5 Pretraining on unlabeled data

This chapter covers

- Computing the training and validation set losses to assess the quality of LLM-generated text during training
- Implementing a training function and pretraining the LLM
- Saving and loading model weights to continue training an LLM
- Loading pretrained weights from OpenAI

Thus far, we have implemented the data sampling and attention mechanism and coded the LLM architecture. It is now time to implement a training function and pretrain the LLM. We will learn about basic model evaluation techniques to measure the quality of the generated text, which is a requirement for optimizing the LLM during the training process. Moreover, we will discuss how to load pretrained weights, giving our LLM a solid starting point for fine-tuning. Figure 5.1 lays out our overall plan, highlighting what we will discuss in this chapter.

5 预训练
在未标记数据上

本章涵盖

- 计算训练集和验证集的损失，以评估在训练过程中LLM-生成的文本的质量
- 实现训练函数和预训练LLM
- 保存和加载模型权重以继续训练 LLM
- 从 OpenAI 加载预训练权重

截至目前，我们已经实现了数据采样和注意力机制，并编码了LLM架构。现在是时候实现训练函数并预训练LLM了。我们将学习基本模型评估技术来衡量生成文本的质量，这是在训练过程中优化LLM的要求。此外，我们将讨论如何加载预训练权重，为我们LLM的微调提供一个坚实的基础。图 5.1 概述了我们的整体计划，突出了本章将讨论的内容。

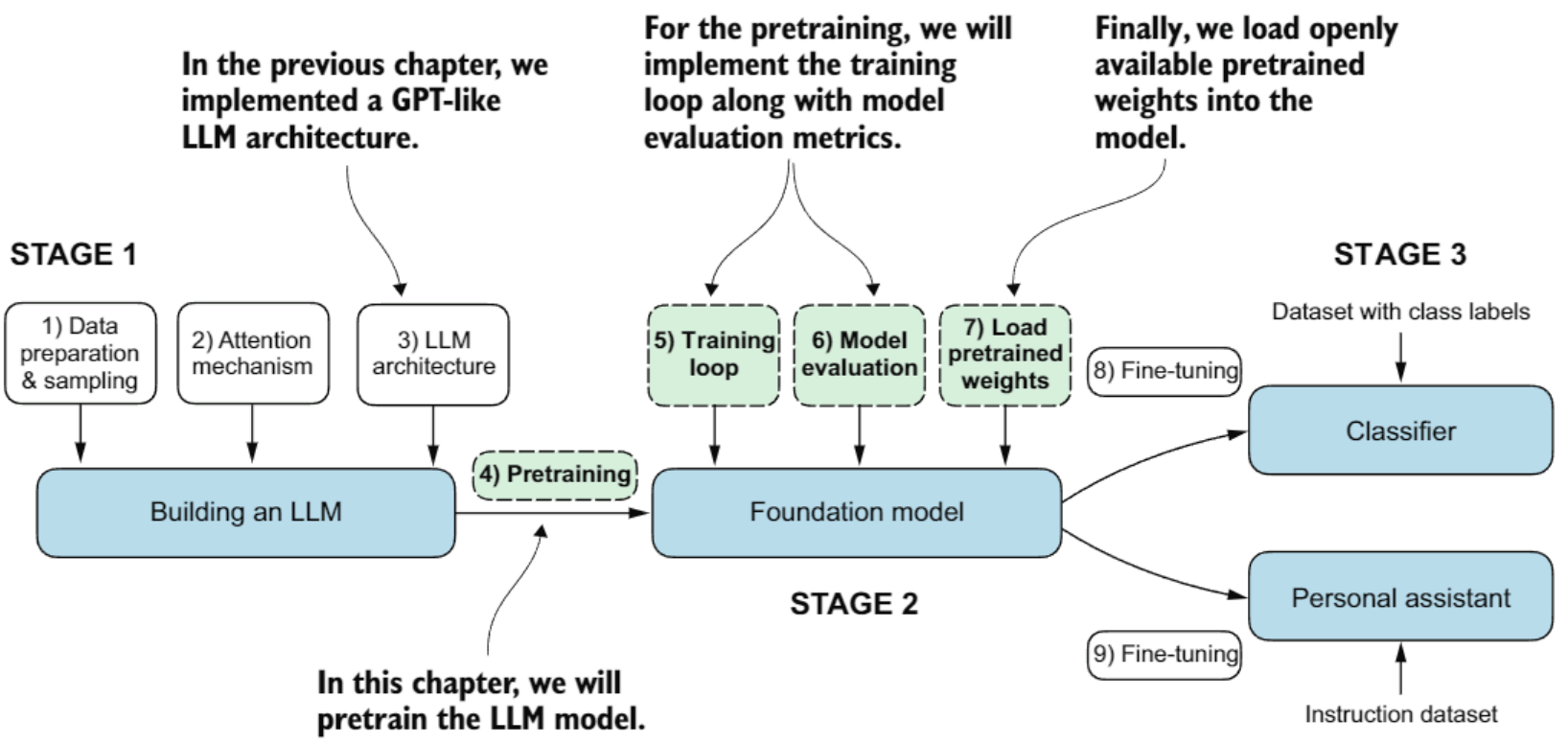

Figure 5.1 The three main stages of coding an LLM. This chapter focuses on stage 2: pretraining the LLM (step 4), which includes implementing the training code (step 5), evaluating the performance (step 6), and saving and loading model weights (step 7).

Weight parameters

In the context of LLMs and other deep learning models, *weights* refer to the trainable parameters that the learning process adjusts. These weights are also known as *weight parameters* or simply *parameters*. In frameworks like PyTorch, these weights are stored in linear layers; we used these to implement the multi-head attention module in chapter 3 and the `GPTModel` in chapter 4. After initializing a layer (`new_layer = torch.nn.Linear(...)`), we can access its weights through the `.weight` attribute, `new_layer.weight`. Additionally, for convenience, PyTorch allows direct access to all a model's trainable parameters, including weights and biases, through the method `model.parameters()`, which we will use later when implementing the model training.

5.1 Evaluating generative text models

After briefly recapping the text generation from chapter 4, we will set up our LLM for text generation and then discuss basic ways to evaluate the quality of the generated text. We will then calculate the training and validation losses. Figure 5.2 shows the topics covered in this chapter, with these first three steps highlighted.

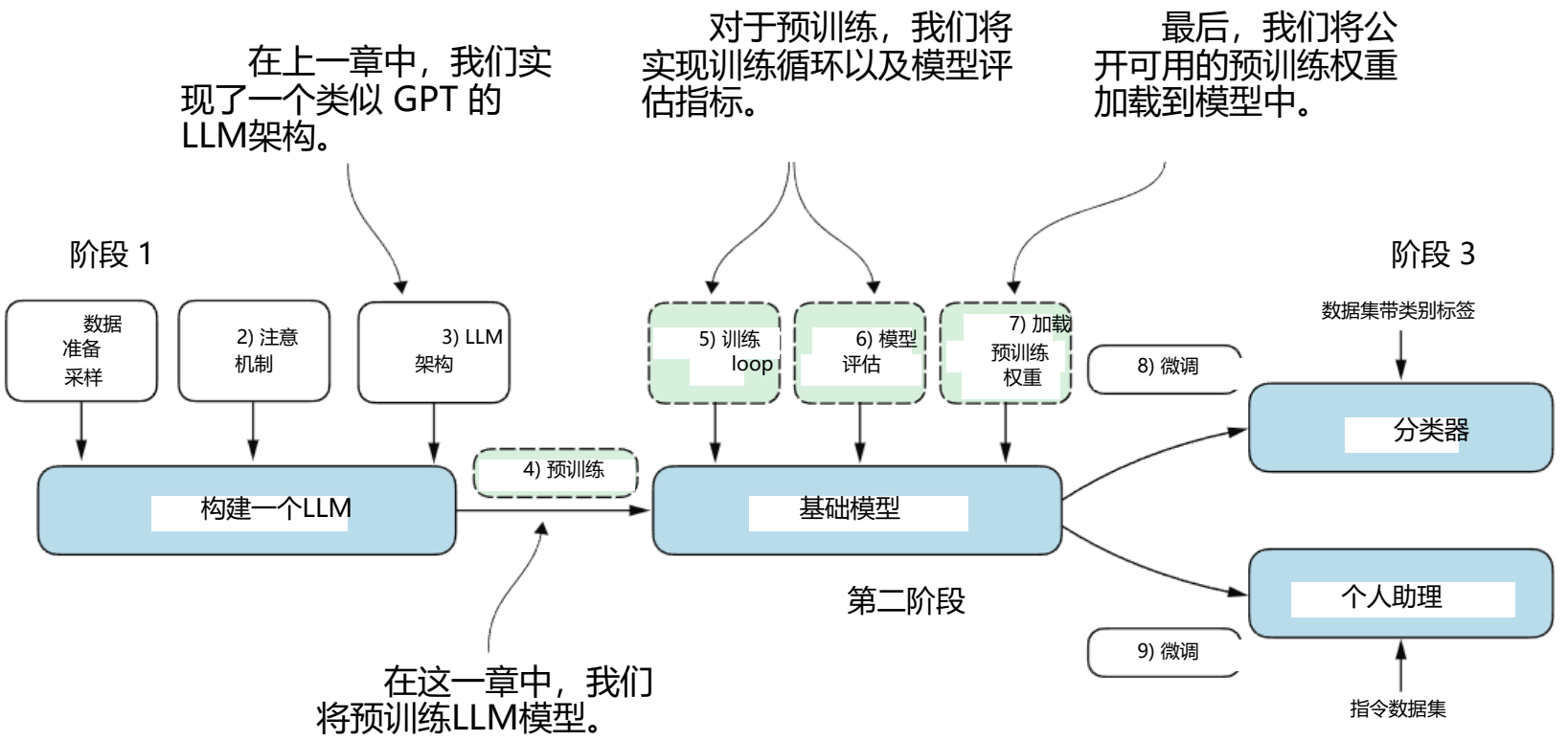

图 5.1 编码LLM的三个主要阶段。本章重点介绍第二阶段：预训练LLM（步骤 4），包括实现训练代码（步骤 5）、评估性能（步骤 6）和保存和加载模型权重（步骤 7）。

权重参数

在LLMs和其他深度学习模型中，权重指的是学习过程调整的可训练参数。这些权重也被称为权重参数或简单地称为参数。在 PyTorch 等框架中，这些权重存储在线性层中；我们使用这些来在第三章实现多头注意力模块和在第四章实现 GPTModel。初始化一个层（new_layer = torch.nn.Linear(...)）后，我们可以通过.weight 属性访问其权重。

直接访问所有模型的训练参数比如，包括权重和偏差更通过方法 PyTorch 允许model.parameters()，我们将在实现模型训练时使用。

5.1 评估生成文本模型

简要回顾第 4 章的文本生成后，我们将设置我们的LLM进行文本生成，然后讨论评估生成文本质量的基本方法。接着，我们将计算训练和验证损失。图 5.2 展示了本章涵盖的主题，并突出了这三个步骤。

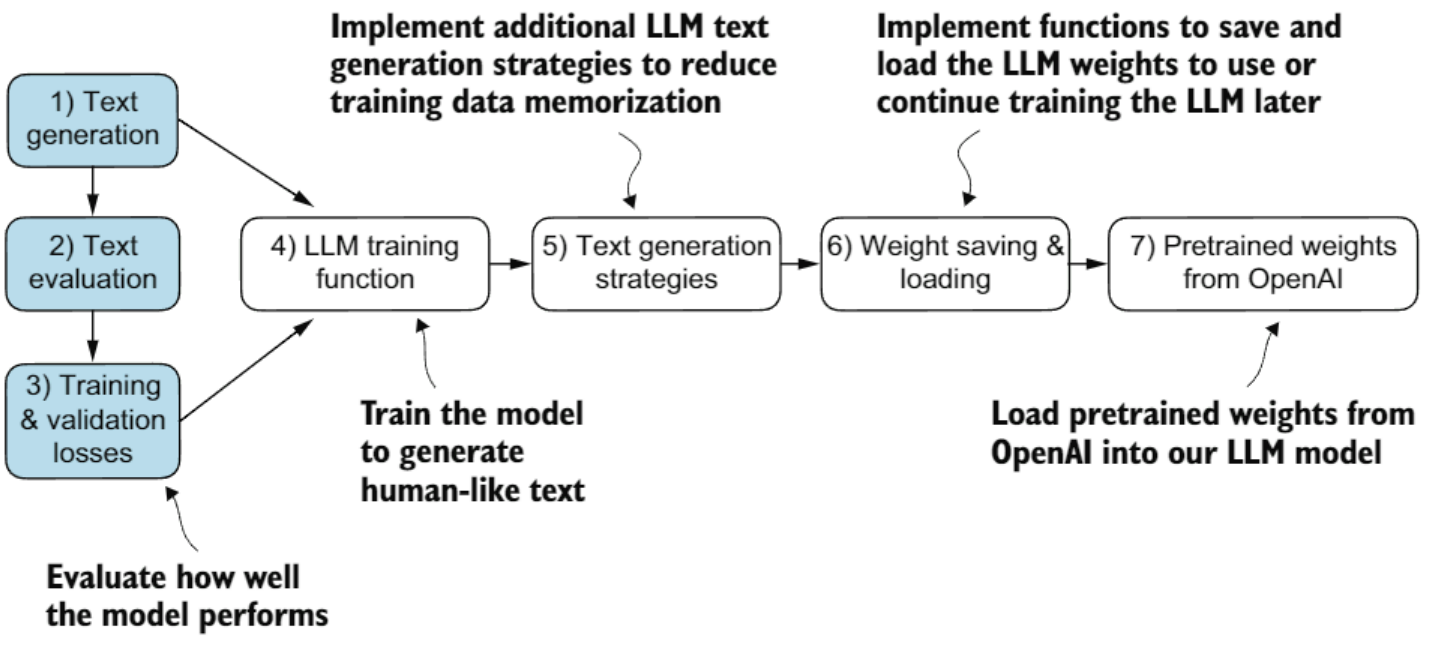

Figure 5.2 An overview of the topics covered in this chapter. We begin by recapping text generation (step 1) before moving on to discuss basic model evaluation techniques (step 2) and training and validation losses (step 3).

5.1.1 Using GPT to generate text

Let's set up the LLM and briefly recap the text generation process we implemented in chapter 4. We begin by initializing the GPT model that we will later evaluate and train using the `GPTModel` class and `GPT_CONFIG_124M` dictionary (see chapter 4):

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

We shorten the context length from 1,024 to 256 tokens.

It's possible and common to set dropout to 0.

Considering the `GPT_CONFIG_124M` dictionary, the only adjustment we have made compared to the previous chapter is that we have reduced the context length (`context_length`) to 256 tokens. This modification reduces the computational demands of training the model, making it possible to carry out the training on a standard laptop computer.

Originally, the GPT-2 model with 124 million parameters was configured to handle up to 1,024 tokens. After the training process, we will update the context size setting

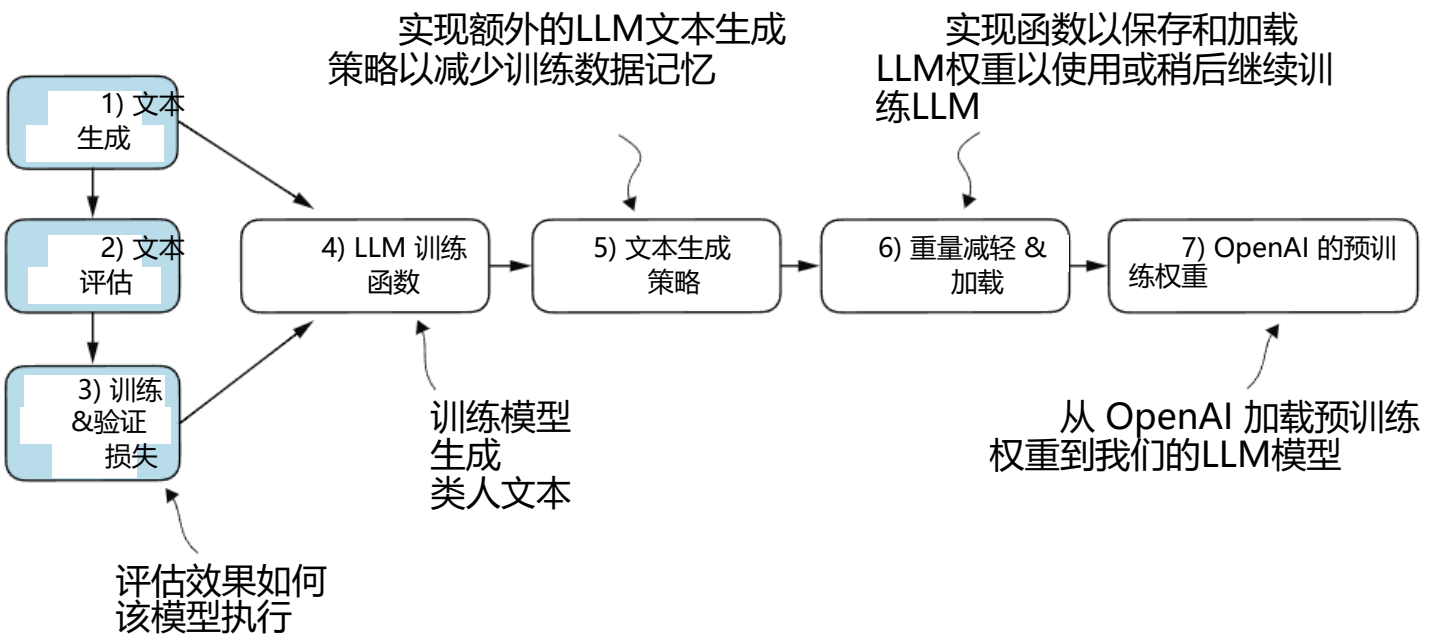

图 5.2 本章涵盖主题概述。我们首先回顾文本生成（步骤 1），然后讨论基本模型评估技术（步骤 2）以及训练和验证损失（步骤 3）。

5.1.1 使用 GPT 生成文本

让我们设置LLM，并简要回顾我们在第 4 章中实现的文本生成过程。我们首先通过 GPTModel 类和 GPT_CONFIG_124M 字典（见第 4 章）初始化我们将要评估和训练的 GPT 模型：

```
    import torch
from chapter04 import GPTModel


    GPT_CONFIG_124M = {
    "vocab_size": 50257,
"context_length": 256, "emb_dim": 768,
"n_heads": 12, "n_layers": 12,
"drop_rate": 0.1, "qkv_bias": False }
torch.manual_seed(123) 模型 =
GPTModel(GPT_CONFIG_124M) 模型.eval()
```

我们缩短了上下文长度从 1,024 到 256 个标记。

它将 dropout 设置为 0 是可能的且很常见。

考虑到 GPT_CONFIG_124M 字典，与上一章相比，我们唯一做出的调整是将上下文长度（context_）减少了

长度限制为 256 个标记。此修改降低了训练模型的计算需求，使得在标准笔记本电脑上执行训练成为可能。

最初，具有 1.24 亿参数的 GPT-2 模型被配置为处理多达 1024 个标记。在训练过程之后，我们将更新上下文大小设置

and load pretrained weights to work with a model configured for a 1,024-token context length.

