 ID))
```

打印输出显示这导致一个 4×3 矩阵：

```
张量([[   [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>)
```

每行输出矩阵都是通过从嵌入权重矩阵中进行查找操作获得的，如图 2.16 所示。

现在已从标记 ID 创建了嵌入向量，接下来我们将对这些嵌入向量进行微小修改，以编码文本中标记的位置信息。

2.8 编码单词位置

原则上，标记嵌入是LLM的合适输入。然而，LLMs的一个小缺点是它们的自注意力机制（见第 3 章）没有对序列中标记的位置或顺序的概念。先前引入的嵌入层的工作方式是，相同的标记 ID 总是映射到相同的向量表示，无论该标记 ID 在输入序列中的位置如何，如图 2.17 所示。

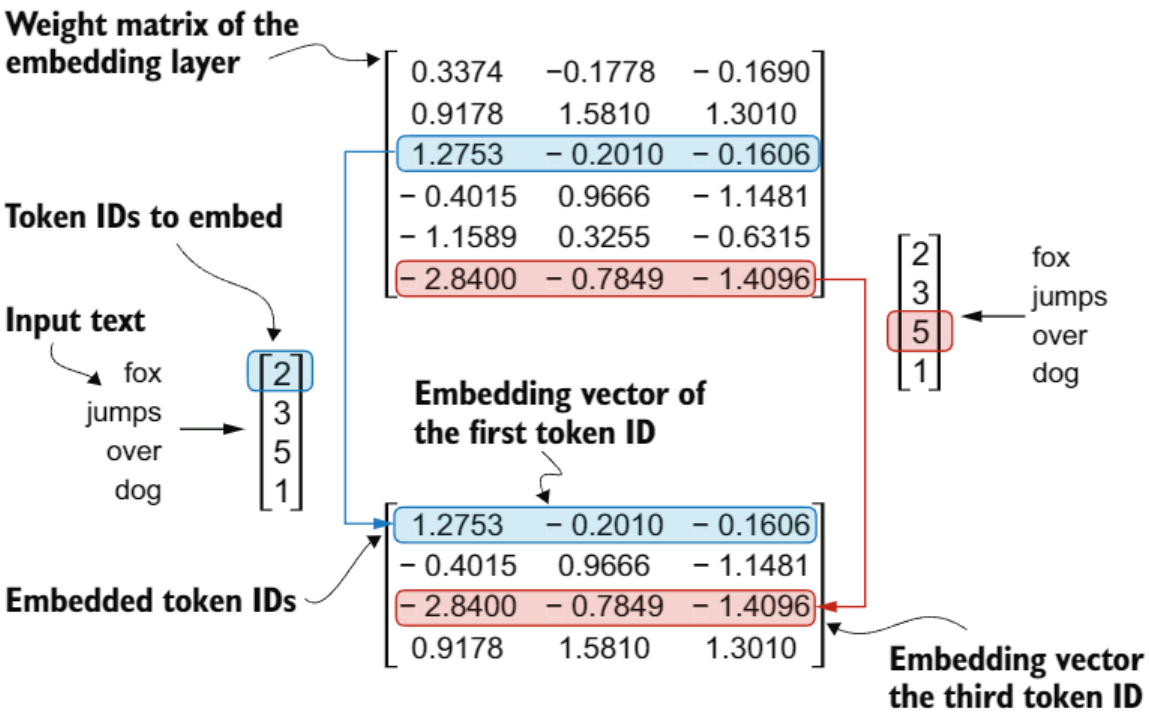

Figure 2.16 Embedding layers perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth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

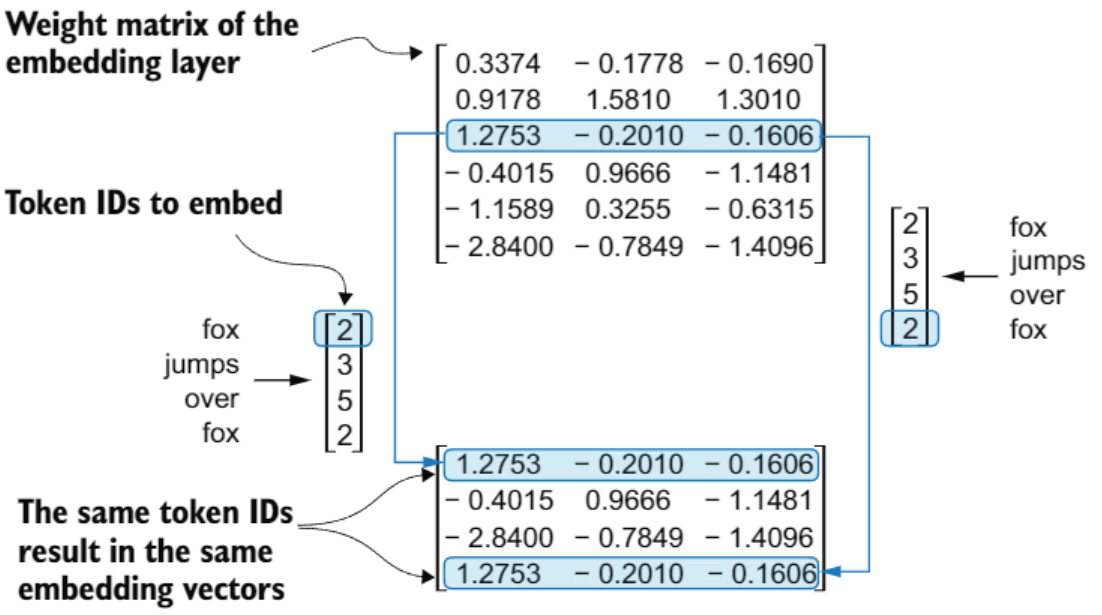

Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or fourth position in the token ID input vector, will result in the same embedding vector.

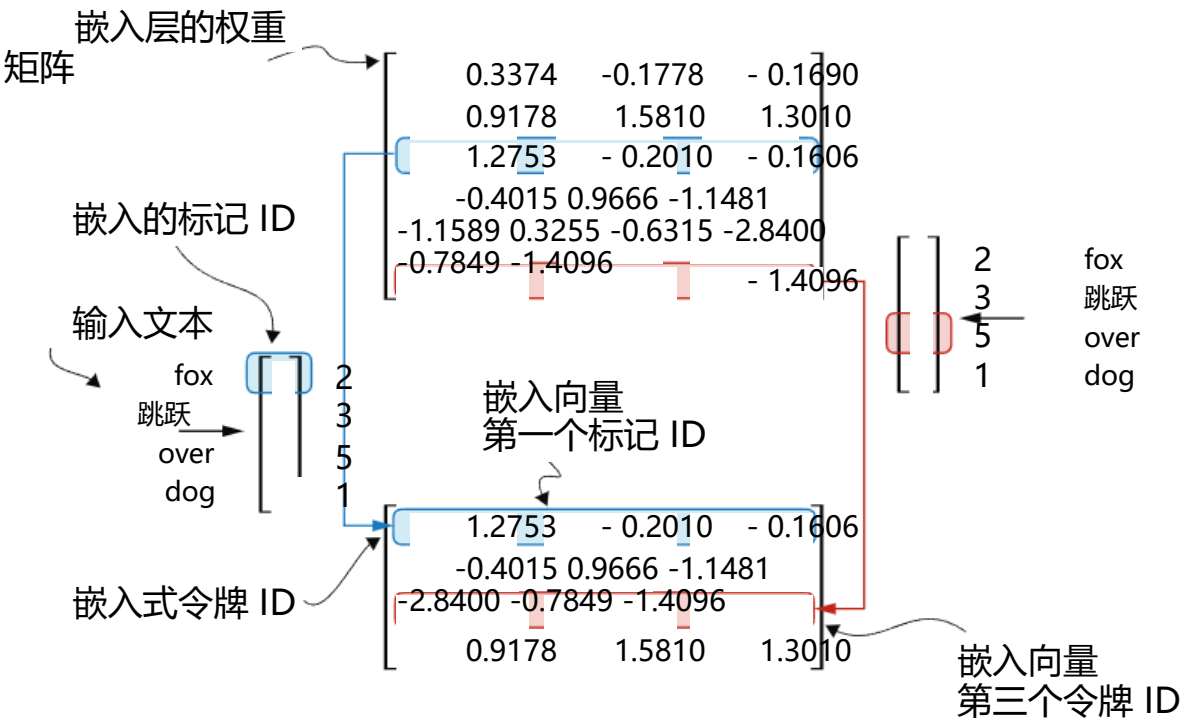