Using the `GPTModel` instance, we adopt the `generate_text_simple` function from chapter 4 and introduce two handy functions: `text_to_token_ ids` and `token_ids_to_text`. These functions facilitate the conversion between text and token representations, a technique we will utilize throughout this chapter.

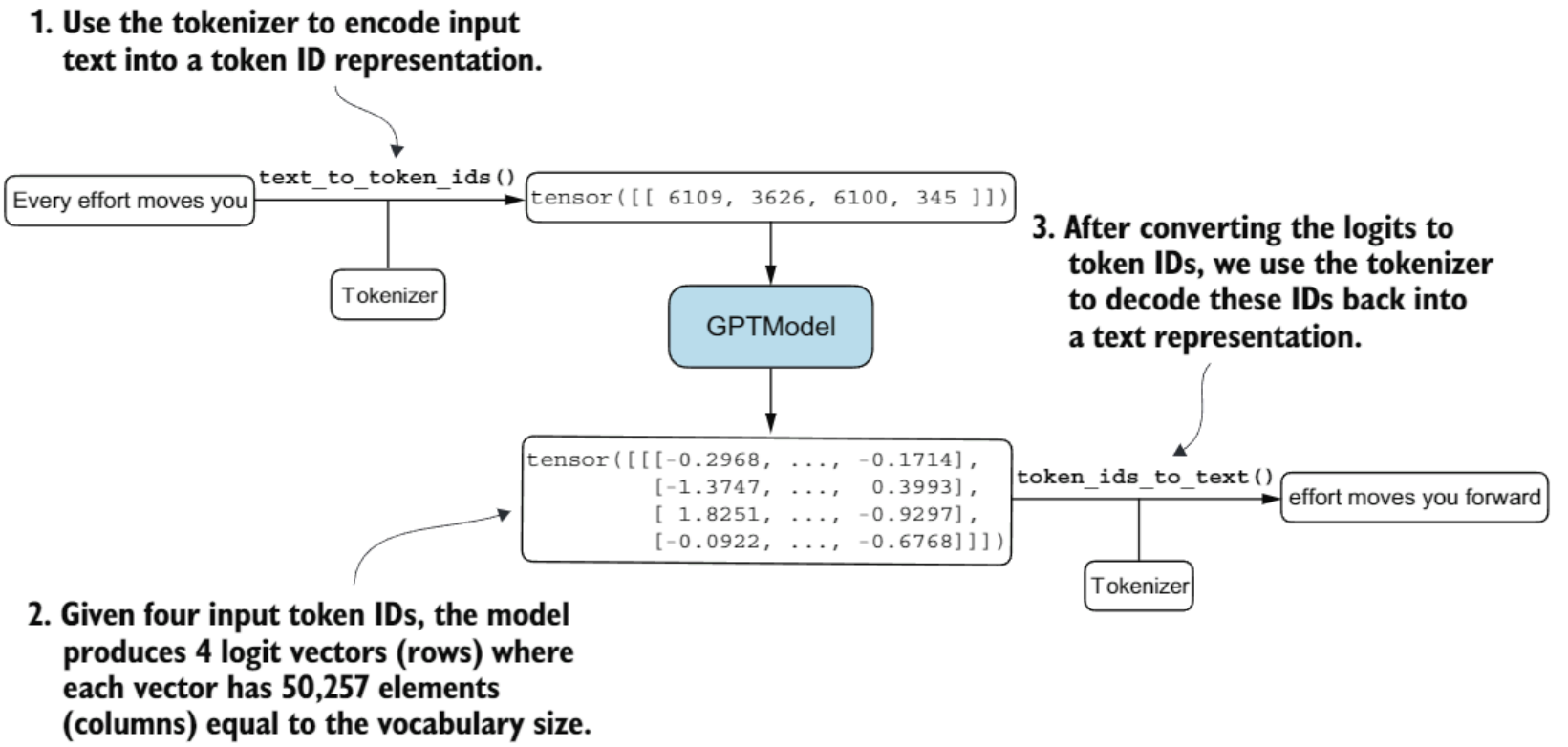

Figure 5.3 Generating text involves encoding text into token IDs that the LLM processes into logit vectors. The logit vectors are then converted back into token IDs, detokenized into a text representation.

Figure 5.3 illustrates a three-step text generation process using a GPT model. First, the tokenizer converts input text into a series of token IDs (see chapter 2). Second, the model receives these token IDs and generates corresponding logits, which are vectors representing the probability distribution for each token in the vocabulary (see chapter 4). Third, these logits are converted back into token IDs, which the tokenizer decodes into human-readable text, completing the cycle from textual input to textual output.

We can implement the text generation process, as shown in the following listing.

Listing 5.1 Utility functions for text to token ID conversion

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
```

将预训练权重加载到配置为 1,024 个 token 上下文长度的模型中。

使用 GPTModel 实例，我们采用第 4 章中的 generate_text_simple 函数，并引入两个便捷函数：text_to_token_ids 和 token_ids_to_text。这些函数便于在文本和标记表示之间进行转换，我们将在本章中利用这一技术。

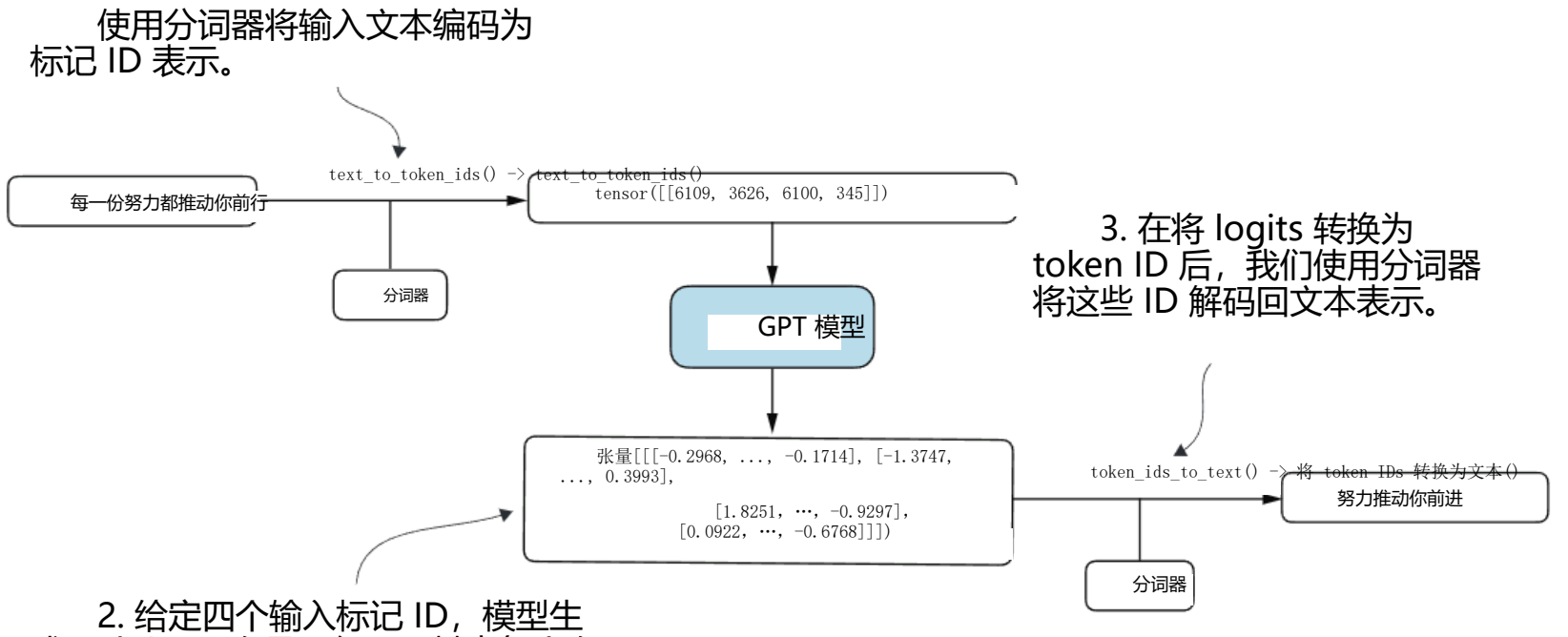

图 5.3 生成文本涉及将文本编码为 token ID，LLM将这些 token ID 处理成 logit 向量。然后，将这些 logit 向量转换回 token ID，并反序列化为文本表示。

图 5.3 展示了使用 GPT 模型的三步文本生成过程。首先，分词器将输入文本转换为一系列的标记 ID（见第 2 章）。其次，模型接收这些标记 ID 并生成相应的 logits，这些 logits 是表示词汇表中每个标记概率分布的向量（见第 4 章）。第三，这些 logits 被转换回标记 ID，分词器将其解码为可读文本，从而完成从文本输入到文本输出的循环。

我们可以实现文本生成过程，如下所示列表。

列表 5.1 文本到 token ID 的实用函数

```
导入 tiktoken
from  第四章 导入 generate_text_simple

def text_to_token_ids(text, tokenizer): # 将文本转换为 token ID 列表，其中 text 是待转换的文本，tokenizer 是用于分词的标记化器
    编码 = 分词器对文本进行编码，允许的特殊字符集为空。
```

```
    encoded_tensor = torch.tensor(encoded).unsqueeze(0)
    return encoded_tensor


def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0)
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

.unsqueeze(0) adds the batch dimension

Removes batch dimension

Using this code, the `model` generates the following text:

```
Output text:
 Every effort moves you rentingetic wasn? refres RexMeCHicular stren
```

Clearly, the model isn't yet producing coherent text because it hasn't undergone training. To define what makes text "coherent" or "high quality," we have to implement a numerical method to evaluate the generated content. This approach will enable us to monitor and enhance the model's performance throughout its training process.

Next, we will calculate a *loss metric* for the generated outputs. This loss serves as a progress and success indicator of the training progress. Furthermore, in later chapters, when we fine-tune our LLM, we will review additional methodologies for assessing model quality.

5.1.2 Calculating the text generation loss

Next, let's explore techniques for numerically assessing text quality generated during training by calculating a *text generation loss*. We will go over this topic step by step with a practical example to make the concepts clear and applicable, beginning with a short recap of how the data is loaded and how the text is generated via the `generate_text_simple` function.

Figure 5.4 illustrates the overall flow from input text to LLM-generated text using a five-step procedure. This text-generation process shows what the `generate_text_simple` function does internally. We need to perform these same initial steps before we can compute a loss that measures the generated text quality later in this section.

Figure 5.4 outlines the text generation process with a small seven-token vocabulary to fit this image on a single page. However, our `GPTModel` works with a much larger

```
        encoded_tensor = torch.tensor(encoded).unsqueeze(0) 返回
     encoded_tensor


    def token_ids_to_text(token_ids, tokenizer): 将 token_ids 转换为文本
        flat = token_ids.squeeze(0)
     返回        tokenizer.decode(flat.tolist())

    }) start_context = "每一点努力都让你前进"
tokenizer = tiktoken.get_encoding("gpt2")


    token_ids = generate_text_simple(
    model=model, idx=text_to_token_ids(start_context,
tokenizer), max_new_tokens=10,
context_size=GPT_CONFIG_124M["context_length"]) 打印("输出")



                    文本: ", 将 token_ids 转换为文本(token_ids_to_text(token_ids, tokenizer))"
```

.unsqueeze(0) 保留原文
添加批次
维度

移除批维
度

使用此代码，模型生成了以下文本：

```
输出文本:
 每个    努力     您租的“rentingetic”是什么？      刷新     RexMeCHicular    强烈
```

显然，该模型尚未产生连贯的文本，因为它还没有经过训练。为了定义什么使文本“连贯”或“高质量”，我们必须实施一种数值方法来评估生成的内容。这种方法将使我们能够在其训练过程中监控和提升模型的表现。

接下来，我们将计算生成输出的损失指标。这个损失指标作为训练进度的进度和成功指标。此外，在后面的章节中，当我们微调我们的LLM时，我们将回顾评估模型质量的额外方法。

5.1.2 计算文本生成损失

接下来，我们将探讨在训练过程中通过计算文本生成损失来数值评估生成文本质量的技术。我们将通过一个实际例子逐步讲解这个主题，从简要回顾数据加载和文本生成的方式开始。

生成文本简单函数。

图 5.4 展示了从输入文本到LLM生成的文本的整体流程，采用五步程序。此文本生成过程展示了 generate_text_simple 函数内部执行的操作。在我们能够计算衡量生成文本质量的损失之前，我们需要执行这些相同的初始步骤。

图 5.4 概述了使用一个小型七个词元的词汇表来适应此图像的单页文本生成过程。然而，我们的 GPTModel 与一个更大的词汇表一起工作。

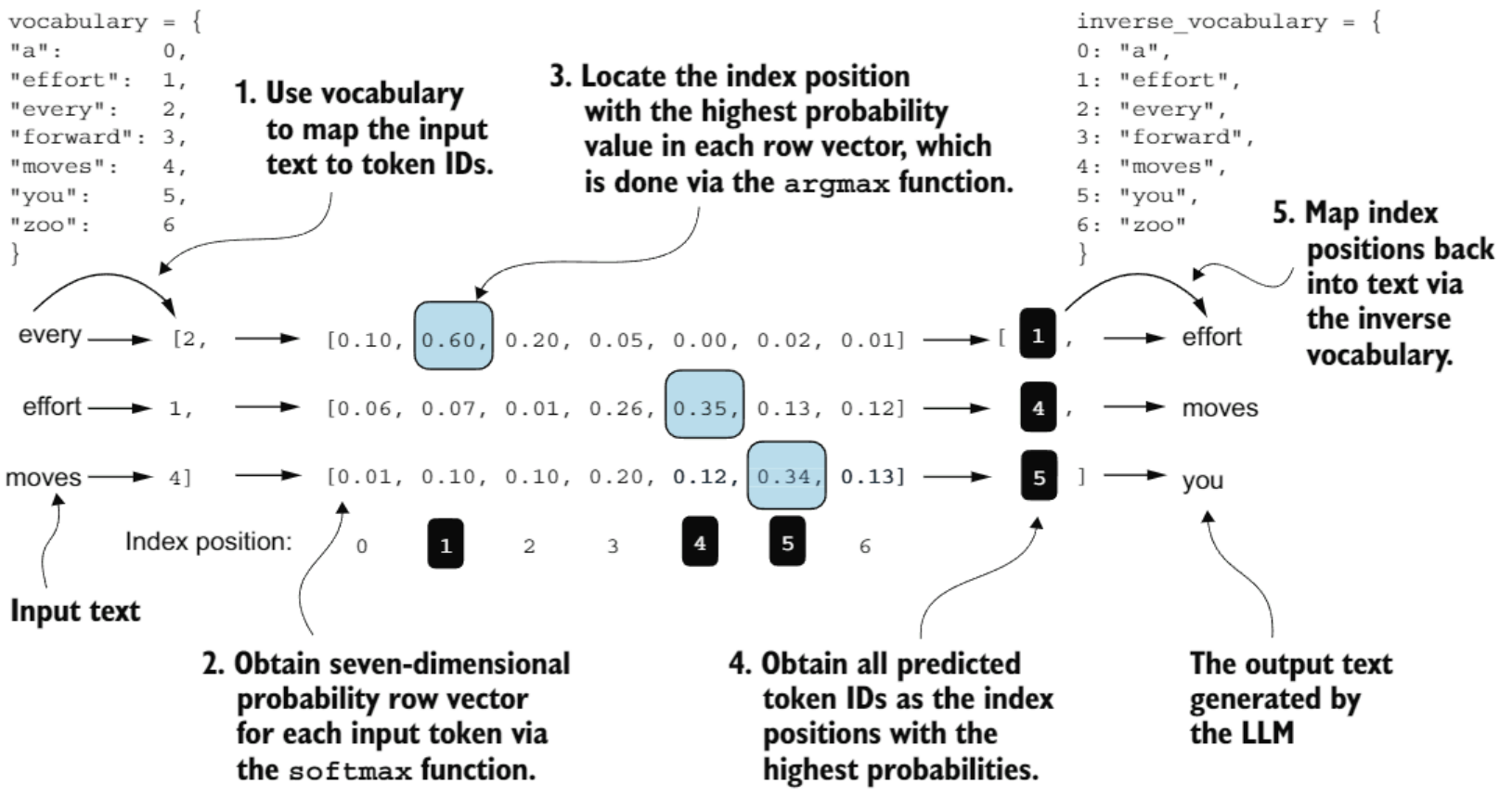

Figure 5.4 For each of the three input tokens, shown on the left, we compute a vector containing probability scores corresponding to each token in the vocabulary. The index position of the highest probability score in each vector represents the most likely next token ID. These token IDs associated with the highest probability scores are selected and mapped back into a text that represents the text generated by the model.

vocabulary consisting of 50,257 words; hence, the token IDs in the following code will range from 0 to 50,256 rather than 0 to 6.

Also, figure 5.4 only shows a single text example (`"every effort moves"`) for simplicity. In the following hands-on code example that implements the steps in the figure, we will work with two input examples for the GPT model (`"every effort moves"` and `"I really like"`).

Consider these two input examples, which have already been mapped to token IDs (figure 5.4, step 1):

```
inputs = torch.tensor([[16833, 3626, 6100],   # ["every effort moves",
                       [40,    1107, 588]])   #  "I really like"]
```

Matching these inputs, the `targets` contain the token IDs we want the model to produce:

```
targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [1107, 588, 11311]])  #  " really like chocolate"]
```

Note that the targets are the inputs but shifted one position forward, a concept we covered in chapter 2 during the implementation of the data loader. This shifting strategy is crucial for teaching the model to predict the next token in a sequence.