图 2.16 嵌入层执行查找操作，从嵌入层的权重矩阵中检索与标记 ID 对应的嵌入向量。例如，标记 ID 5 的嵌入向量是嵌入层权重矩阵的第六行（它是第六行而不是第五行，因为 Python 从 0 开始计数）。我们假设标记 ID 是由 2.3 节中的小词汇表生成的。

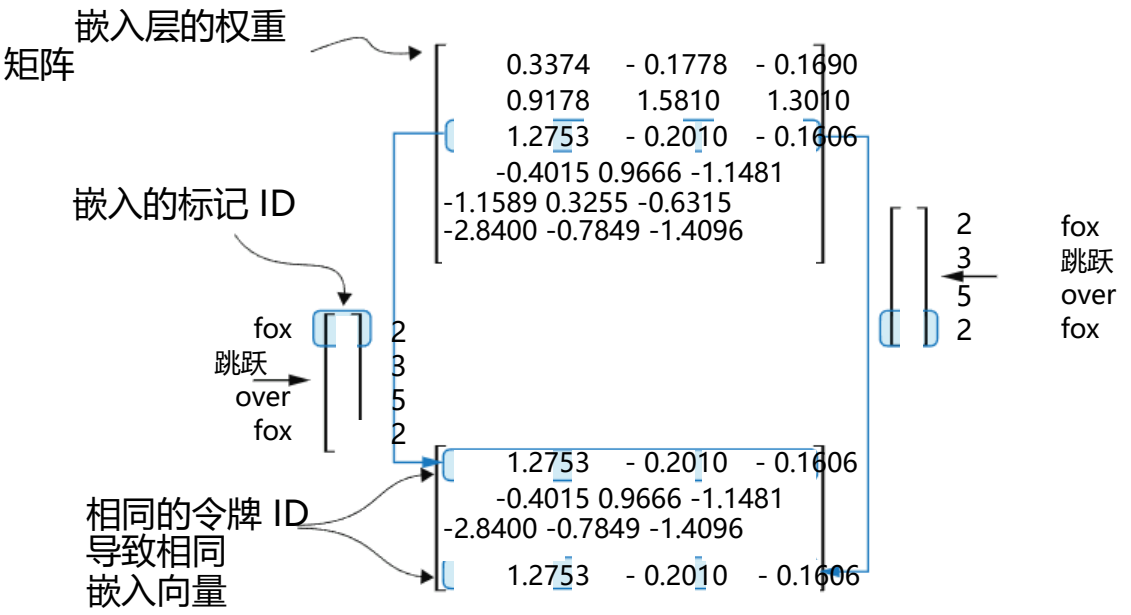

图 2.17 嵌入层将令牌 ID 转换为相同的向量表示，无论其在输入序列中的位置如何。例如，令牌 ID 5，无论是在令牌 ID 输入向量的第一个还是第四个位置，都会产生相同的嵌入向量。

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, we can use 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, and so on, as illustrated in figure 2.18.