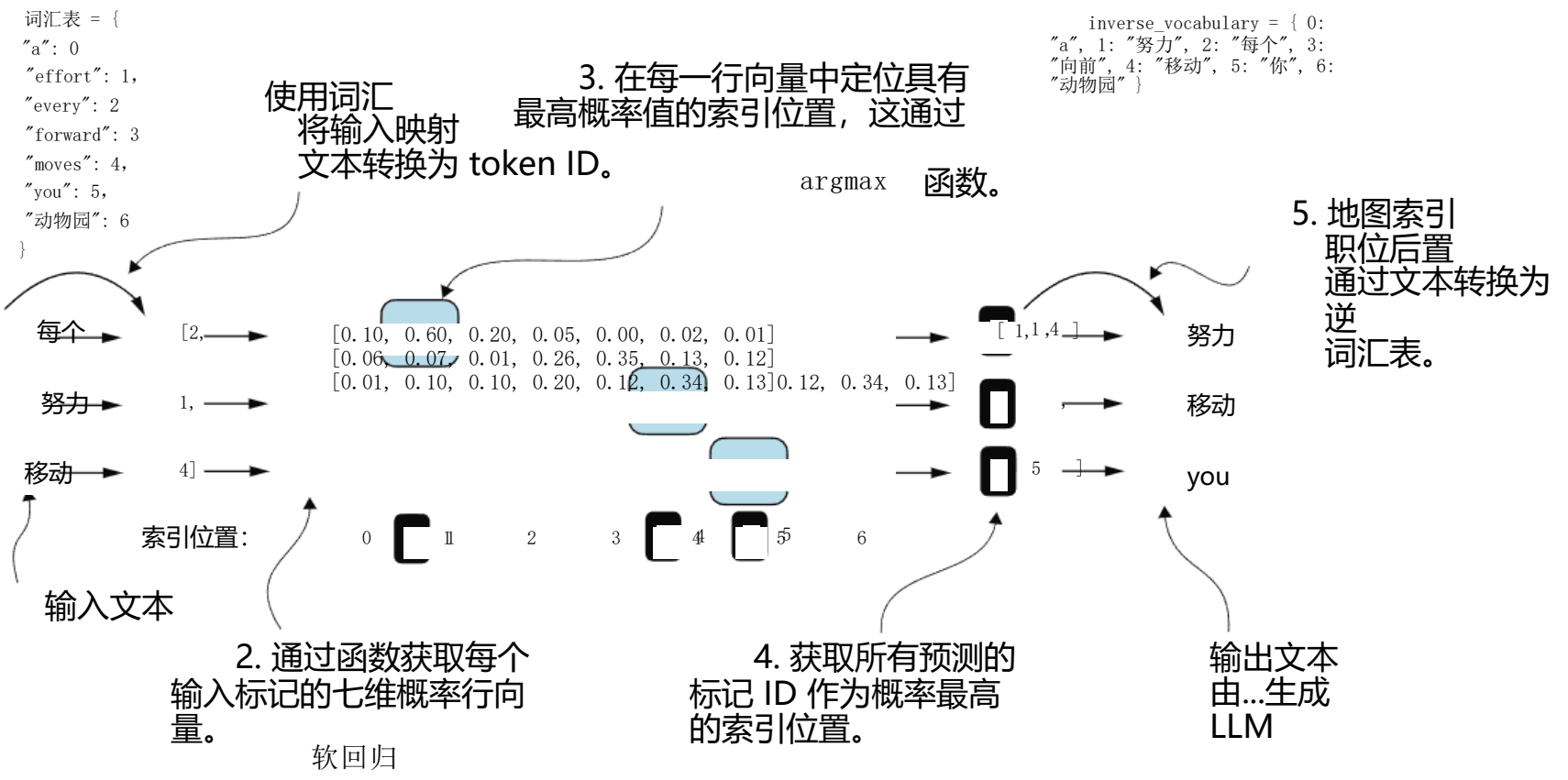

图 5.4 对于左侧显示的每个三个输入标记，我们计算一个包含对应于词汇表中每个标记的概率得分的向量。每个向量中最高概率得分的索引位置代表最可能的下一个标记 ID。与最高概率得分相关的这些标记 ID 被选中并映射回表示模型生成的文本的文本中。

词汇量包含 50,257 个单词；因此，以下代码中的标记 ID 将从 0 到 50,256，而不是从 0 到 6。

此外，图 5.4 仅为了简化展示了单个文本示例（"每一点努力都值得"），在下文中，我们将通过实际代码示例实现图中的步骤，我们将使用两个 GPT 模型的输入示例（"每一点努力都值得"）

并且“我真的喜欢”）。

考虑这两个输入示例，它们已经被映射到标记 ID（图 5.4，步骤 1）：

```
输入      = torch.tensor([[16833,       [3626, 6100]       每个努力都推动
                          [40,        1107, 588]]]       #   "I  真的很   喜欢"]
```

匹配这些输入，目标包含模型需要生成的标记 ID：

```
targets = 目标orch.tensor([[3626,       6100, 345     ],    # 努力移动             您",
                          [1107，]588,  11311]])    #   真的很喜欢巧克力
```

请注意，目标输入是向前移动了一个位置的输入，这是我们在第 2 章中实现数据加载器时讨论的概念。这种移动策略对于教会模型预测序列中的下一个标记至关重要。

Now we feed the inputs into the model to calculate logits vectors for the two input examples, each comprising three tokens. Then we apply the `softmax` function to transform these logits into probability scores (`probas`; figure 5.4, step 2):

```
with torch.no_grad():
    logits = model(inputs)
probas = torch.softmax(logits, dim=-1)
print(probas.shape)
```

Disables gradient tracking since we are not training yet

Probability of each token in vocabulary

The resulting tensor dimension of the probability score (`probas`) tensor is

```
torch.Size([2, 3, 50257])
```

The first number, 2, corresponds to the two examples (rows) in the inputs, also known as batch size. The second number, 3, corresponds to the number of tokens in each input (row). Finally, the last number corresponds to the embedding dimensionality, which is determined by the vocabulary size. Following the conversion from logits to probabilities via the `softmax` function, the `generate_text_simple` function then converts the resulting probability scores back into text (figure 5.4, steps 3–5).

We can complete steps 3 and 4 by applying the `argmax` function to the probability scores to obtain the corresponding token IDs:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Given that we have two input batches, each containing three tokens, applying the `argmax` function to the probability scores (figure 5.4, step 3) yields two sets of outputs, each with three predicted token IDs:

```
Token IDs:
 tensor([[[16657],        <-- First batch
          [  339],
          [42826]],
         [[49906],        <-- Second batch
          [29669],
          [41751]]])
```

Finally, step 5 converts the token IDs back into text:

```
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1:"
      f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
```

When we decode these tokens, we find that these output tokens are quite different from the target tokens we want the model to generate:

```
Targets batch 1:  effort moves you
Outputs batch 1:  Armed heNetflix
```

现在我们将输入喂入模型以计算两个输入示例的 logits 向量，每个示例包含三个标记。然后我们应用 softmax 函数将这些 logits 转换为概率分数（概率；图 5.4，步骤 2）：

```
使用 torch.no_grad():
    logits = 模型(inputs) probas =
torch.softmax(logits, dim=-1) 打印
(probas.shape)
```

禁用梯度跟踪，因为我们尚未开始训练

每个的概率
词汇中的标记

结果概率分数（probas）张量的维度为

```
torch.Size([2, 3,])   50257])
```

第一个数字，2，对应于输入中的两个示例（行），也称为批量大小。第二个数字，3，对应于每个输入（行）中的标记数。最后，最后一个数字对应于嵌入维度，该维度由词汇表大小确定。在通过 softmax 函数将 logits 转换为概率之后，generate_text_simple 函数随后将得到的概率分数转换回文本（图 5.4，步骤 3-5）。

我们可以通过将概率分数应用于 argmax 函数来完成步骤 3 和 4，以获得相应的标记 ID：

```
    token_ids = torch.argmax(probas, dim=-1, keepdim=True) 打印
("Token IDs:\n", token_ids)
```

鉴于我们有两个输入批次，每个批次包含三个标记，将 argmax 函数应用于概率分数（图 5.4，步骤 3）得到两组输出，每组输出包含三个预测的标记 ID：

```
Token IDs:
 张量([[[16657],          <-- 第一批
        [ 339]
        [42826]]
       [49906]]          <-- 第二批次
        [29669]
        [41751]]])
```

最后，步骤 5 将令牌 ID 转换回文本：

```
打印目标批次 1：{token_ids_to_text(targets[0], tokenizer)} 打印输出批次 1：

      f" {token_ids_to_text(token_ids[0].展平(), 分词器)}"
```

当我们解码这些标记时，我们发现这些输出标记与我们希望模型生成的目标标记有很大不同：

```
目标批次 1：努力移动你 输出批次 1：武
装他 Netflix
```

The model produces random text that is different from the target text because it has not been trained yet. We now want to evaluate the performance of the model's generated text numerically via a loss (figure 5.5). Not only is this useful for measuring the quality of the generated text, but it's also a building block for implementing the training function, which we will use to update the model's weight to improve the generated text.

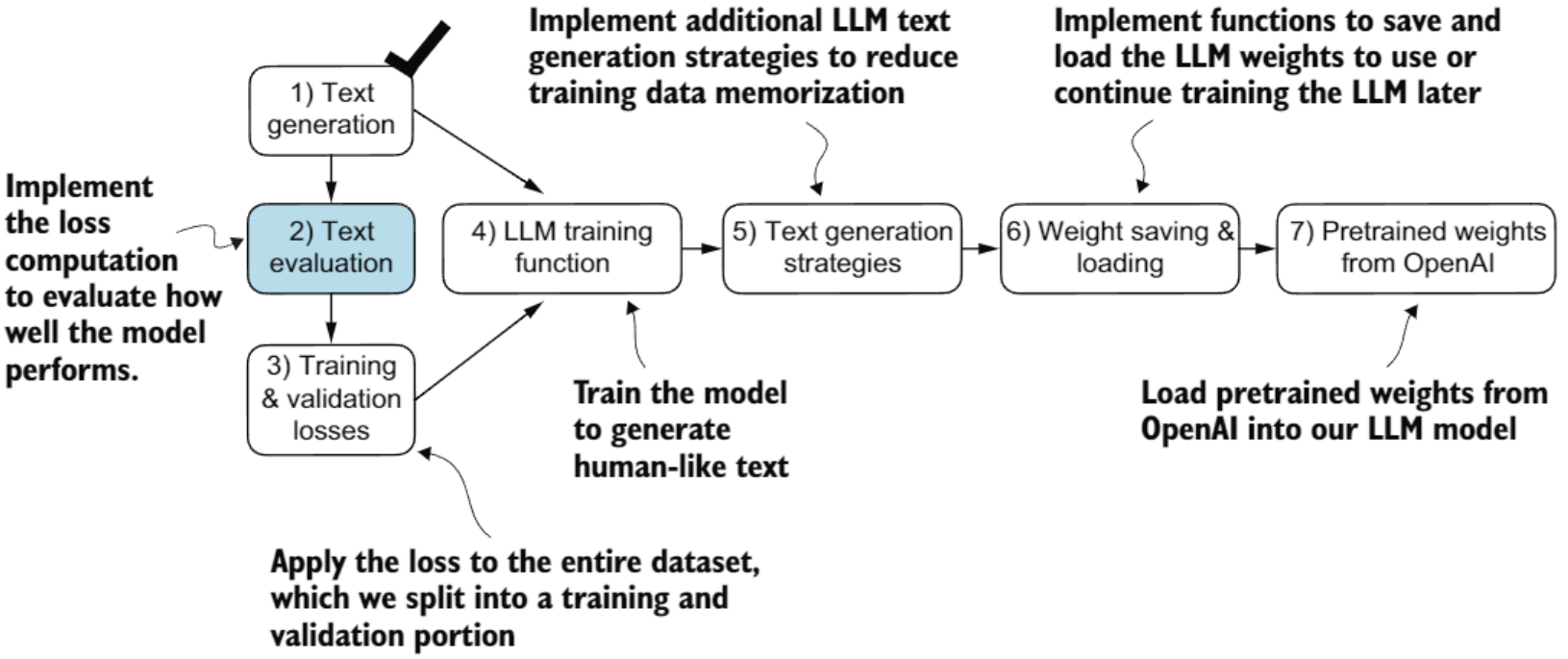

Figure 5.5 An overview of the topics covered in this chapter. We have completed step 1. We are now ready to implement the text evaluation function (step 2).

Part of the text evaluation process that we implement, as shown in figure 5.5, is to measure "how far" the generated tokens are from the correct predictions (targets). The training function we implement later will use this information to adjust the model weights to generate text that is more similar to (or, ideally, matches) the target text.

The model training aims to increase the softmax probability in the index positions corresponding to the correct target token IDs, as illustrated in figure 5.6. This softmax probability is also used in the evaluation metric we will implement next to numerically assess the model's generated outputs: the higher the probability in the correct positions, the better.

Remember that figure 5.6 displays the softmax probabilities for a compact seven-token vocabulary to fit everything into a single figure. This implies that the starting random values will hover around 1/7, which equals approximately 0.14. However, the vocabulary we are using for our GPT-2 model has 50,257 tokens, so most of the initial probabilities will hover around 0.00002 (1/50,257).

该模型生成的文本与目标文本不同，因为它尚未经过训练。我们现在想通过损失（图 5.5）对模型生成文本的性能进行数值评估。这不仅有助于衡量生成文本的质量，而且也是实现训练函数的基石，我们将使用它来更新模型的权重，以改进生成的文本。

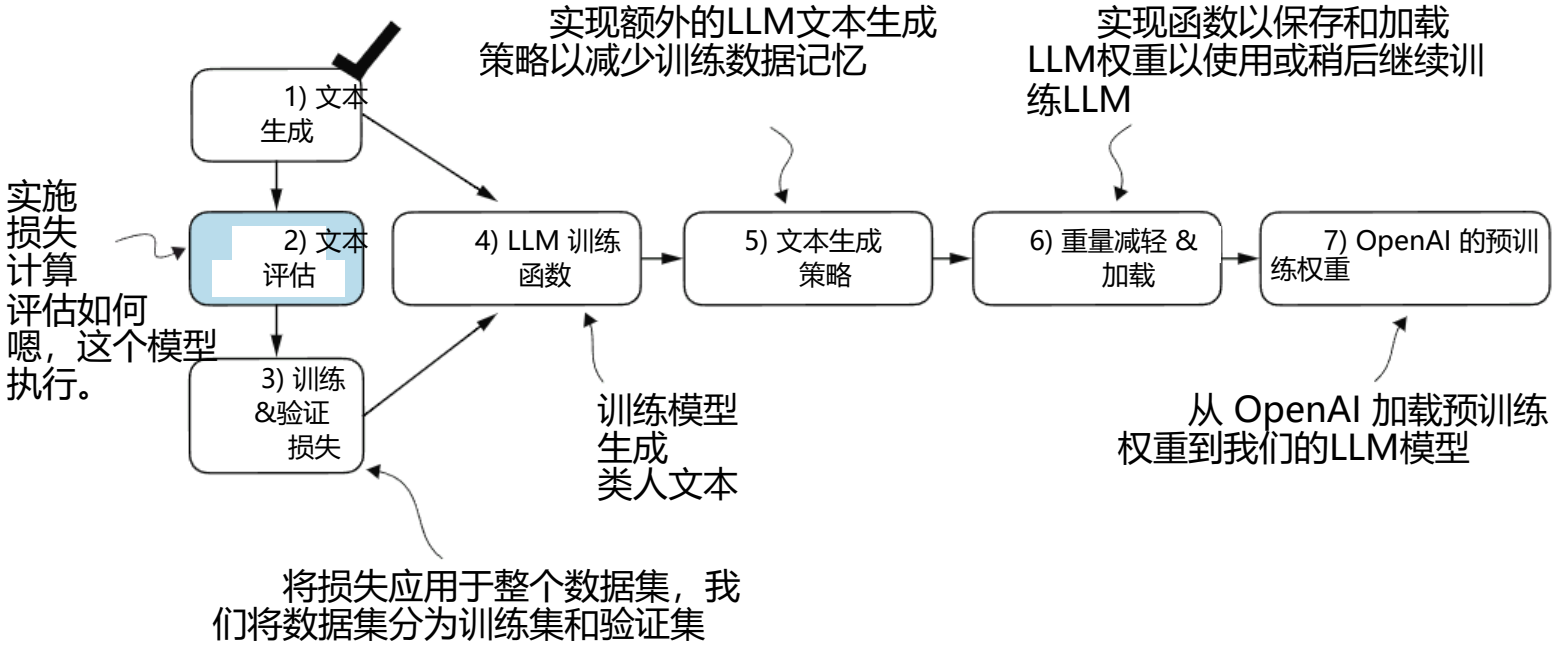

图 5.5 本章涵盖主题概述。我们已经完成第一步。现在我们准备实现文本评估函数（第二步）。

该文本评估过程的一部分，如图 5.5 所示，是衡量生成的标记与正确预测（目标）之间的“距离”。我们稍后实施的训练函数将使用此信息来调整模型权重，以生成更接近（或理想情况下匹配）目标文本的文本。

模型训练旨在增加对应正确目标标记 ID 的索引位置的 softmax 概率，如图 5.6 所示。此 softmax 概率也用于我们将在下一部分实现的评估指标中，以数值评估模型的生成输出：正确位置的概率越高，越好。

记住，图 5.6 显示了用于将所有内容放入单个图中的紧凑型七词词汇的 softmax 概率。这意味着起始随机值将围绕 1/7，即大约 0.14。然而，我们用于 GPT-2 模型的词汇表有 50,257 个标记，因此大多数初始概率将围绕 0.00002（1/50,257）。

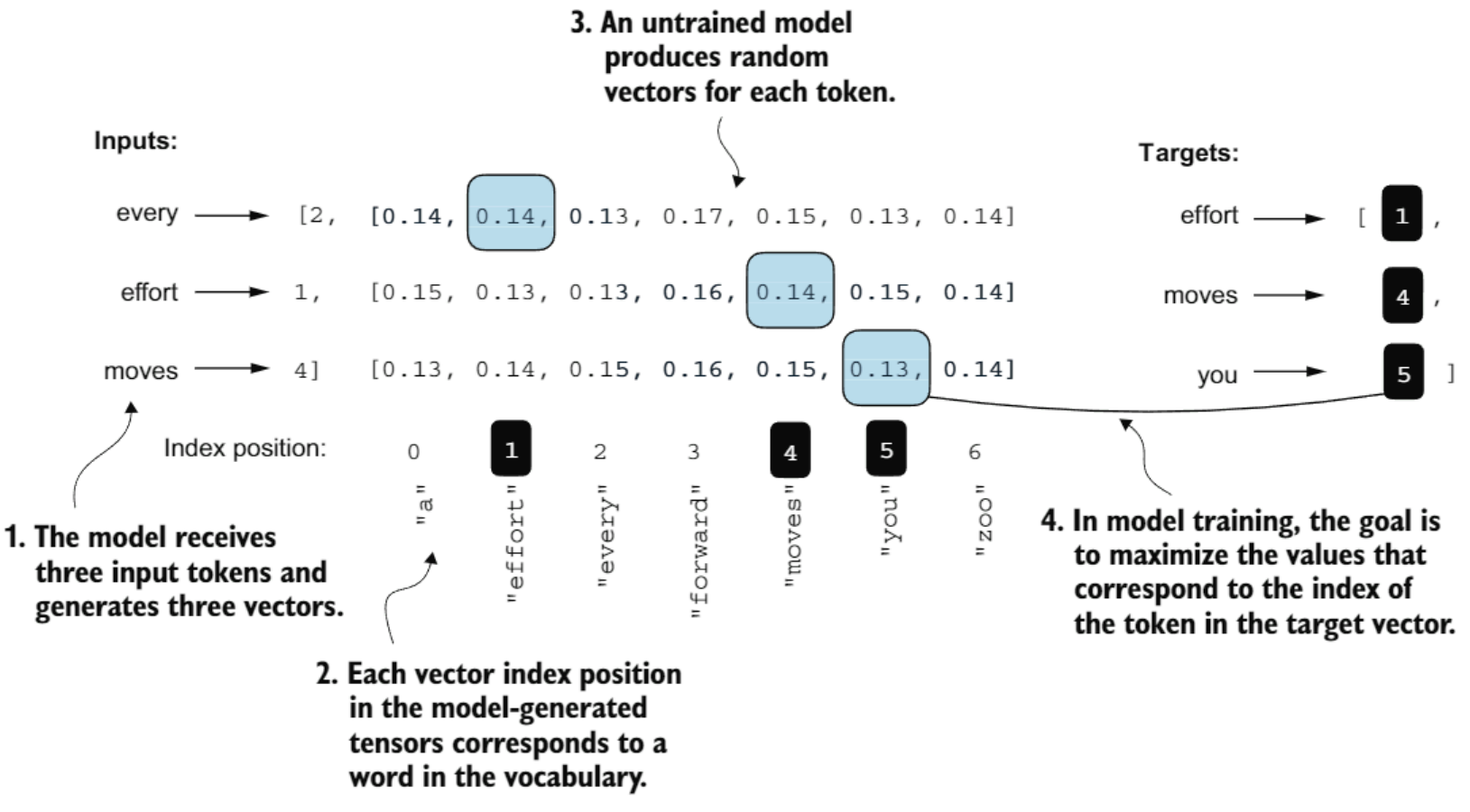

Figure 5.6 Before training, the model produces random next-token probability vectors. The goal of model training is to ensure that the probability values corresponding to the highlighted target token IDs are maximized.