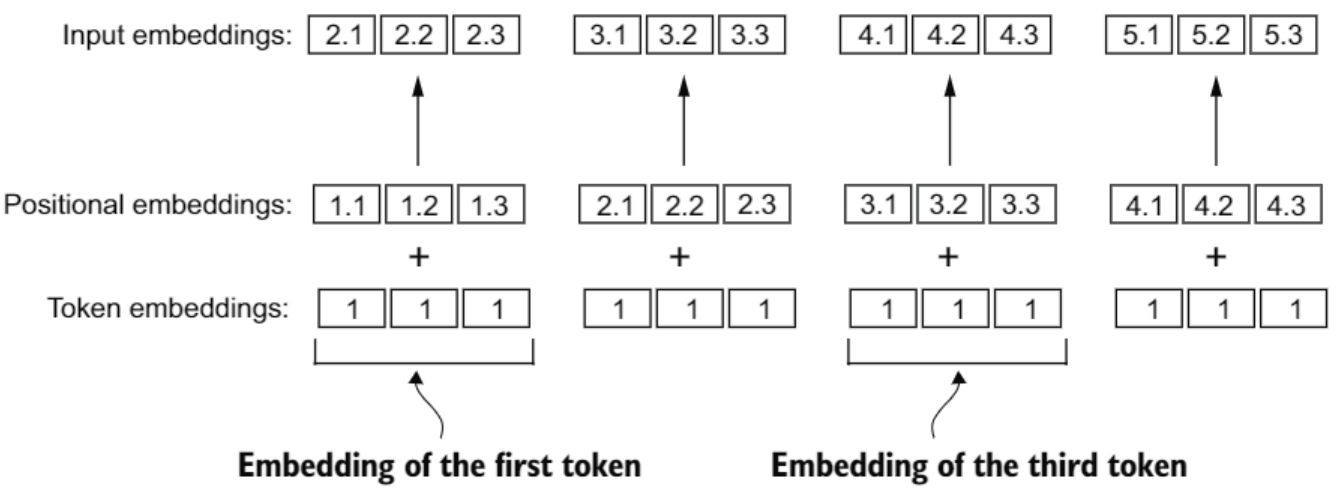

Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.

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 now, let's create the initial positional embeddings to create the LLM inputs.

原则上，令牌 ID 的确定性、位置无关的嵌入对于可重复性是有益的。然而，由于LLMs自身的自注意力机制也是位置无关的，向LLM注入额外的位置信息是有帮助的。

为了实现这一点，我们可以使用两种广泛类别的位置感知嵌入：相对位置嵌入和绝对位置嵌入。绝对位置嵌入直接与序列中的特定位置相关联。对于输入序列中的每个位置，都会添加一个唯一的嵌入到标记的嵌入中，以传达其确切位置。例如，第一个标记将具有特定的位置嵌入，第二个标记另一个不同的嵌入，依此类推，如图 2.18 所示。

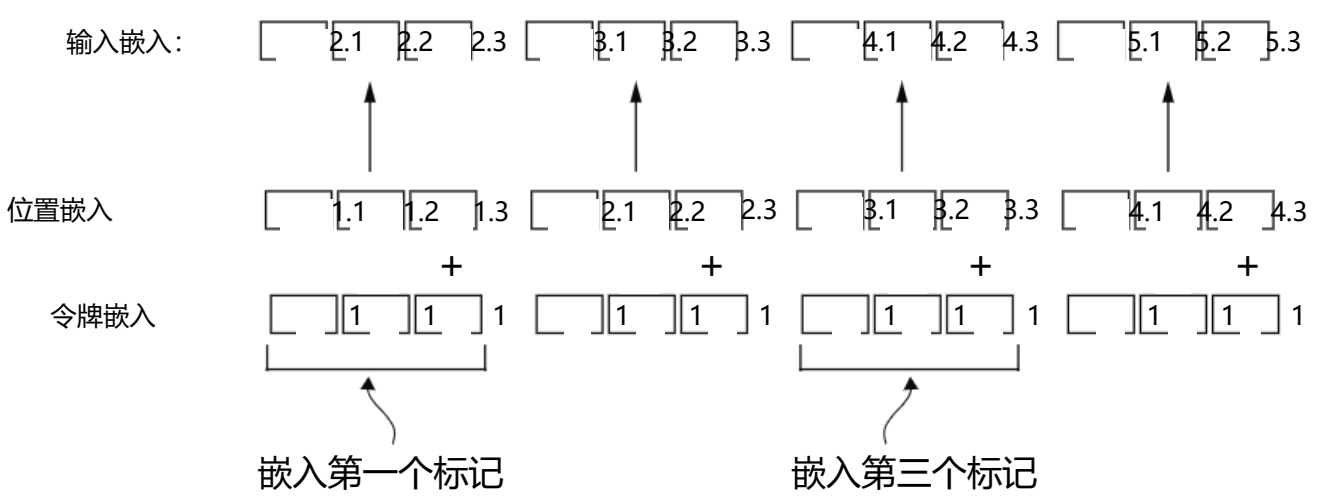

图 2.18 位置嵌入被添加到标记嵌入向量中，以创建LLM的输入嵌入。位置向量与原始标记嵌入具有相同的维度。为简单起见，标记嵌入显示为值 1。

代替关注标记的绝对位置，相对位置嵌入的焦点在于标记之间的相对位置或距离。这意味着模型通过“距离多远”来学习关系，而不是“在哪个确切位置”。这里的优势是，即使模型在训练期间没有看到过这样的长度，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旨在增强LLMs理解标记之间的顺序和关系的能力，确保更准确和上下文感知的预测。它们之间的选择通常取决于具体的应用和数据处理的性质。

OpenAI 的 GPT 模型使用的是在训练过程中优化的绝对位置嵌入，而不是像原始 Transformer 模型中的位置编码那样是固定的或预定义的。这个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生成LLM输入。

Previously, we focused on very small embedding sizes for simplicity. Now, let's consider more realistic and useful embedding sizes and encode the input tokens into a 256-dimensional vector representation, which is smaller than what the original GPT-3 model used (in GPT-3, the embedding size is 12,288 dimensions) but still reasonable for experimentation. Furthermore, we assume that the token IDs were created by the BPE tokenizer we implemented earlier, which has a vocabulary size of 50,257:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

Using the previous `token_embedding_layer`, if we sample data from the data loader, we embed each token in each batch into a 256-dimensional vector. If we have a batch size of 8 with four tokens each, the result will be an $8 \times 4 \times 256$ tensor.