For each of the two input texts, we can print the initial softmax probability scores corresponding to the target tokens using the following code:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
```

The three target token ID probabilities for each batch are

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

The goal of training an LLM is to maximize the likelihood of the correct token, which involves increasing its probability relative to other tokens. This way, we ensure the LLM consistently picks the target token—essentially the next word in the sentence—as the next token it generates.

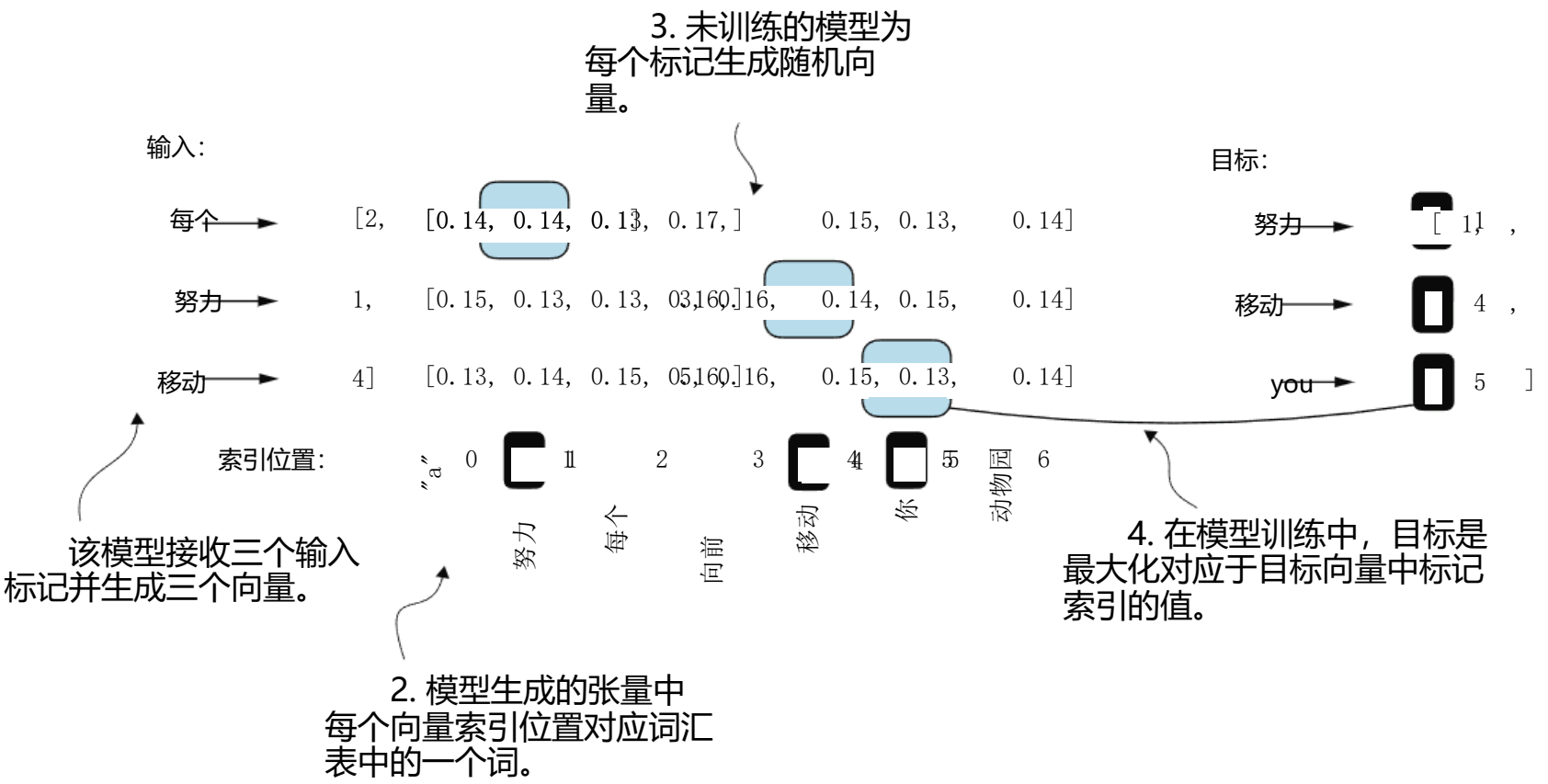

图 5.6 在训练前，模型生成随机的下一个标记概率向量。模型训练的目标是确保对应于高亮显示的目标标记 ID 的概率值最大化。

对于两个输入文本中的每一个，我们可以使用以下代码打印出对应目标标记的初始 softmax 概率得分：

```
    text_idx = 0
    target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]] 打印("文本 1:", target_probas_1)

    text_idx = 1
    target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]] 打印("文本 2:", target_probas_2)
```

每个批次的目标标记 ID 概率为三个

```
    文本 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05]) 文本 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

训练LLM的目标是最大化正确标记的可能性，这涉及到增加其相对于其他标记的概率。这样，我们确保LLM始终选择目标标记——即句子中的下一个单词——作为它生成的下一个标记。

Backpropagation

How do we maximize the softmax probability values corresponding to the target tokens? The big picture is that we update the model weights so that the model outputs higher values for the respective token IDs we want to generate. The weight update is done via a process called *backpropagation*, a standard technique for training deep neural networks (see sections A.3 to A.7 in appendix A for more details about backpropagation and model training).

Backpropagation requires a loss function, which calculates the difference between the model's predicted output (here, the probabilities corresponding to the target token IDs) and the actual desired output. This loss function measures how far off the model's predictions are from the target values.

Next, we will calculate the loss for the probability scores of the two example batches, `target_probas_1` and `target_probas_2`. The main steps are illustrated in figure 5.7. Since we already applied steps 1 to 3 to obtain `target_probas_1` and `target_probas_2`, we proceed with step 4, applying the *logarithm* to the probability scores:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
```

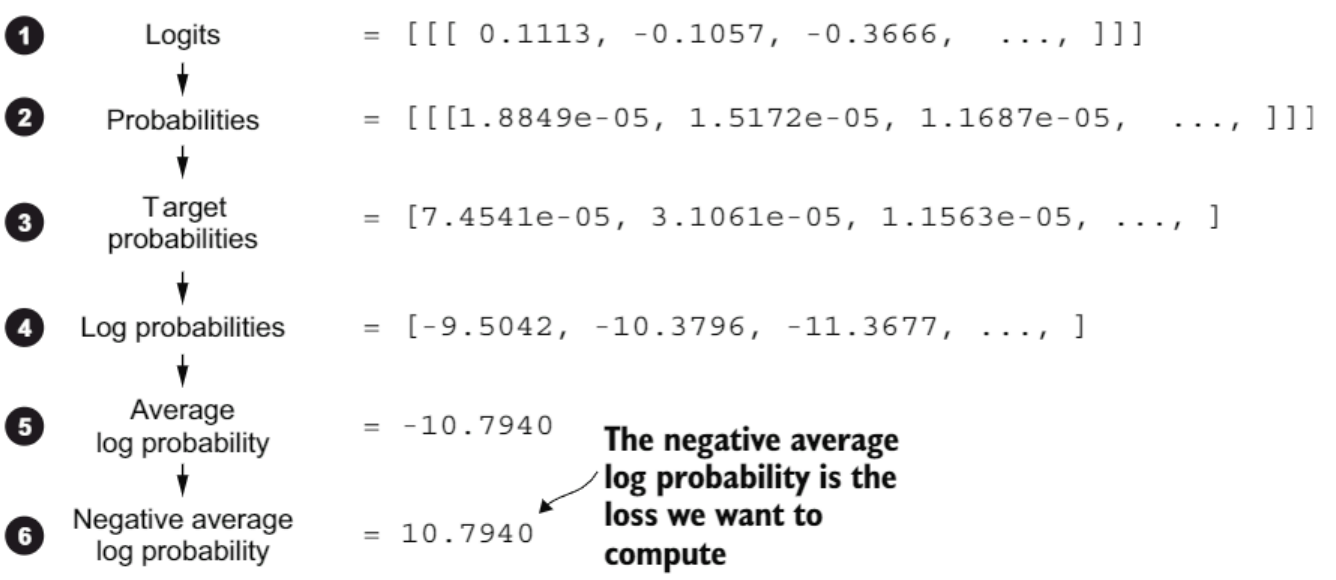

Figure 5.7 Calculating the loss involves several steps. Steps 1 to 3, which we have already completed, calculate the token probabilities corresponding to the target tensors. These probabilities are then transformed via a logarithm and averaged in steps 4 to 6.

This results in the following values:

```
tensor([ -9.5042, -10.3796, -11.3677, -11.4798,  -9.7764, -12.2561])
```

反向传播

如何最大化对应目标标记的 softmax 概率值？总体来说，我们更新模型权重，使得模型对想要生成的相应标记 ID 输出更高的值。权重更新是通过一个称为反向传播的过程完成的，这是训练深度神经网络的标准技术（有关反向传播和模型训练的更多详细信息，请参阅附录 A 中的 A.3 至 A.7 部分）。

反向传播需要一个损失函数，该函数计算模型预测输出（此处为对应目标标记 ID 的概率）与实际期望输出之间的差异。此损失函数衡量模型的预测与目标值之间的偏差程度。

接下来，我们将计算两个示例批次的目标概率得分 target_probas_1 和 target_probas_2 的损失。主要步骤如图 5.7 所示。由于我们已将步骤 1 至 3 应用于...

目标概率_1 and target_probas_2：概率_2

我们进行第 4 步，将对概率得分应用对数：

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2))) 打印(log_probas)
```

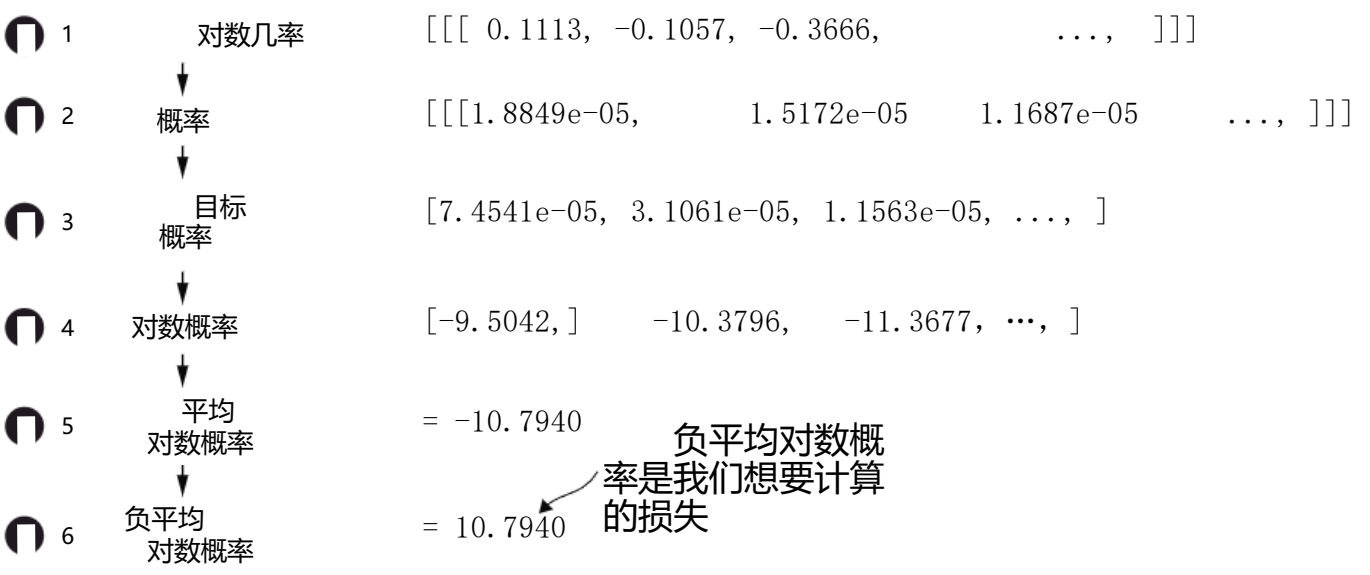

图 5.7 计算损失涉及多个步骤。步骤 1 到 3，我们已经完成，计算与目标张量对应的标记概率。然后，这些概率通过对数变换，并在步骤 4 到 6 中平均。

这导致以下值：

张量([-9.5042, -10.3796,]) -11.3677, -11.4798, -9.7764, -12.2561]]

Working with logarithms of probability scores is more manageable in mathematical optimization than handling the scores directly. This topic is outside the scope of this book, but I've detailed it further in a lecture, which can be found in appendix B.

Next, we combine these log probabilities into a single score by computing the average (step 5 in figure 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

The resulting average log probability score is

```
tensor(-10.7940)
```

The goal is to get the average log probability as close to 0 as possible by updating the model's weights as part of the training process. However, in deep learning, the common practice isn't to push the average log probability up to 0 but rather to bring the negative average log probability down to 0. The negative average log probability is simply the average log probability multiplied by –1, which corresponds to step 6 in figure 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

This prints `tensor(10.7940)`. In deep learning, the term for turning this negative value, –10.7940, into 10.7940, is known as the *cross entropy* loss. PyTorch comes in handy here, as it already has a built-in `cross_entropy` function that takes care of all these six steps in figure 5.7 for us.

Cross entropy loss

At its core, the cross entropy loss is a popular measure in machine learning and deep learning that measures the difference between two probability distributions—typically, the true distribution of labels (here, tokens in a dataset) and the predicted distribution from a model (for instance, the token probabilities generated by an LLM).

In the context of machine learning and specifically in frameworks like PyTorch, the `cross_entropy` function computes this measure for discrete outcomes, which is similar to the negative average log probability of the target tokens given the model's generated token probabilities, making the terms "cross entropy" and "negative average log probability" related and often used interchangeably in practice.

Before we apply the `cross_entropy` function, let's briefly recall the shape of the logits and target tensors:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

与概率得分的对数相比，在数学优化中处理得分更为方便。这个主题超出了本书的范围，但我已在讲座中进一步详细阐述，可在附录 B 中找到。

接下来，我们将这些对数概率合并成一个单一得分，通过计算平均值（图 5.7 中的步骤 5）：

```
    avg_log_probas = torch.mean(log_probas) 打印
(avg_log_probas)
```

结果平均对数概率分数是

```
张量(-10.7940)
```

目标是通过在训练过程中更新模型的权重，将平均对数概率尽可能接近 0。然而，在深度学习中，常见的做法并不是将平均对数概率推到 0，而是将负平均对数概率降低到 0。负平均对数概率就是对数概率的平均值乘以-1，这对应于图 5.7 的第 6 步。

```
    neg_avg_log_probas = avg_log_probas * -1 打印
(neg_avg_log_probas)
```

这会打印出 tensor(10.7940)。在深度学习中，将这个负值-10.7940 转换为 10.7940 的术语称为交叉熵损失。PyTorch 在这里很有用，因为它已经内置了一个交叉熵函数，可以为我们处理图 5.7 中的所有六个步骤。

交叉熵损失

核心上，交叉熵损失是机器学习和深度学习中一种流行的度量，用于衡量两个概率分布之间的差异——通常是标签的真实分布（在此，数据集中的标记）和模型预测的分布（例如，由LLM生成的标记概率）。

在机器学习领域，特别是在 PyTorch 等框架中，cross_entropy 函数计算离散结果的这个度量，这与给定模型生成的标记概率的目标标记的负平均对数概率相似，使得“交叉熵”和“负平均对数概率”这两个术语相关，并且在实践中经常可以互换使用。

在我们应用交叉熵函数之前，让我们简要回顾一下 logits 和目标张量的形状：

```
    打印("Logits 形状：", logits.shape) 打印
("Targets 形状：", targets.shape)
```

The resulting shapes are

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

As we can see, the `logits` tensor has three dimensions: batch size, number of tokens, and vocabulary size. The `targets` tensor has two dimensions: batch size and number of tokens.

For the `cross_entropy` loss function in PyTorch, we want to flatten these tensors by combining them over the batch dimension:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

The resulting tensor dimensions are

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Remember that the `targets` are the token IDs we want the LLM to generate, and the `logits` contain the unscaled model outputs before they enter the `softmax` function to obtain the probability scores.

Previously, we applied the `softmax` function, selected the probability scores corresponding to the target IDs, and computed the negative average log probabilities. PyTorch's `cross_entropy` function will take care of all these steps for us:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

The resulting loss is the same that we obtained previously when applying the individual steps in figure 5.7 manually:

```
tensor(10.7940)
```

Perplexity

Perplexity is a measure often used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling. It can provide a more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.

Perplexity measures how well the probability distribution predicted by the model matches the actual distribution of the words in the dataset. Similar to the loss, a lower perplexity indicates that the model predictions are closer to the actual distribution.

结果形状是

```
   Logits 形状：torch.Size([2, 3, 50257]) 形状)
目标形状：torch.Size([2, 3])
```

如您所见，logits 张量有三个维度：批次大小、标记数和词汇大小。targets 张量有两个维度：批次大小和标记数。

对于 PyTorch 中的交叉熵损失函数，我们希望通过在批次维度上组合它们来展平这些张量：

```
   logits_flat = logits.flatten(0, 1) targets_flat =
targets.flatten() 打印("展平的 logits 的形状：",
logits_flat.shape) 打印("展平的 targets 的形状：")
                                        targets_flat.shape)
```

结果张量的维度是

```
   展平的 logits：torch.Size([6, 50257]) 展平的
目标：torch.Size([6])
```

记住，目标是我们要让LLM生成的标记 ID，以及
logits 包含在它们进入 softmax 函数以获得概率分数之前未缩放的模型输出。
之前，我们应用了 softmax 函数，选择了对应目标 ID 的概率分数，并计算了负平均对数概率。PyTorch 的 cross_entropy 函数将为我们处理所有这些步骤：

```
   损失 = torch.nn.functional.cross_entropy(logits_flat, targets_flat) 打印(损
失)
```

结果损失与我们在手动应用图 5.7 中的各个步骤时获得的结果相同：

```
张量(10.7940)
```

困惑度

困惑度是常与交叉熵损失一起使用来评估模型在语言建模等任务中性能的度量。它可以提供一种更可解释的方式来理解模型在预测序列中下一个标记时的不确定性。

困惑度衡量模型预测的概率分布与数据集中单词的实际分布匹配的程度。类似于损失，较低的困惑度表示模型预测更接近实际分布。

(continued)

Perplexity can be calculated as `perplexity = torch.exp(loss)`, which returns `tensor(48725.8203)` when applied to the previously calculated loss.

Perplexity is often considered more interpretable than the raw loss value because it signifies the effective vocabulary size about which the model is uncertain at each step. In the given example, this would translate to the model being unsure about which among 48,725 tokens in the vocabulary to generate as the next token.

We have now calculated the loss for two small text inputs for illustration purposes. Next, we will apply the loss computation to the entire training and validation sets.

5.1.3 Calculating the training and validation set losses

We must first prepare the training and validation datasets that we will use to train the LLM. Then, as highlighted in figure 5.8, we will calculate the cross entropy for the training and validation sets, which is an important component of the model training process.

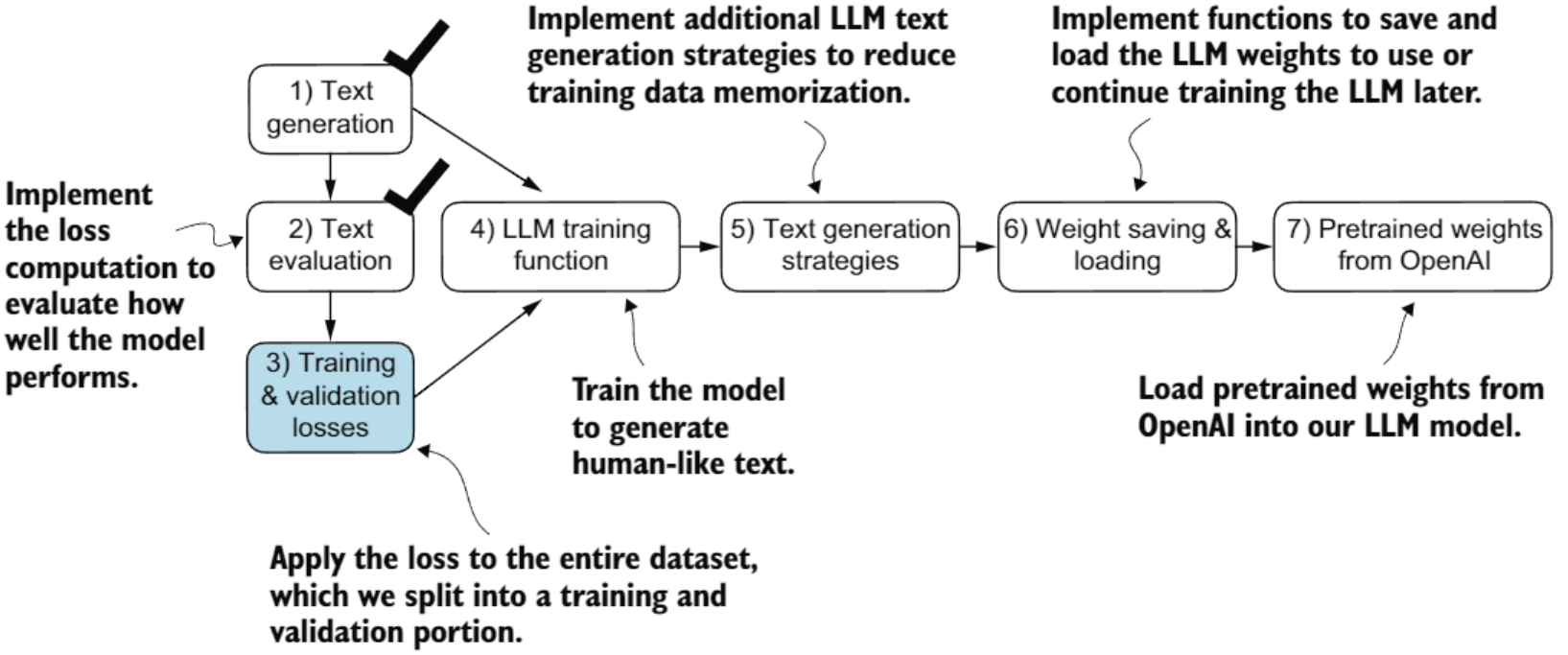

Figure 5.8 Having completed steps 1 and 2, including computing the cross entropy loss, we can now apply this loss computation to the entire text dataset that we will use for model training.

To compute the loss on the training and validation datasets, we use a very small text dataset, the "The Verdict" short story by Edith Wharton, which we have already worked with in chapter 2. By selecting a text from the public domain, we circumvent any concerns related to usage rights. Additionally, using such a small dataset allows for the execution of code examples on a standard laptop computer in a matter of

(继续)

困惑度可以计算为 perplexity = torch.exp(loss)，当应用于之前计算的损失时，返回 tensor(48725.8203)。

困惑度通常被认为比原始损失值更可解释，因为它表示模型在每一步对有效词汇量的不确定性。在给定示例中，这意味着模型不确定在词汇表中的 48,725 个标记中应该生成哪个作为下一个标记。

我们现在已计算了两个小型文本输入的损失，以供说明之用。

接下来，我们将对整个训练集和验证集应用损失计算。

5.1.3 计算训练集和验证集的损失

我们必须首先准备我们将用于训练LLM的训练和验证数据集。然后，如图 5.8 所示，我们将计算训练集和验证集的交叉熵，这是模型训练过程中的一个重要组成部分。

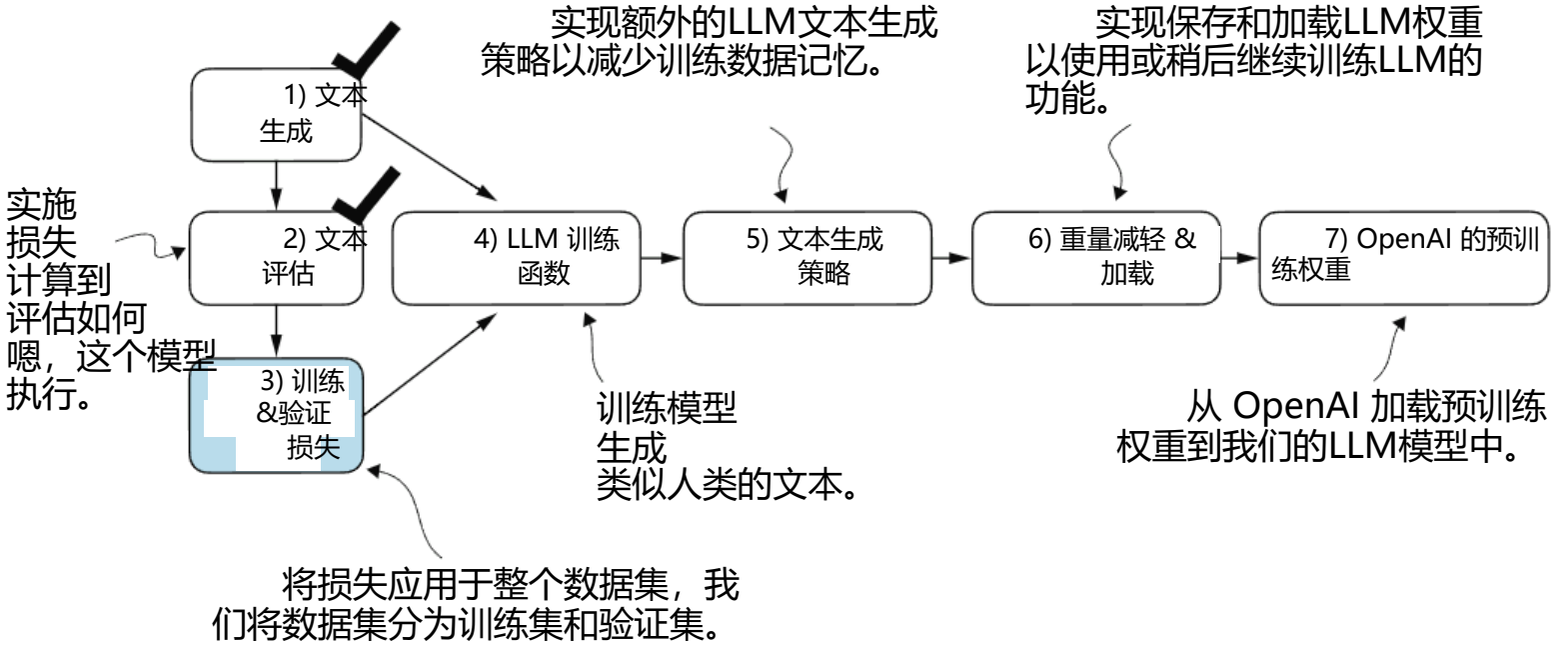

图 5.8 完成步骤 1 和 2，包括计算交叉熵损失后，我们现在可以将这个损失计算应用于我们将用于模型训练的整个文本数据集。

为了计算训练集和验证集上的损失，我们使用了一个非常小的文本数据集，即爱迪丝·华顿的短篇小说《判决》，我们在第 2 章中已经使用过它。通过选择公共领域的文本，我们规避了任何与使用权相关的问题。此外，使用如此小的数据集允许在标准笔记本电脑上执行代码示例，只需

minutes, even without a high-end GPU, which is particularly advantageous for educational purposes.

> **NOTE** Interested readers can also use the supplementary code for this book to prepare a larger-scale dataset consisting of more than 60,000 public domain books from Project Gutenberg and train an LLM on these (see appendix D for details).

The cost of pretraining LLMs

To put the scale of our project into perspective, consider the training of the 7 billion parameter Llama 2 model, a relatively popular openly available LLM. This model required 184,320 GPU hours on expensive A100 GPUs, processing 2 trillion tokens. At the time of writing, running an 8 × A100 cloud server on AWS costs around $30 per hour. A rough estimate puts the total training cost of such an LLM at around $690,000 (calculated as 184,320 hours divided by 8, then multiplied by $30).

The following code loads the "The Verdict" short story:

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

After loading the dataset, we can check the number of characters and tokens in the dataset:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

The output is

```
Characters: 20479
Tokens: 5145
```

With just 5,145 tokens, the text might seem too small to train an LLM, but as mentioned earlier, it's for educational purposes so that we can run the code in minutes instead of weeks. Plus, later we will load pretrained weights from OpenAI into our `GPTModel` code.

Next, we divide the dataset into a training and a validation set and use the data loaders from chapter 2 to prepare the batches for LLM training. This process is visualized in figure 5.9. Due to spatial constraints, we use a `max_length=6`. However, for the actual data loaders, we set the `max_length` equal to the 256-token context length that the LLM supports so that the LLM sees longer texts during training.

使用如此小的数据集可以在几分钟内在一个标准笔记本电脑上执行代码示例，即使没有高端 GPU，这对于教育目的尤其有利。

注意：感兴趣的读者也可以使用本书的补充代码来准备一个包含超过 60,000 本公共领域书籍的大规模数据集，并在这些数据上训练一个LLM（详情见附录 D）。

预训练成本 LLMs

将我们的项目规模置于适当的角度，考虑训练 700 亿参数的 Llama 2 模型，这是一个相对流行的公开可用的LLM。该模型在昂贵的 A100 GPU 上需要 184,320 个 GPU 小时，处理了 2000 亿个 token。在撰写本文时，在 AWS 上运行 8 × A100 云服务器每小时大约花费 30 美元。粗略估计，这种LLM的总训练成本约为 69 万美元（计算方法为 184,320 小时除以 8，然后乘以 30 美元）。

以下代码加载了“The Verdict”短篇小说：

```
file_path = "the-verdict.txt" with open(file_path, "r",
encoding="utf-8") as file:
```

请提供需要翻译的文本数据

在加载数据集后，我们可以检查数据集中的字符和标记数量：

```
总字符数 = len(text_data) 总标记数 =
len(tokenizer.encode(text_data)) 打印("字符数：",
total_characters) 打印("标记数：", total_tokens)
```

输出结果

```
字符：20479
代币：5145
```

仅用 5,145 个标记，文本可能看起来太小，不足以训练LLM，但如前所述，这是出于教育目的，以便我们能在几分钟内而不是几周内运行代码。此外，稍后我们将从 OpenAI 加载预训练权重到我们的

GPT 模型代码。

接下来，我们将数据集分为训练集和验证集，并使用第 2 章中的数据加载器为LLM训练准备批次。此过程在图 5.9 中进行了可视化。由于空间限制，我们使用 max_length=6。然而，对于实际的数据加载器，我们将 max_length 设置为LLM支持的 256 个 token 的上下文长度，以便LLM在训练期间看到更长的文本。

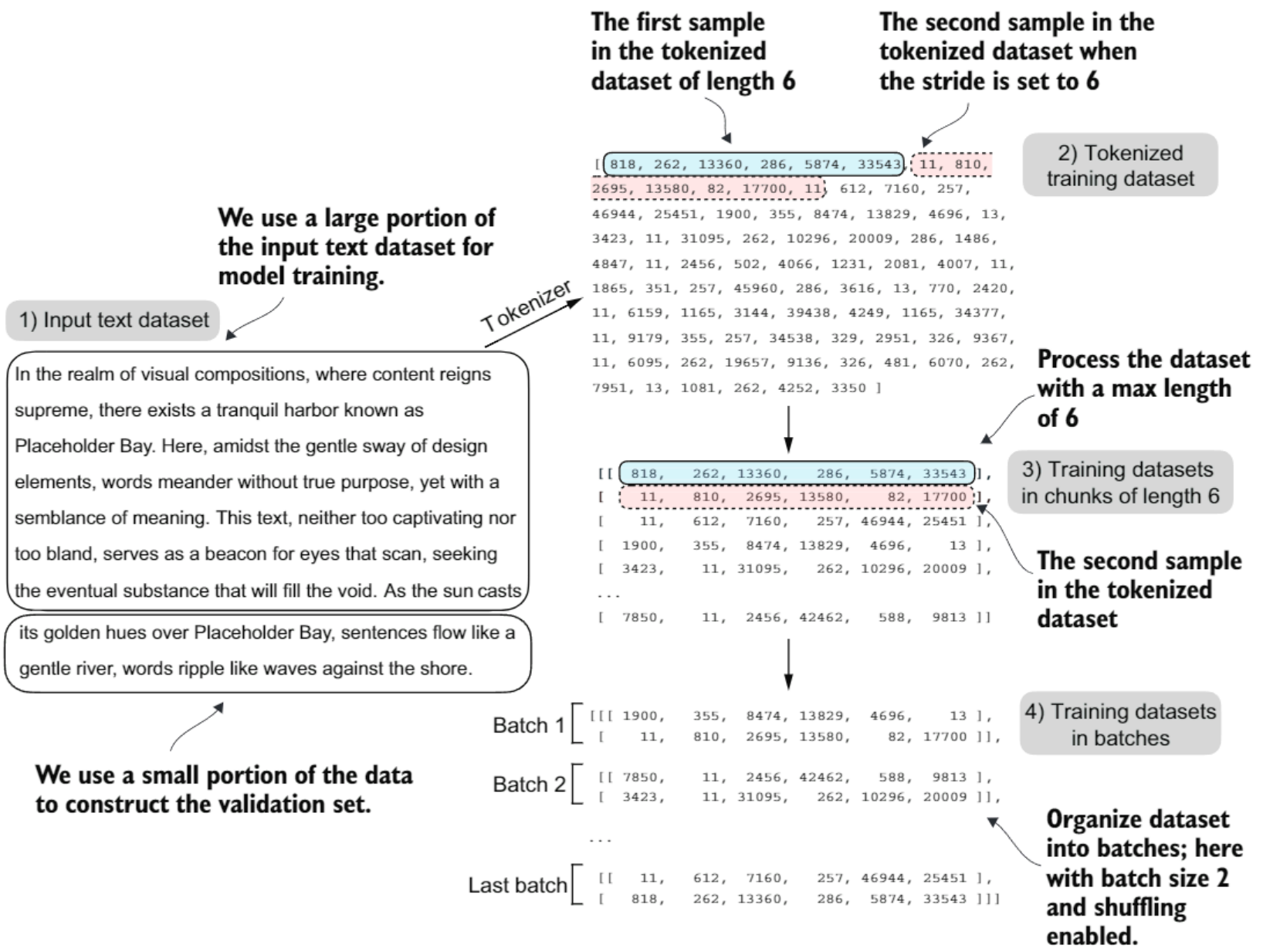

Figure 5.9 When preparing the data loaders, we split the input text into training and validation set portions. Then we tokenize the text (only shown for the training set portion for simplicity) and divide the tokenized text into chunks of a user-specified length (here, 6). Finally, we shuffle the rows and organize the chunked text into batches (here, batch size 2), which we can use for model training.

> **NOTE** We are training the model with training data presented in similarly sized chunks for simplicity and efficiency. However, in practice, it can also be beneficial to train an LLM with variable-length inputs to help the LLM to better generalize across different types of inputs when it is being used.

To implement the data splitting and loading, we first define a `train_ratio` to use 90% of the data for training and the remaining 10% as validation data for model evaluation during training:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

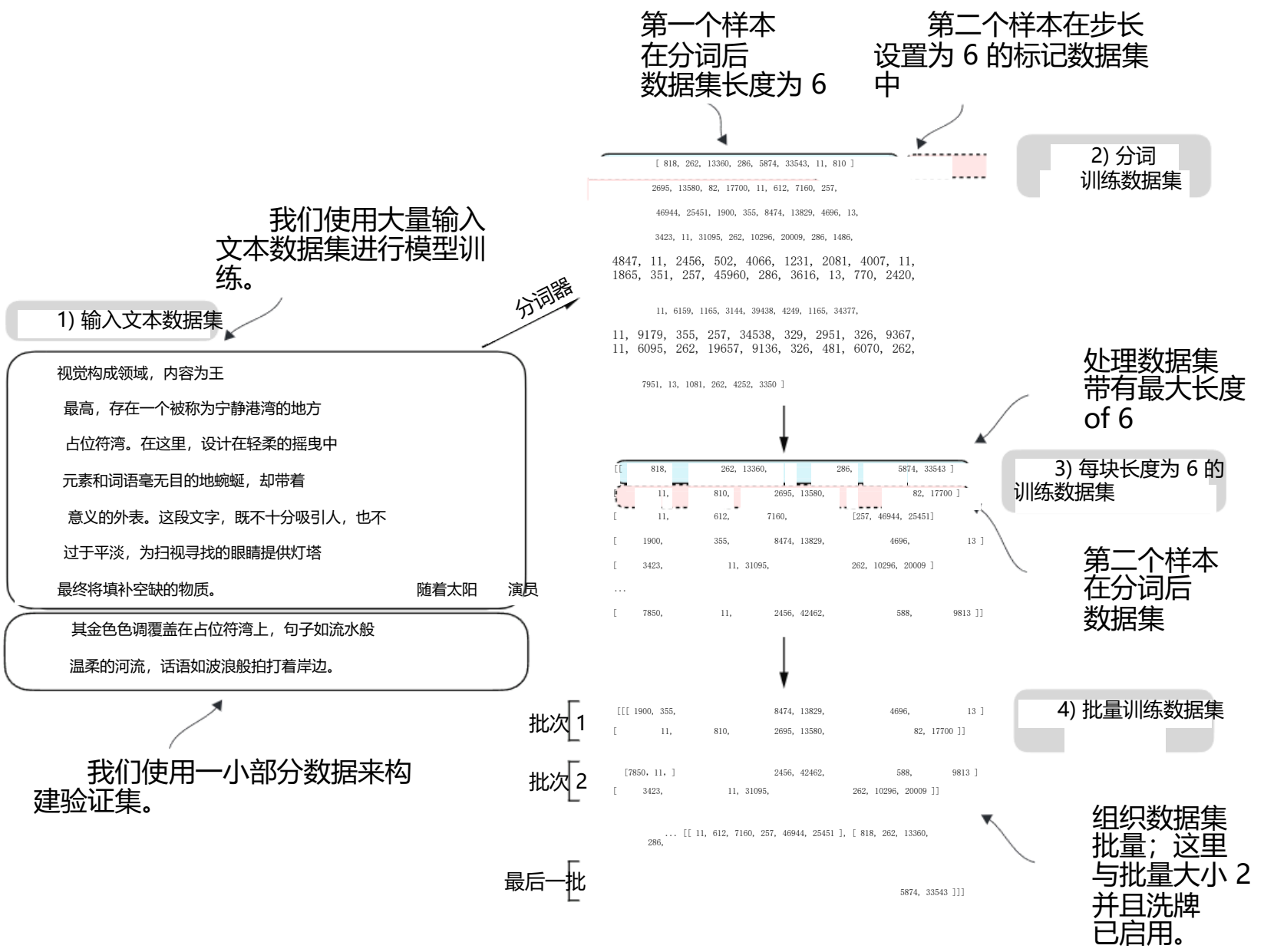

图 5.9 在准备数据加载器时，我们将输入文本分为训练集和验证集部分。然后我们对文本进行分词（为了简单起见，这里只展示了训练集部分）并将分词后的文本划分为用户指定的长度块（这里为 6）。最后，我们打乱行顺序并将分块后的文本组织成批次（这里批次大小为 2），这些批次可以用于模型训练。