Let's instantiate the data loader (see section 2.6) first:

```
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)
```

This code prints

```
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])
```

As we can see, the token ID tensor is 8×4 dimensional, meaning that the data batch consists of eight text samples with four tokens each.

Let's now use the embedding layer to embed these token IDs into 256-dimensional vectors:

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

之前，我们为了简单起见，专注于非常小的嵌入尺寸。现在，让我们考虑更现实和有用的嵌入尺寸，并将输入标记编码为 256 维向量表示，这比原始 GPT-3 模型使用的尺寸小（在 GPT-3 中，嵌入尺寸为 12,288 维）但仍然适合实验。此外，我们假设标记 ID 是由我们之前实现的 BPE 标记器创建的，其词汇量为 50,257：

```
vocab_size = 50257
输出维度 = 256 令牌嵌入层 = torch.nn.Embedding(vocab_size, output_dim)
```

使用之前的 token_embedding_layer，如果我们从数据加载器中采样数据，我们将每个批次中的每个标记嵌入到一个 256 维的向量中。如果我们有一个包含四个标记的 8 个批次的批次大小，结果将是一个 8 × 4 × 256 的张量。

首先实例化数据加载器（见第 2.6 节）：

```
最大长度 = 4
数据加载器 = create_dataloader_v1(
raw_text, 批处理大小=8, 最大长度=max_length, 步长
=max_length, 不打乱顺序) data_iter = iter(dataloader) 输入,
目标 = next(data_iter) 打印("标记 ID:\n", 输入) 打印("\n 输入
形状:\n", 输入.shape)
```

这段代码打印

```
Token IDs:
 张量([[       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]])

输入形状：
 torch.Size([8,])  4])
```

如您所见，标记 ID 张量是 8×4 维的，这意味着数据批次由八个文本样本组成，每个样本包含四个标记。

现在让我们使用嵌入层将这些标记 ID 嵌入到 256 维向量中：

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

The print function call returns

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

The 8 × 4 × 256–dimensional tensor output shows that each token ID is now embedded as a 256-dimensional vector.