注意：我们以相似大小的块来训练模型，以简化并提高效率。然而，在实践中，训练一个具有可变长度输入的LLM也有利于LLM在使用时更好地泛化到不同类型的输入。

为了实现数据拆分和加载，我们首先定义一个 train_ratio，使用 90%的数据进行训练，剩余的 10%作为验证数据，用于在训练过程中评估模型：

```
    train_ratio = 0.90 split_idx = int(train_ratio *
len(text_data)) train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

Using the `train_data` and `val_data` subsets, we can now create the respective data loader reusing the `create_dataloader_v1` code from chapter 2:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

We used a relatively small batch size to reduce the computational resource demand because we were working with a very small dataset. In practice, training LLMs with batch sizes of 1,024 or larger is not uncommon.

As an optional check, we can iterate through the data loaders to ensure that they were created correctly:

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

We should see the following outputs:

```
Train loader:
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
```

使用 train_data 和 val_data 子集，我们现在可以创建相应的数据加载器，重用第 2 章中的 create_dataloader_v1 代码：

```
    从第 02 章导入 create_dataloader_v1, torch 手动
设置随机种子为 123


    train_loader = create_dataloader_v1(
        训练数据
        batch_size=2, 最大长度
     =GPT_CONFIG_124M["context_length"], 步长
     =GPT_CONFIG_124M["context_length"], drop_last=True,
     shuffle=True, num_workers=0



    val_loader = create_dataloader_v1()


        val_data,
    batch_size=2, 最大长度=GPT_CONFIG_124M["上下文长
度"], 步长=GPT_CONFIG_124M["上下文长度"],
drop_last=False, shuffle=False, num_workers=0 )
```

我们使用了相对较小的批量大小以减少计算资源需求，因为我们正在处理一个非常小的数据集。在实践中，使用 1,024 或更大的批量大小来训练LLMs并不罕见。

作为一个可选的检查，我们可以遍历数据加载器以确保它们被正确创建：

```
    打印("训练加载器：") for x,
y in train_loader:
        打印 x 的形状      y 的形状

    打印("\n 验证加载器：") for x, y
in val_loader:
        打印 x 的形状      y 的形状
```

我们应该看到以下输出：

```
    训练加载器：torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256])
```

```
Validation loader:
torch.Size([2, 256]) torch.Size([2, 256])
```

Based on the preceding code output, we have nine training set batches with two samples and 256 tokens each. Since we allocated only 10% of the data for validation, there is only one validation batch consisting of two input examples. As expected, the input data (`x`) and target data (`y`) have the same shape (the batch size times the number of tokens in each batch) since the targets are the inputs shifted by one position, as discussed in chapter 2.

Next, we implement a utility function to calculate the cross entropy loss of a given batch returned via the training and validation loader:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

The transfer to a given device allows us to transfer the data to a GPU.

We can now use this `calc_loss_batch` utility function, which computes the loss for a single batch, to implement the following `calc_loss_loader` function that computes the loss over all the batches sampled by a given data loader.

Listing 5.2 Function to compute the training and validation loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Iteratives over all batches if no fixed num_batches is specified

Reduces the number of batches to match the total number of batches in the data loader if num_batches exceeds the number of batches in the data loader

Sums loss for each batch

Averages the loss over all batches

By default, the `calc_loss_loader` function iterates over all batches in a given data loader, accumulates the loss in the `total_loss` variable, and then computes and

256]) 验证加载器：
torch.Size([2, 256]) torch.Size([2, 256])

基于前面的代码输出，我们有九个训练集批次，每个批次包含两个样本和 256 个标记。由于我们只分配了 10%的数据用于验证，因此只有一个包含两个输入示例的验证批次。正如预期的那样，输入数据（x）和目标数据（y）具有相同的形状（批次大小乘以每个批次的标记数），因为目标数据是输入数据向右移动一个位置，正如第 2 章所讨论的。

接下来，我们实现一个实用函数来计算通过训练和验证加载器返回的给定批次的交叉熵损失：

```
def calc_loss_batch(输入批次, 目标批次, 模型, 设备):
    input_batch = 输入批次转换为设备
 target_batch = 目标批次转换为设备
 logits = 模型输入输入批次
 loss = torch.nn.functional.cross_entropy(

    logits 展平(0, 1), target_batch 展平() 返回 loss
```

将转换到
给定设备允许
我们转移到
数据传输到 GPU。

我们现在可以使用这个 calc_loss_batch 实用函数，它计算单个批次的损失，来实现以下 calc_loss_loader 函数，该函数计算由给定数据加载器采样的所有批次的损失。

列表 5.2 函数用于计算训练和验证损失

```
def calc_loss_loader(data_loader, model, device, num_batches=None): 总损失 =
0.
    如果 len(data_loader) == 0:
        返回浮点数"nan"
    elif num_batches 是 None:
    num_batches = len(data_loader) 否
 则：
    num_batches = min(num_batches, 数据加载器长度) for i, (输入批次, 目标
 批次) in enumerate(数据加载器):
        如果 i < num_batches:
            损失 = calc_loss_batch(
        输入批次，目标批次，模型，设备）总损失 += 损
     失.item() 否则：
            break
    返回 中断总损失 / 批次数量
```

遍历所有
批次，如果没有指
定固定的 num_batches

减少数量
批次匹配
总数量
数据加载器中的
批次，如果
num_batches 超过
批次中的
数据加载器

求和损失
对于每个
批量

平均所有批次的损失

默认情况下，calc_loss_loader 函数遍历给定数据加载器中的所有批次，将损失累加到 total_loss 变量中，然后计算并

averages the loss over the total number of batches. Alternatively, we can specify a smaller number of batches via `num_batches` to speed up the evaluation during model training.

Let's now see this `calc_loss_loader` function in action, applying it to the training and validation set loaders:

If you have a machine with a CUDA-supported GPU, the LLM will train on the GPU without making any changes to the code.

Disables gradient tracking for efficiency because we are not training yet

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

Via the "device" setting, we ensure the data is loaded onto the same device as the LLM model.

The resulting loss values are

```
Training loss: 10.98758347829183
Validation loss: 10.98110580444336
```

The loss values are relatively high because the model has not yet been trained. For comparison, the loss approaches 0 if the model learns to generate the next tokens as they appear in the training and validation sets.

Now that we have a way to measure the quality of the generated text, we will train the LLM to reduce this loss so that it becomes better at generating text, as illustrated in figure 5.10.

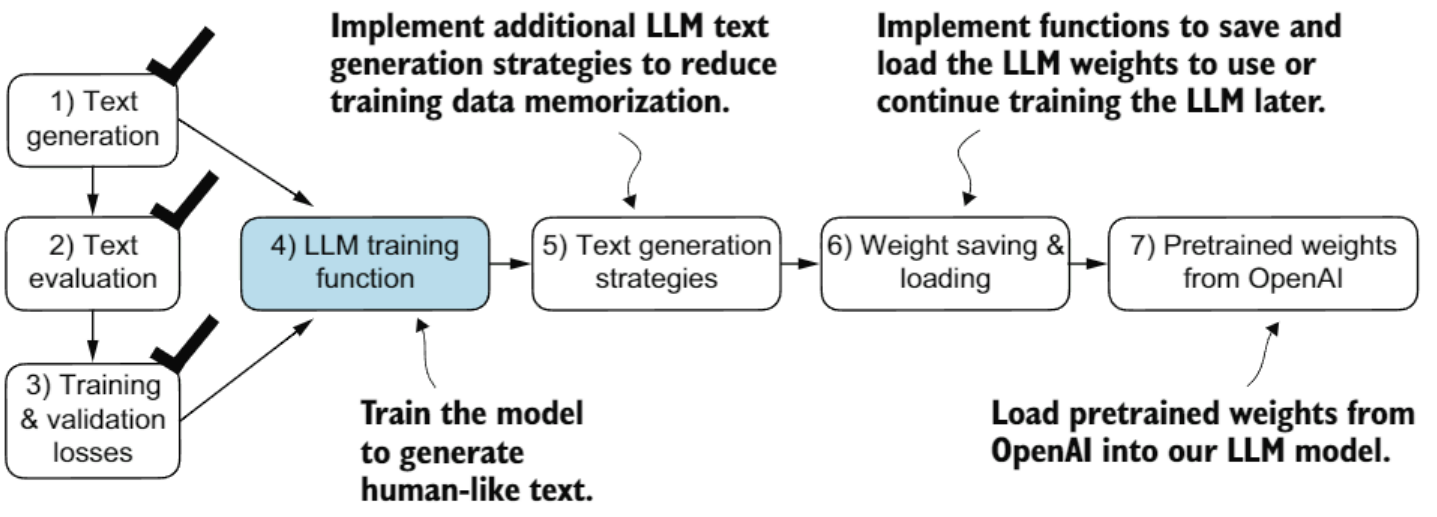

Figure 5.10 We have recapped the text generation process (step 1) and implemented basic model evaluation techniques (step 2) to compute the training and validation set losses (step 3). Next, we will go to the training functions and pretrain the LLM (step 4).

平均总批次数上的损失。或者，我们可以通过 num_batches 指定更少的批次数量，以加快模型训练期间的评估速度。

现在让我们看看这个 calc_loss_loader 函数的实际应用，将其应用于训练集和验证集加载器：

如果您有一台支持 CUDA 的 GPU 的机器，LLM将在 GPU 上训练而无需对代码进行任何修改。

禁用梯度跟踪以提高效率，因为我们尚未开始训练

```
设备 = torch.device("cuda" if torch.cuda.is_available() else "cpu") 模型.to(设备) with torch.no_grad():

训练损失：calc_loss_loader(train_loader, model, device) 验证损失：calc_loss_loader(val_loader, model, device) 打印("训练损失：", train_loss) 打印("验证损失：", val_loss)
```

通过“设备”设置，我们确保数据加载到与LLM模型相同的设备上。

结果损失值

```
训练损失：10.98758347829183 验证损失：
                                10.98110580444336
```

损失值相对较高，因为模型尚未经过训练。为了比较，如果模型学会生成与训练和验证集中出现的下一个标记相同的标记，则损失趋近于 0。

现在我们有了衡量生成文本质量的方法，我们将训练LLM以减少这种损失，使其在生成文本方面变得更好，如图 5.10 所示。

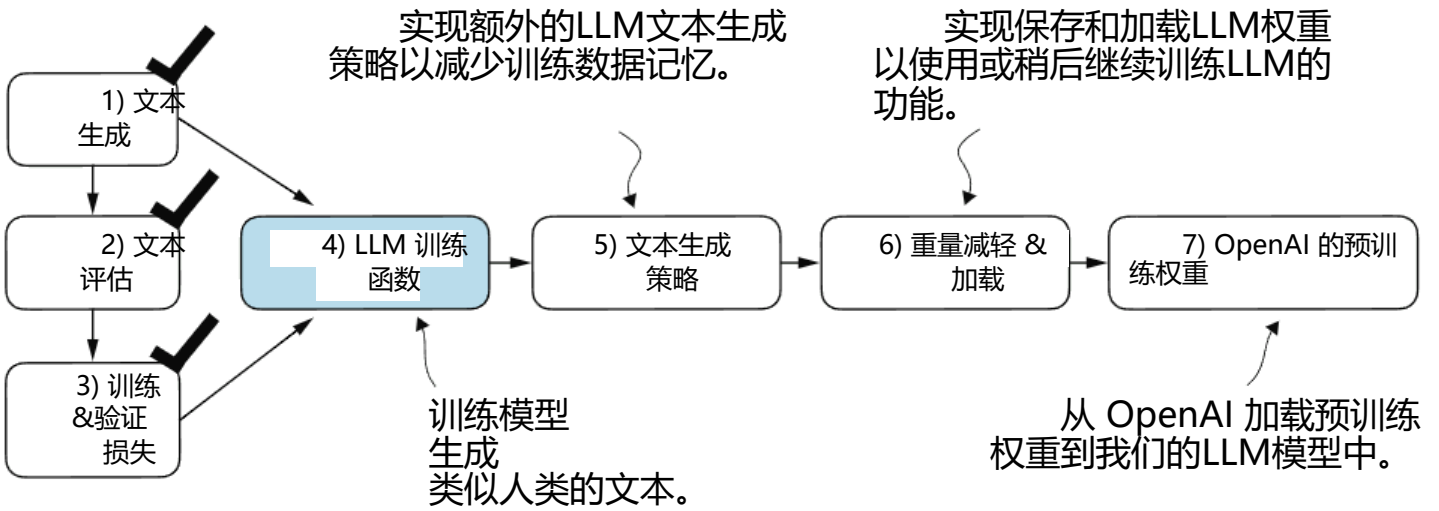

图 5.10 我们概述了文本生成过程（步骤 1）并实现了基本模型评估技术（步骤 2）以计算训练集和验证集损失（步骤 3）。接下来，我们将转向训练函数并预训练LLM（步骤 4）。

Next, we will focus on pretraining the LLM. After model training, we will implement alternative text generation strategies and save and load pretrained model weights.

5.2 Training an LLM

It is finally time to implement the code for pretraining the LLM, our `GPTModel`. For this, we focus on a straightforward training loop to keep the code concise and readable.

> **NOTE** Interested readers can learn about more advanced techniques, including *learning rate warmup*, *cosine annealing*, and *gradient clipping*, in appendix D.

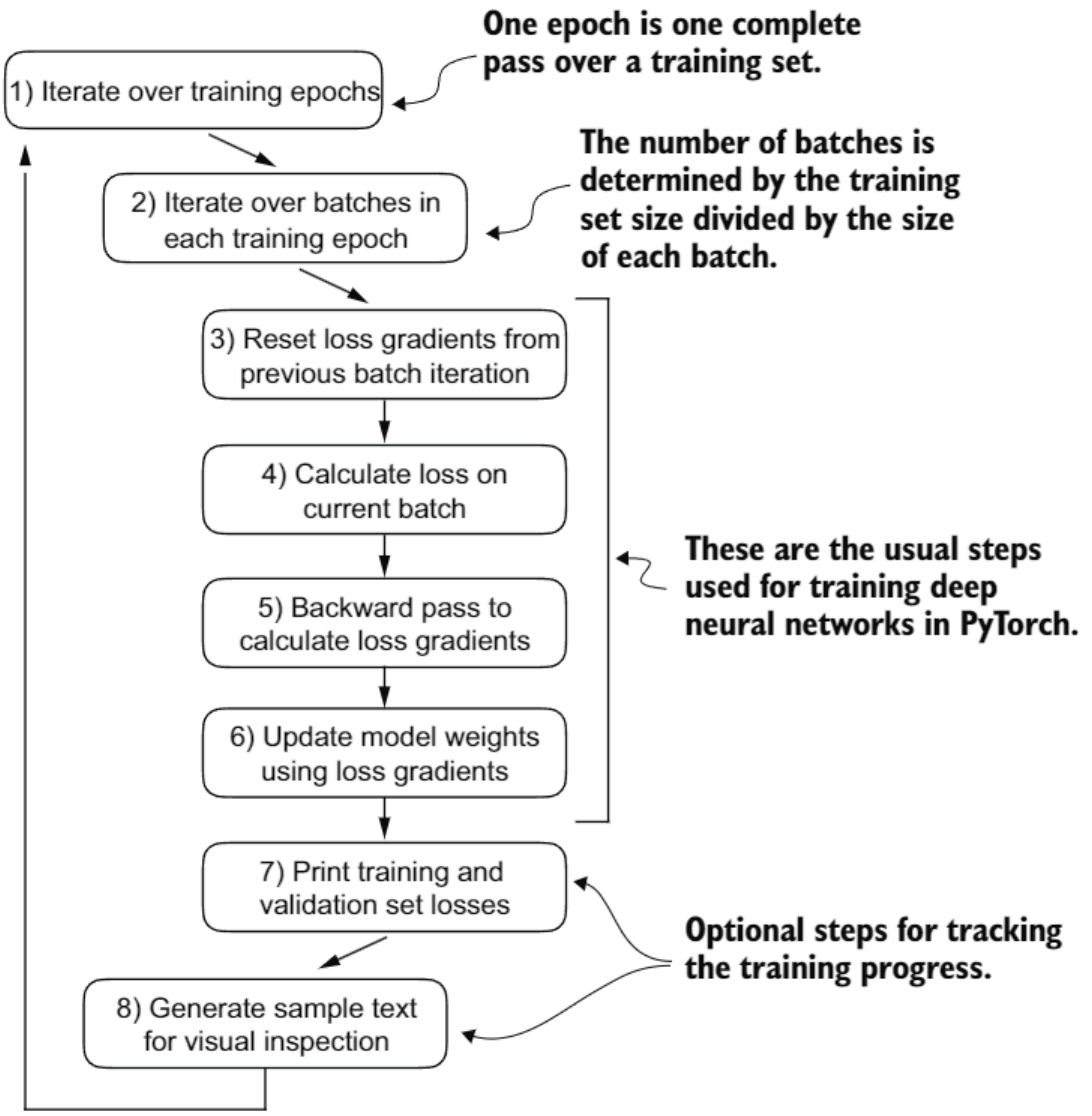

Figure 5.11 A typical training loop for training deep neural networks in PyTorch consists of numerous steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights so that the training set loss is minimized.

The flowchart in figure 5.11 depicts a typical PyTorch neural network training workflow, which we use for training an LLM. It outlines eight steps, starting with iterating over each epoch, processing batches, resetting gradients, calculating the loss and new

接下来，我们将专注于预训练LLM。在模型训练后，我们将实施替代文本生成策略，并保存和加载预训练模型权重。

5.2 训练一个LLM

终于到了实现预训练LLM（我们的 GPTModel）代码的时候了。为此，我们专注于一个简单的训练循环，以保持代码简洁易读。

NOTE 感兴趣的读者可以在附录 D 中了解更多高级技术，包括学习率预热、余弦退火和梯度裁剪。

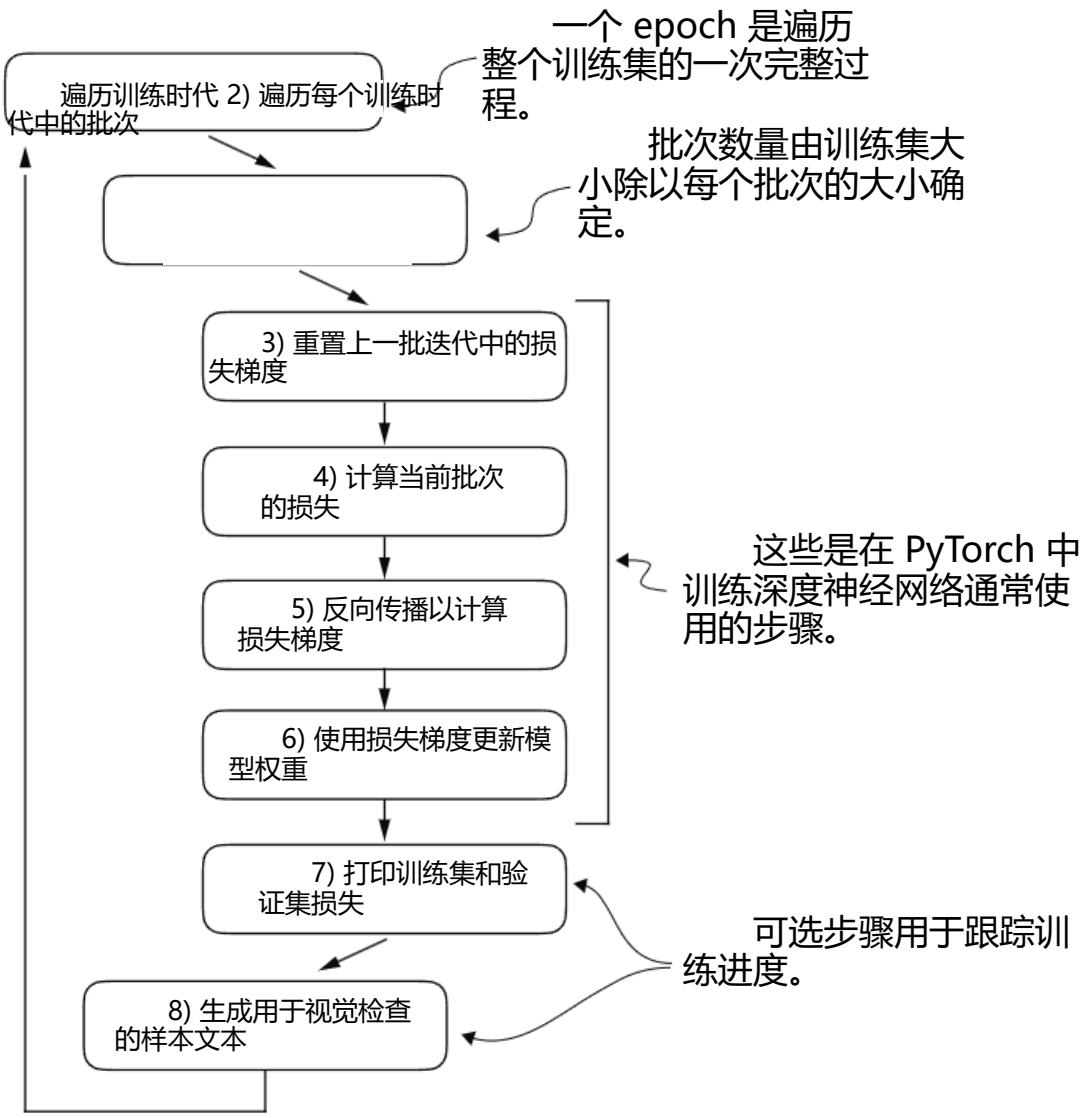

图 5.11 PyTorch 中训练深度神经网络的典型训练循环包括多个步骤，遍历训练集的批次进行多个 epoch。在每个循环中，我们计算每个训练集批次的损失以确定损失梯度，然后使用这些梯度来更新模型权重，以最小化训练集损失。

图 5.11 中的流程图展示了典型的 PyTorch 神经网络训练工作流程，我们用它来训练LLM。它概述了八个步骤，从遍历每个 epoch 开始，处理批次，重置梯度，计算损失和新的

gradients, and updating weights and concluding with monitoring steps like printing losses and generating text samples.

> **NOTE** If you are relatively new to training deep neural networks with PyTorch and any of these steps are unfamiliar, consider reading sections A.5 to A.8 in appendix A.

We can implement this training flow via the `train_model_simple` function in code.

Listing 5.3 The main function for pretraining LLMs