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

```
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)
```

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.

The output of the print statement is

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

As we can see, the positional embedding tensor consists of four 256-dimensional vectors. We can now add these directly to the token embeddings, where PyTorch will add the 4 × 256–dimensional `pos_embeddings` tensor to each 4 × 256–dimensional token embedding tensor in each of the eight batches:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

The print output is

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

The `input_embeddings` we created, as summarized in figure 2.19, are the embedded input examples that can now be processed by the main LLM modules, which we will begin implementing in the next chapter.

打印函数调用返回

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

8 × 4 × 256 维度的张量输出表明，每个标记 ID 现在都嵌入为一个 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_相同的嵌入维度的嵌入层

层：

```
context_length = 最大长度 pos_embedding_layer = torch.nn.Embedding(context_length, 输出维度) pos_embeddings = pos_embedding_layer(torch.arange(context_length)) print(pos_embeddings.shape)
```

输入到 pos_embeddings 的通常是占位符向量 torch.arange(context_length)，其中包含从 0 到最大输入长度-1 的数字序列。context_length 是一个表示LLM支持的输入大小的变量。在这里，我们选择它与输入文本的最大长度相似。在实践中，输入文本可能比支持的内容长度长，在这种情况下，我们必须截断文本。

打印语句的输出是

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

如您所见，位置嵌入张量由四个 256 维向量组成。现在我们可以直接将这些向量添加到标记嵌入中，PyTorch 会将 4×256 维的 pos_embeddings 张量添加到每个 8 个批次中的每个 4×256 维标记嵌入张量中：

```
输入嵌入 = 标记嵌入 + 词性嵌入 print(input_embeddings.shape)
```

打印输出

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

我们创建的输入嵌入，如图 2.19 所示，是现在可以被主LLM模块处理的嵌入输入示例，我们将在下一章开始实现这些模块。

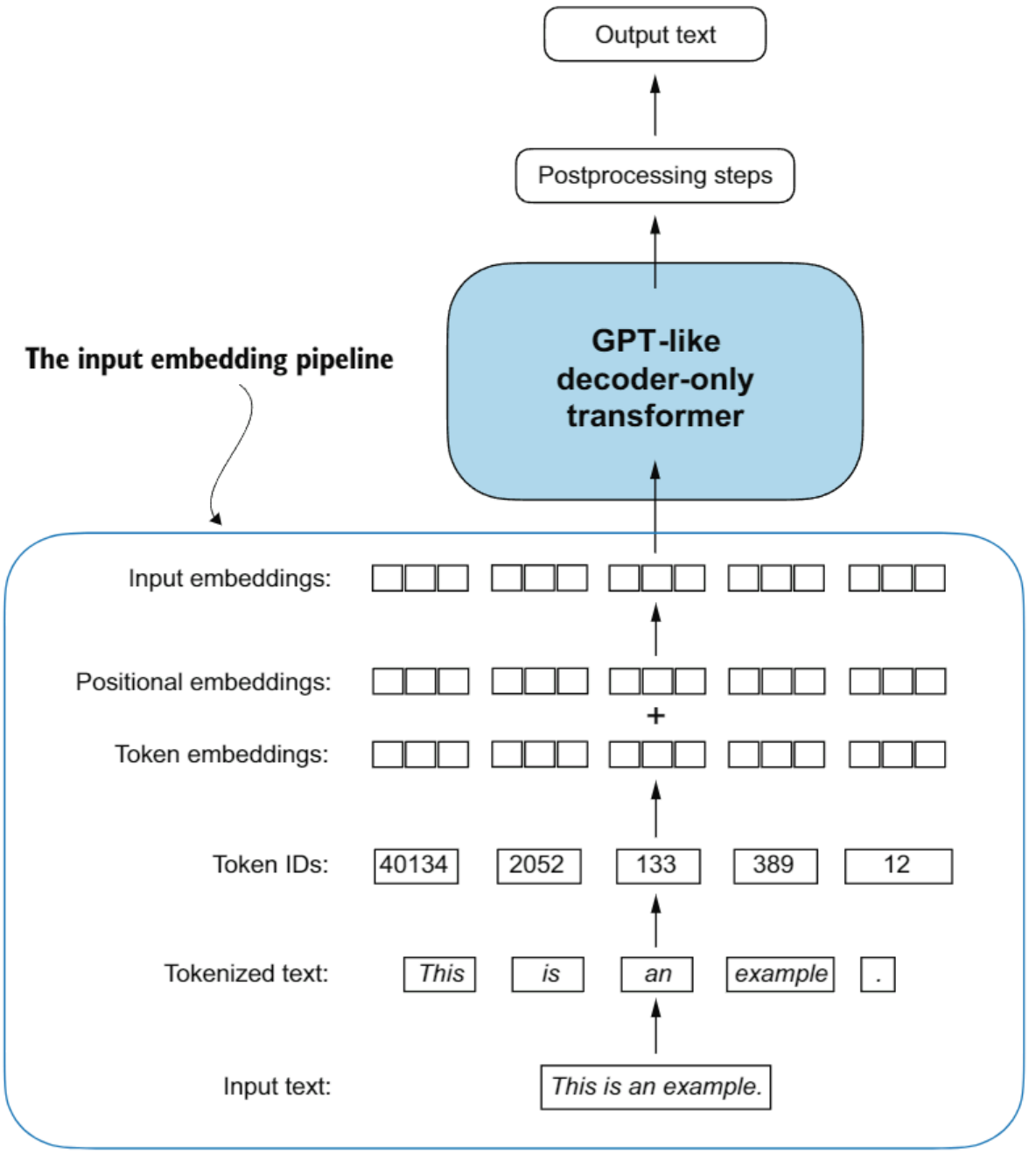

Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input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