```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        generate_and_print_sample(
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

Initializes lists to track losses and tokens seen

Starts the main training loop

Resets loss gradients from the previous batch iteration

Calculates loss gradients

Updates model weights using loss gradients

Optional evaluation step

Prints a sample text after each epoch

Note that the `train_model_simple` function we just created uses two functions we have not defined yet: `evaluate_model` and `generate_and_print_sample`.

The `evaluate_model` function corresponds to step 7 in figure 5.11. It prints the training and validation set losses after each model update so we can evaluate whether the training improves the model. More specifically, the `evaluate_model` function calculates the loss over the training and validation set while ensuring the model is in eval-

计算损失和新梯度，并更新权重，最后进行如打印损失和生成文本样本的监控步骤。

注意：如果您相对较新于使用 PyTorch 训练深度神经网络，并且对以下步骤不熟悉，请考虑阅读附录 A 中的 A.5 至 A.8 节。

我们可以通过代码中的 train_model_simple 函数实现这个训练流程。

列表 5.3 主预训练功能 LLMs

```
def 训练模型简单(model, train_loader, val_loader,
                                优化器，设备，训练轮数，评估频
    start_context, tokenizer) 评估迭代次数，评估起始语境，训练损失，验证损失，跟踪已见标记 = [], [], [] 已见
标记，全局步数 = 0, -1


    for epoch 在 range(num_epochs):
        model.train()
        for 输入批次，目标批次 in 训练加载器：
            optimizer.zero_grad() 损
        失 = calc_loss_batch()
            输入批次，目标批次，模型，设备）反向传播损失() 步
        骤优化器().step() tokens_seen += input_batch.numel()
        global_step += 1

        如果 global_step % eval_freq 等于 0：
            训练损失，验证损失 = 评估模型(
            模型，训练加载器，验证加载器，设备，评估迭代器）将训练损失添加
        到 train_losses 中 val_losses.append(val_loss)
        track_tokens_seen.append(tokens_seen) 打印(f"Ep {epoch+1} (Step
        {global_step:06d}): ")


            f"训练损失 {train_loss:.3f}, " f"验
        证损失 {val_loss:.3f}"

            生成并打印示例()
    模型，分词器，设备，起始上下文）返回训练损失，验证损
失，跟踪已见令牌
```

初始化列表以跟踪损失和已看到的标记

启动主程序训练循环

重置上一批迭代中的损失梯度

计算损失梯度

使用损失梯度更新模型权重

可选评估步骤

在每个 epoch 后打印示例文本

请注意，我们刚刚创建的 train_model_simple 函数使用了两个函数我们尚未定义：evaluate_model 和 generate_and_print_sample。

该 evaluate_model 函数对应于图 5.11 中的第 7 步。它在每次模型更新后打印训练集和验证集的损失，以便我们可以评估训练是否提高了模型。更具体地说，evaluate_model 函数在确保模型处于 eval-

uation mode with gradient tracking and dropout disabled when calculating the loss over the training and validation sets:

Dropout is disabled during evaluation for stable, reproducible results.

Disables gradient tracking, which is not required during evaluation, to reduce the computational overhead

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

Similar to `evaluate_model`, the `generate_and_print_sample` function is a convenience function that we use to track whether the model improves during the training. In particular, the `generate_and_print_sample` function takes a text snippet (`start_context`) as input, converts it into token IDs, and feeds it to the LLM to generate a text sample using the `generate_text_simple` function we used earlier:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))
    model.train()
```

Compact print format

While the `evaluate_model` function gives us a numeric estimate of the model's training progress, this `generate_and_print_sample` text function provides a concrete text example generated by the model to judge its capabilities during training.

AdamW

Adam optimizers are a popular choice for training deep neural networks. However, in our training loop, we opt for the *AdamW* optimizer. AdamW is a variant of Adam that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights. This adjustment allows AdamW to achieve more effective regularization and better generalization; thus, AdamW is frequently used in the training of LLMs.

训练和验证集计算损失时，禁用梯度跟踪和 dropout 的评估模式：

Dropout 在评估期间被禁用，以确保稳定、可重复的结果。

禁用梯度跟踪，在评估期间不需要，以减少计算开销

```
def 评估模型(model, 训练加载器, 验证加载器, 设备, 评估迭代次数):
    model.eval() 使用
torch.no_grad():
        train_loss = 计算损失加载器(
        train_loader, model, device, num_batches=eval_iter) val_loss =
 calc_loss_loader()


    val_loader、model、device、num_batches=eval_iter) 模型.train()
返回 train_loss、val_loss
```

与 evaluate_model 类似，generate_and_print_sample 函数是一个方便函数，我们用它来跟踪模型在训练过程中的改进情况。特别是，generate_and_print_sample 函数接受一个文本片段（start_context）作为输入，将其转换为 token ID，并将其输入到LLM，使用我们之前使用的 generate_text_simple 函数生成文本样本：

```
def 生成并打印示例(model, tokenizer, device, start_context)
    model.eval() 上下文大小 = model.pos_emb.weight.shape[0] 编码 =
text_to_token_ids(start_context, tokenizer).to(device) with torch.no_grad():


        token_ids = generate_text_simple(
    model=model, idx=encoded, 最大新令牌数=50, 上下文大小
=context_size ) 解码文本 = token_ids_to_text(token_ids, tokenizer)
打印(decoded_text.replace("\n", " ")) model.train()
```

紧凑打印格式

虽然 evaluate_model 函数为我们提供了模型训练进度的数值估计，但 generate_and_print_sample 文本函数提供了由模型生成的具体文本示例，以便在训练过程中判断其能力。

AdamW

Adam 优化器是训练深度神经网络的流行选择。然而，在我们的训练循环中，我们选择 AdamW 优化器。AdamW 是 Adam 的一个变体，它改进了权重衰减方法，旨在通过惩罚较大的权重来最小化模型复杂度并防止过拟合。这种调整使得 AdamW 能够实现更有效的正则化和更好的泛化；因此，AdamW 经常用于LLMs的训练中。

Let's see this all in action by training a `GPTModel` instance for 10 epochs using an `AdamW` optimizer and the `train_model_simple` function we defined earlier:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
     model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

The .parameters() method returns all trainable weight parameters of the model.

Executing the `train_model_simple` function starts the training process, which takes about 5 minutes to complete on a MacBook Air or a similar laptop. The output printed during this execution is as follows:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and,
 and, and, and, and, and, and, and, and, and, and, and,, and, and,
[...]
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?"  "Yes--quite insensible to the irony. She wanted
him vindicated--and by me!"  He laughed again, and threw back the
window-curtains, I had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed luncheon-table, when, on a
later day, I had again run over from Monte Carlo; and Mrs. Gis
```

Intermediate results removed to save space

As we can see, the training loss improves drastically, starting with a value of 9.781 and converging to 0.391. The language skills of the model have improved quite a lot. In the beginning, the model is only able to append commas to the start context (`Every effort moves you,,,,,,,,,,,,`) or repeat the word `and`. At the end of the training, it can generate grammatically correct text.

Similar to the training set loss, we can see that the validation loss starts high (9.933) and decreases during the training. However, it never becomes as small as the training set loss and remains at 6.452 after the 10th epoch.

Before discussing the validation loss in more detail, let's create a simple plot that shows the training and validation set losses side by side:

让我们通过使用 AdamW 优化器和之前定义的 train_model_simple 函数，训练一个 GPTModel 实例 10 个 epoch，来看看这一切是如何实现的：

```
torch.manual_seed(123) 模型 = GPTModel(GPT_CONFIG_124M) 模型.to(设备)
优化器 = torch.optim.AdamW(
    model.parameters(), lr=0.0004, weight_decay=0.1) num_epochs = 10 训练损失，验证损失，已见令牌数量 = train_model_simple()
    模型，训练加载器，验证加载器，优化器，设备，epoch 数=num_epochs，评估频率=5，评估迭代次数=5，起始上下文="每一步努力都会推动你"，分词器=分词器()
```

该 .parameters() 方法返回模型的所有可训练权重参数。

执行 train_model_simple 函数开始训练过程，在 MacBook Air 或类似笔记本电脑上完成大约需要 5 分钟。在此执行过程中打印的输出如下：

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933 Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339 Every effort moves you,,,,,,,,,,,,. Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048 Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616 Every effort moves you, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and,, and, and, [...] Ep 9 (Step 000080): Train loss 0.391, Val loss 6.393 Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me!" He laughed again, and threw back the window-curtains, I had the donkey. "There were days when I Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452 Every effort moves you know," was one of the axioms he laid down across the Sevres and silver of an exquisitely appointed luncheon-table, when, on a later day, I had again run over from Monte Carlo; and Mrs. Gis
```

中级结果已移除以节省空间

随着我们可以看到，训练损失急剧下降，从 9.781 的值收敛到 0.391。该模型的语言技能有了很大提升。一开始，模型只能将逗号添加到起始上下文（每一个努力都推动你,,,,,,,,,,,,）或重复单词 and。训练结束时，它能够生成语法正确的文本。

与训练集损失类似，我们可以看到验证损失开始较高（9.933）并在训练过程中下降。然而，它从未变得像训练集损失那样小，并在第 10 个 epoch 后保持在 6.452。

在更详细地讨论验证损失之前，让我们创建一个简单的图表，展示训练集和验证集损失并排显示：

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(
        epochs_seen, val_losses, linestyle="-.", label="Validation loss"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Creates a second x-axis that shares the same y-axis

Invisible plot for aligning ticks

The resulting training and validation loss plot is shown in figure 5.12. As we can see, both the training and validation losses start to improve for the first epoch. However, the losses start to diverge past the second epoch. This divergence and the fact that the validation loss is much larger than the training loss indicate that the model is overfitting to the training data. We can confirm that the model memorizes the training data verbatim by searching for the generated text snippets, such as `quite insensible to the irony` in the "The Verdict" text file.

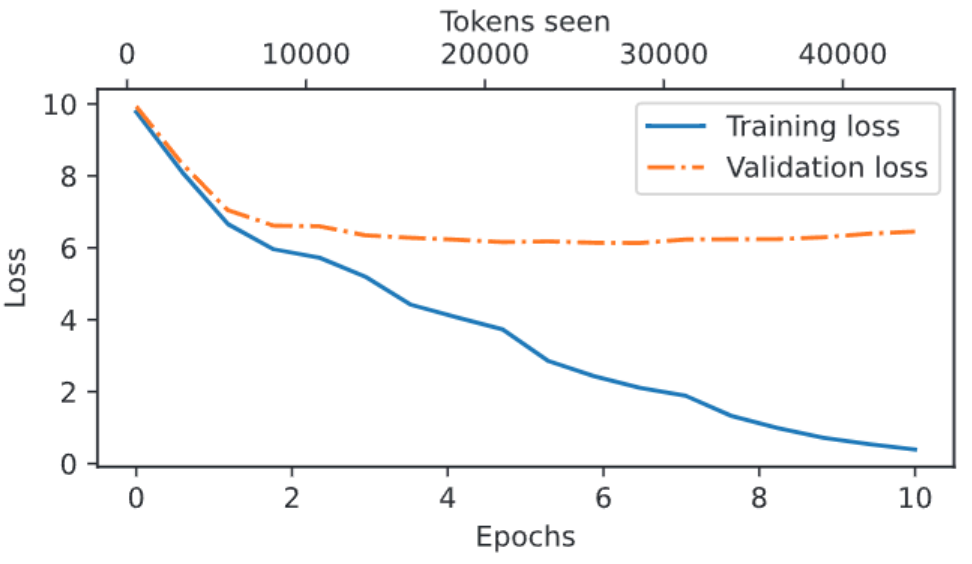

Figure 5.12 At the beginning of the training, both the training and validation set losses sharply decrease, which is a sign that the model is learning. However, the training set loss continues to decrease past the second epoch, whereas the validation loss stagnates. This is a sign that the model is still learning, but it's overfitting to the training set past epoch 2.

```
    import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
        fig, ax1 = plt.subplots(figsize=(5, 3)) ax1.plot(epochs_seen,
    train_losses, 标签="训练损失") ax1.plot(
        epochs_seen, val_losses, 样式="-.", 标签="验证损失" ) ax1.set_xlabel("时代")
    ax1.set_ylabel("损失") ax1.legend(loc="右上角")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True)) ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, 透明度=0) ax2.set_xlabel("已见令牌")
    fig.tight_layout() plt.show()
```

创建第二个 x 轴共享相同的 y 轴

隐形剧情对齐刻度