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 or characters. 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.

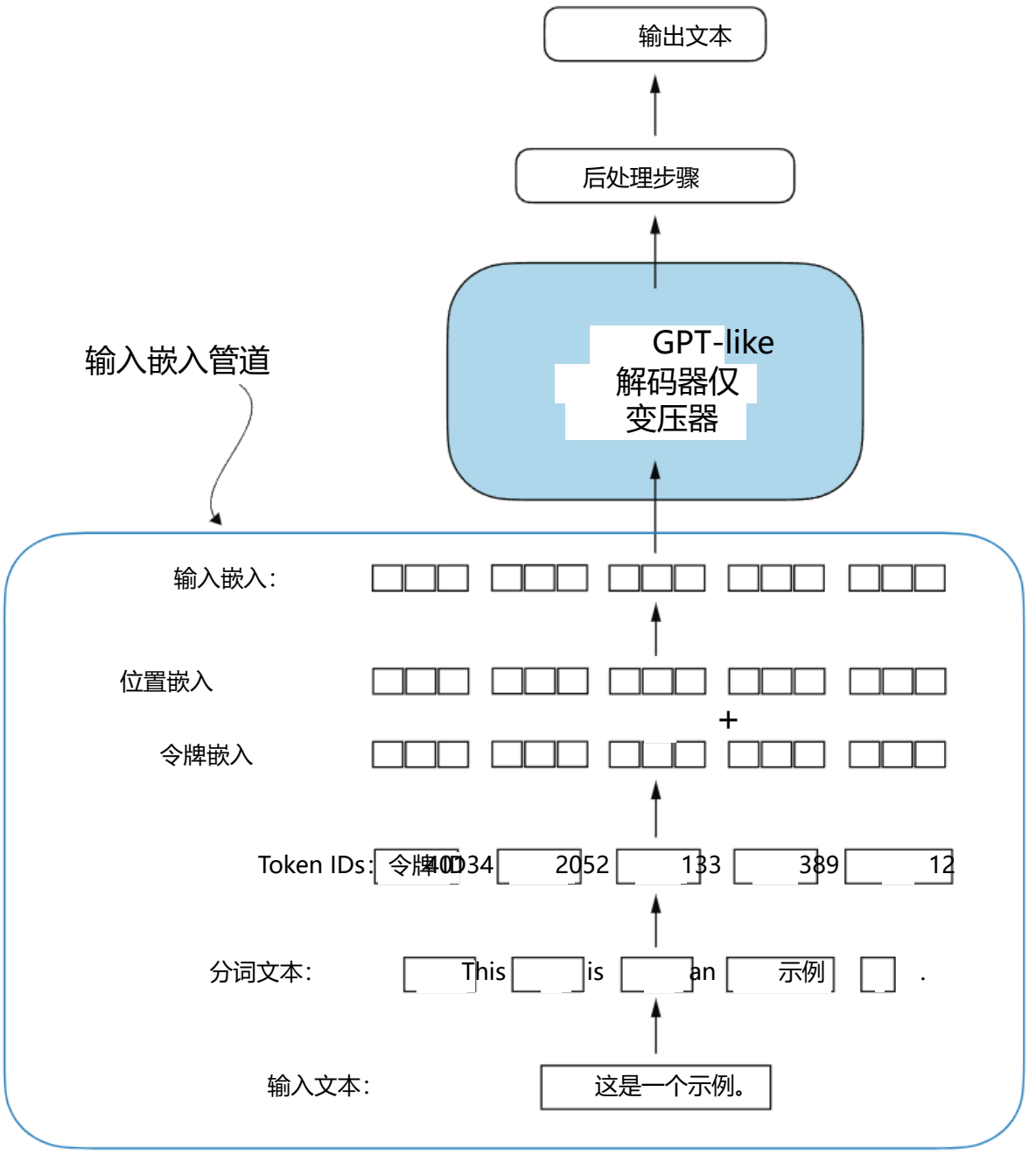

图 2.19 作为输入处理管道的一部分，输入文本首先被分解成单个标记。然后，这些标记使用词汇表转换为标记 ID。这些标记 ID 被转换为嵌入向量，并添加了类似大小的位置嵌入，从而得到作为主LLM层输入的输入嵌入。

摘要

- 需要将文本数据转换为数值向量，称为嵌入，因为它们无法处理原始文本。嵌入将离散数据（如单词或图像）转换为连续向量空间，使它们与神经网络操作兼容。

- 作为第一步，原始文本被分解成标记，这些标记可以是单词或字符。然后，标记被转换为整数表示，称为标记 ID。
- 特殊标记，如<|unk|>和逗号，可以添加以增强模型对各种上下文的理解和处理，例如未知单词或标记无关文本之间的边界。

- 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, which are added to the token embedding vectors and are optimized during the model training.

- 字节对编码（BPE）分词器，用于像 GPT-2 和 GPT-3 这样的LLMs，可以有效地通过将未知单词分解为子词单元或单个字符来处理未知单词。

- 我们使用滑动窗口方法对分词数据进行处理，以生成用于LLM训练的输入-目标对。

- 嵌入层在 PyTorch 中作为查找操作，检索与标记 ID 对应的向量。生成的嵌入向量提供了标记的连续表示，这对于训练像LLMs这样的深度学习模型至关重要。

- 虽然标记嵌入为每个标记提供了一致的向量表示，但它们缺乏对标记在序列中位置的感知。为了纠正这一点，存在两种主要类型的定位嵌入：绝对和相对。OpenAI 的 GPT 模型使用绝对定位嵌入，这些嵌入被添加到标记嵌入向量中，并在模型训练过程中进行优化。