```
    epochs_tensor = torch.linspace(0, num_epochs, len(train_losses)) 绘制
train_losses 的 epochs_tensor, tokens_seen, train_losses
                                                                    val_losses)
```

结果训练和验证损失图如图 5.12 所示。如图所示，训练和验证损失在第一个 epoch 开始改善。然而，损失在第二个 epoch 之后开始发散。这种发散以及验证损失远大于训练损失表明模型对训练数据过拟合。我们可以通过搜索生成的文本片段来确认模型逐字逐句地记住了训练数据，例如在“The Verdict”文本文件中对讽刺的相当不敏感。

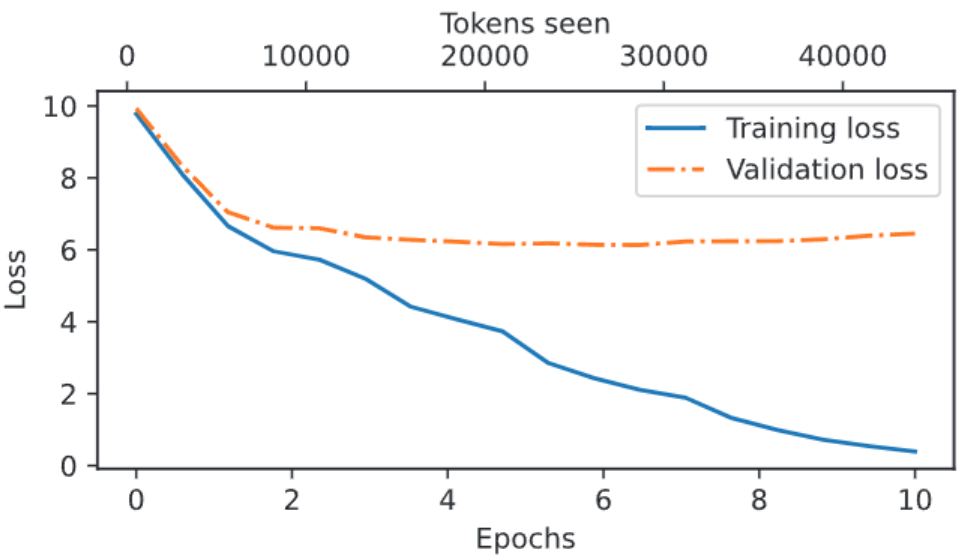

图 5.12 在训练开始时，训练集和验证集的损失急剧下降，这是模型正在学习的标志。然而，训练集损失在第二个 epoch 之后继续下降，而验证集损失停滞。这是模型仍在学习，但超过第二个 epoch 后对训练集过拟合的标志。

This memorization is expected since we are working with a very, very small training dataset and training the model for multiple epochs. Usually, it's common to train a model on a much larger dataset for only one epoch.

NOTE As mentioned earlier, interested readers can try to train the model on 60,000 public domain books from Project Gutenberg, where this overfitting does not occur; see appendix B for details.

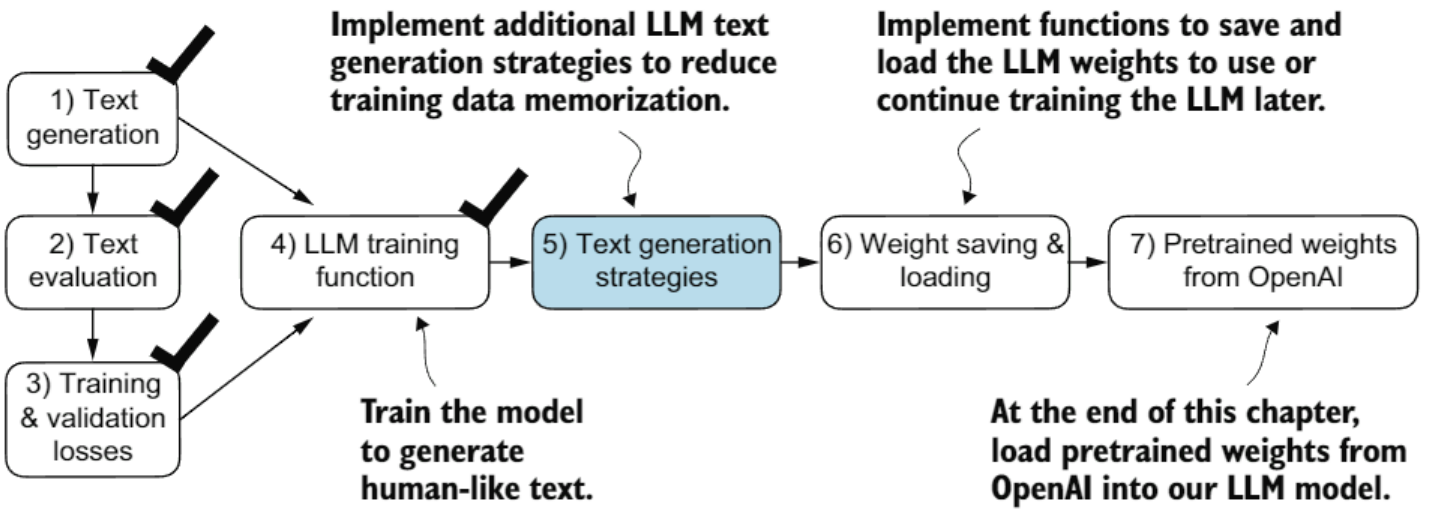

Figure 5.13 Our model can generate coherent text after implementing the training function. However, it often memorizes passages from the training set verbatim. Next, we will discuss strategies to generate more diverse output texts.

As illustrated in figure 5.13, we have completed four of our objectives for this chaper. Next, we will cover text generation strategies for LLMs to reduce training data memorization and increase the originality of the LLM-generated text before we cover weight loading and saving and loading pretrained weights from OpenAI's GPT model.

5.3 Decoding strategies to control randomness

Let's look at text generation strategies (also called decoding strategies) to generate more original text. First, we will briefly revisit the `generate_text_simple` function that we used inside `generate_and_print_sample` earlier. Then we will cover two techniques, *temperature scaling* and *top-k sampling*, to improve this function.

We begin by transferring the model back from the GPU to the CPU since inference with a relatively small model does not require a GPU. Also, after training, we put the model into evaluation mode to turn off random components such as dropout:

```
model.to("cpu")
model.eval()
```

Next, we plug the `GPTModel` instance (`model`) into the `generate_text_simple` function, which uses the LLM to generate one token at a time:

这种记忆是预期的，因为我们正在使用一个非常、非常小的训练数据集，并且对模型进行多次迭代训练。通常，在仅一个迭代周期内，在更大的数据集上训练模型是常见的。

注意：如前所述，感兴趣的读者可以尝试在 Project Gutenberg 的 60,000 本公共领域书籍上训练模型，这里不会出现过拟合；详情见附录 B。

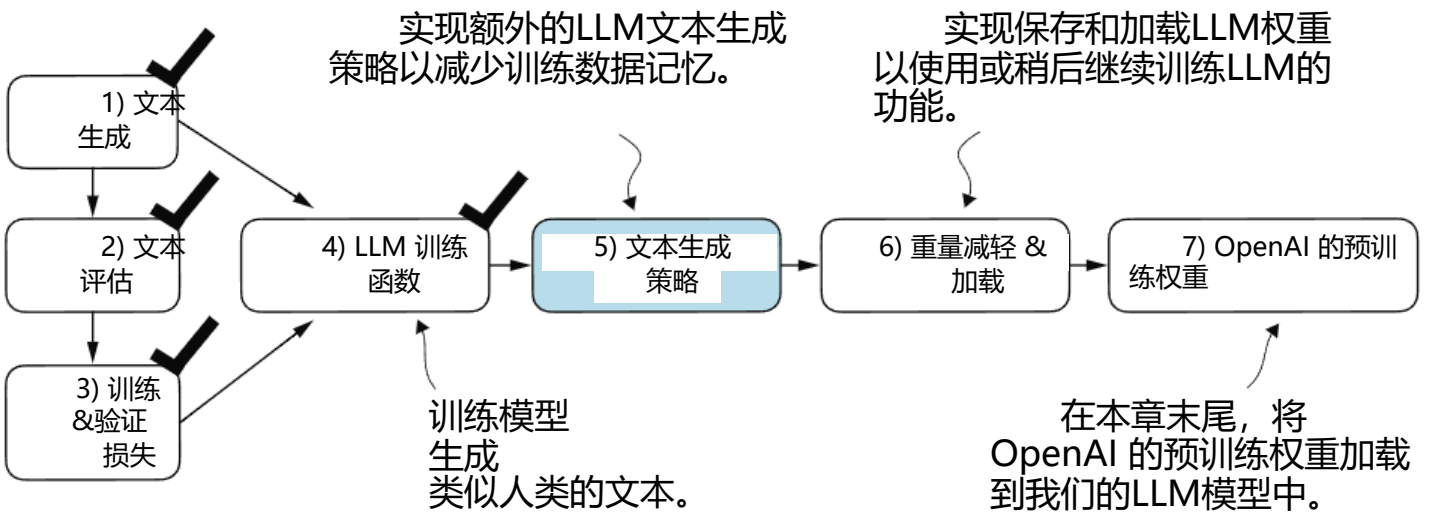

图 5.13 我们在实现训练函数后，模型可以生成连贯的文本。然而，它经常逐字逐句地记住训练集中的段落。接下来，我们将讨论生成更多样化输出文本的策略。

如图 5.13 所示，我们已完成了本章四个目标中的四个。接下来，在介绍权重加载和保存以及从 OpenAI 的 GPT 模型中加载预训练权重之前，我们将讨论针对LLMs的文本生成策略，以减少训练数据记忆并提高LLM生成文本的原创性。

5.3 解码策略以控制随机性

让我们看看文本生成策略（也称为解码策略），以生成更多原创文本。首先，我们将简要回顾一下我们在 generate_and_print_sample 中使用的 generate_text_simple 函数。然后，我们将介绍两种技术，即温度缩放和 top-k 样本，以改进此函数。

我们首先将模型从 GPU 转移到 CPU，因为使用相对较小的模型进行推理不需要 GPU。此外，在训练后，我们将模型置于评估模式以关闭随机组件，如 dropout：

```
model.to("cpu")
model.eval()
```

接下来，我们将 GPTModel 实例（模型）插入到 generate_text_simple 函数中，该函数使用LLM逐个生成一个标记：

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is

```
Output text:
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed lun
```

As explained earlier, the generated token is selected at each generation step corresponding to the largest probability score among all tokens in the vocabulary. This means that the LLM will always generate the same outputs even if we run the preceding `generate_text_simple` function multiple times on the same start context (`Every effort moves you`).

5.3.1 Temperature scaling

Let's now look at temperature scaling, a technique that adds a probabilistic selection process to the next-token generation task. Previously, inside the `generate_text_simple` function, we always sampled the token with the highest probability as the next token using `torch.argmax`, also known as *greedy decoding*. To generate text with more variety, we can replace `argmax` with a function that samples from a probability distribution (here, the probability scores the LLM generates for each vocabulary entry at each token generation step).

To illustrate the probabilistic sampling with a concrete example, let's briefly discuss the next-token generation process using a very small vocabulary for illustration purposes:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}
```

```
    tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    模型=model, 索引=text_to_token_ids("Every effort moves you", 分词器), 最
大新令牌数=25, 上下文大小=GPT_CONFIG_124M["context_length"] ) 打印("输出")
```

```
                                文本: ", 将 token_ids 转换为文本(token_ids_to_text(token_ids, tokenizer))"
```

生成的文本是

```
输出文本：
每一份努力都让你知道，“这是他在 Sevres 和银器上精心布置的午餐时提出的格言之一
```

如前所述，每次生成步骤都会选择词汇表中所有标记中概率分数最大的标记生成的标记。这意味着即使我们在相同的起始上下文（Every）上多次运行 preceding generate_text_simple 函数，LLM 也会始终生成相同的输出。

```
努力使人进步）。
```

5.3.1 温度缩放

现在让我们看看温度缩放技术，这是一种将概率选择过程添加到下一个标记生成任务中的技术。之前，在 generate_text_simple 函数内部，我们总是使用 torch.argmax 样本化概率最高的标记作为下一个标记，也称为贪婪解码。为了生成更多样化的文本，我们可以用从概率分布中采样的函数替换 argmax（在这里，是 LLM 在每个标记生成步骤为每个词汇条目生成的概率分数）。

为了说明概率抽样的具体例子，让我们简要讨论使用一个非常小的词汇量来展示下一个标记生成过程：

```
vocab = {
     "closer": 0,
     "every": 1,
     "effort": 2,
     "forward": 3
     英寸：4
     "moves": 5,
     披萨：6
     向：7
     "you": 你，8,
}
逆词汇表          = {v: k for k, v in 词汇.items()}
```

Next, assume the LLM is given the start context `"every effort moves you"` and generates the following next-token logits:

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

As discussed in chapter 4, inside `generate_text_simple`, we convert the logits into probabilities via the `softmax` function and obtain the token ID corresponding to the generated token via the `argmax` function, which we can then map back into text via the inverse vocabulary:

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

Since the largest logit value and, correspondingly, the largest softmax probability score are in the fourth position (index position 3 since Python uses 0 indexing), the generated word is `"forward"`.

To implement a probabilistic sampling process, we can now replace `argmax` with the `multinomial` function in PyTorch:

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

The printed output is `"forward"` just like before. What happened? The `multinomial` function samples the next token proportional to its probability score. In other words, `"forward"` is still the most likely token and will be selected by `multinomial` most of the time but not all the time. To illustrate this, let's implement a function that repeats this sampling 1,000 times:

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item()
              for i in range(1_000)]
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")

print_sampled_tokens(probas)
```

The sampling output is

```
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
```

接下来，假设LLM被赋予起始上下文“每一步努力都能推动你”并生成以下下一个标记的对数概率：

```
next_token_logits = torch.tensor(
[4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79] )
```

如第 4 章所述，在 generate_text_simple 函数内部，我们通过 softmax 函数将 logits 转换为概率，并通过 argmax 函数获取生成 token 对应的 token ID，然后我们可以通过逆词汇表将其映射回文本：

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

由于最大的对数似然值以及相应的最大的 softmax 概率分数位于第四位（由于 Python 使用 0 索引，因此是索引位置 3），生成的单词是"forward"。

要实现概率抽样过程，我们现在可以在 PyTorch 中将 argmax 替换为多项式函数：

```
torch 手动设置随机种子(123) next_token_id = torch.multinomial(probas,
num_samples=1).item() 打印(inverse_vocab[next_token_id])
```

打印输出仍然是“forward”，就像之前一样。发生了什么？多项式函数根据其概率分数采样下一个标记。换句话说，

"forward" 仍然是可能性最高的标记，并且大多数情况下会被多项式选择，但并非总是如此。为了说明这一点，让我们实现一个函数，重复进行这种采样 1,000 次：

```
def 打印采样标记(probas):
    torch.manual_seed(123) 样本 =
[torch.multinomial(probas,                    num_samples=1).item()
    在 range(1_000)中 i] sampled_ids =
torch.bincount(torch.tensor(sample)) for i, freq in
enumerate(sampled_ids):
        打印(f"{freq} x {inverse_vocab[i]}")

打印采样标记（probas）
```

采样输出

```
73 x 更接近
0 每个 x
0 x 努力
582 x 前进
2 x 英寸
```

```
0 x moves
0 x pizza
343 x toward
```

As we can see, the word `forward` is sampled most of the time (582 out of 1,000 times), but other tokens such as `closer`, `inches`, and `toward` will also be sampled some of the time. This means that if we replaced the `argmax` function with the `multinomial` function inside the `generate_and_print_sample` function, the LLM would sometimes generate texts such as `every effort moves you toward`, `every effort moves you inches`, and `every effort moves you closer` instead of `every effort moves you forward`.

We can further control the distribution and selection process via a concept called *temperature scaling*. Temperature scaling is just a fancy description for dividing the logits by a number greater than 0:

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Temperatures greater than 1 result in more uniformly distributed token probabilities, and temperatures smaller than 1 will result in more confident (sharper or more peaky) distributions. Let's illustrate this by plotting the original probabilities alongside probabilities scaled with different temperature values:

```
temperatures = [1, 0.1, 5]                                    # Original, lower, and higher confidence
scaled_probas = [softmax_with_temperature(next_token_logits, T)
                 for T in temperatures]
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

The resulting plot is shown in figure 5.14.

A temperature of 1 divides the logits by 1 before passing them to the `softmax` function to compute the probability scores. In other words, using a temperature of 1 is the same as not using any temperature scaling. In this case, the tokens are selected with a probability equal to the original softmax probability scores via the `multinomial` sampling function in PyTorch. For example, for the temperature setting 1, the token corresponding to "forward" would be selected about 60% of the time, as we can see in figure 5.14.

```
0 x 移动
0 x 披萨
343 x 朝着
```

我们可以看到，单词 forward 在大多数情况下被采样（1000 次中有 582 次），但其他标记如 closer、inches 和 toward 也会偶尔被采样。这意味着如果我们将 generate_and_print_sample 函数内的 argmax 函数替换为多项式函数，LLM将会...

每一次努力都让你更近一步，每一次努力都让你前进寸许，每一次努力都让你更靠近，而不是每一次努力都让你向前。

我们可以通过一个称为温度缩放的概念进一步控制分布和选择过程。温度缩放只是将 logits 除以一个大于 0 的数的花哨说法：

```
def softmax_with_temperature(logits, 温度):
    缩放后的 logits = logits / 温度
返回 torch.softmax(缩放后的 logits,)                dim=0)
```

温度大于 1 会导致标记概率分布更均匀，而温度小于 1 将导致更自信（更尖锐或更峰值）的分布。让我们通过绘制原始概率与不同温度值缩放的概率并排来展示这一点：

```
温度 = [1, 0.1, 5] 缩放概率
=                    [softmax_with_temperature(下一_token_概率, 温度) for
                  温度 in 温度列表]
    x = torch.arange(len(词汇)) bar_width =
0.15 fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):


        rects = ax.bar(x) + i * 栏宽, 缩放概率[i], 栏宽,
                                    label=f'温度 = {T}'
    ax.set_ylabel('概率') ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend() plt.tight_layout() plt.show()
```

原始，较低，并且更高的置信度

结果图示于图 5.14 中。

温度为 1 时，在将 logits 传递给 softmax 函数计算概率分数之前，将它们除以 1。换句话说，使用温度为 1 等同于不使用任何温度缩放。在这种情况下，通过 PyTorch 中的多项式采样函数，标记的选择概率等于原始 softmax 概率分数。例如，对于温度设置 1，对应于“forward”的标记大约有 60%的时间被选中，如图 5.14 所示。

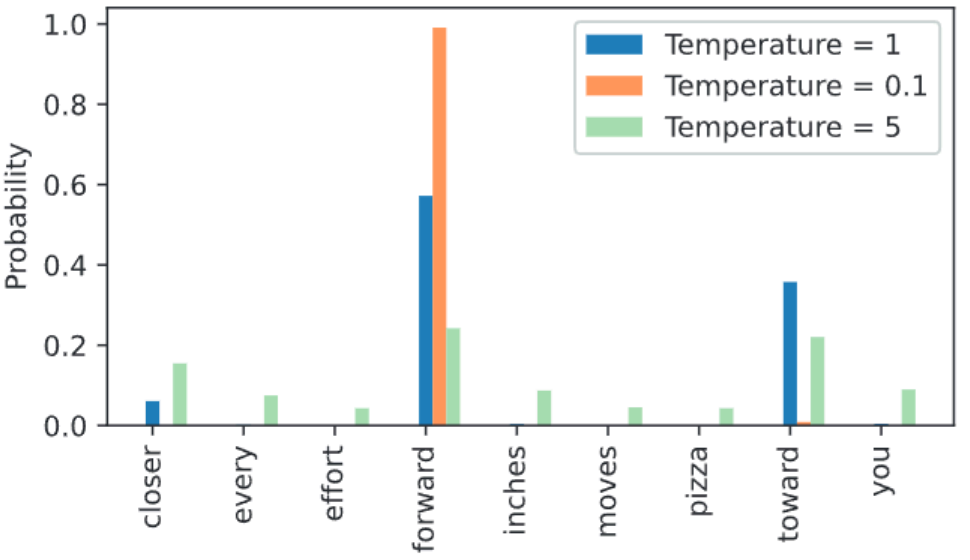

Figure 5.14 A temperature of 1 represents the unscaled probability scores for each token in the vocabulary. Decreasing the temperature to 0.1 sharpens the distribution, so the most likely token (here, "forward") will have an even higher probability score. Likewise, increasing the temperature to 5 makes the distribution more uniform.

Also, as we can see in figure 5.14, applying very small temperatures, such as 0.1, will result in sharper distributions such that the behavior of the `multinomial` function selects the most likely token (here, `"forward"`) almost 100% of the time, approaching the behavior of the `argmax` function. Likewise, a temperature of 5 results in a more uniform distribution where other tokens are selected more often. This can add more variety to the generated texts but also more often results in nonsensical text. For example, using the temperature of 5 results in texts such as `every effort moves you pizza` about 4% of the time.

> **Exercise 5.1**
> Use the `print_sampled_tokens` function to print the sampling frequencies of the softmax probabilities scaled with the temperatures shown in figure 5.14. How often is the word `pizza` sampled in each case? Can you think of a faster and more accurate way to determine how often the word `pizza` is sampled?

5.3.2 Top-k sampling

We've now implemented a probabilistic sampling approach coupled with temperature scaling to increase the diversity of the outputs. We saw that higher temperature values result in more uniformly distributed next-token probabilities, which result in more diverse outputs as it reduces the likelihood of the model repeatedly selecting the most probable token. This method allows for the exploring of less likely but potentially more interesting and creative paths in the generation process. However, one downside of this approach is that it sometimes leads to grammatically incorrect or completely nonsensical outputs such as `every effort moves you pizza`.

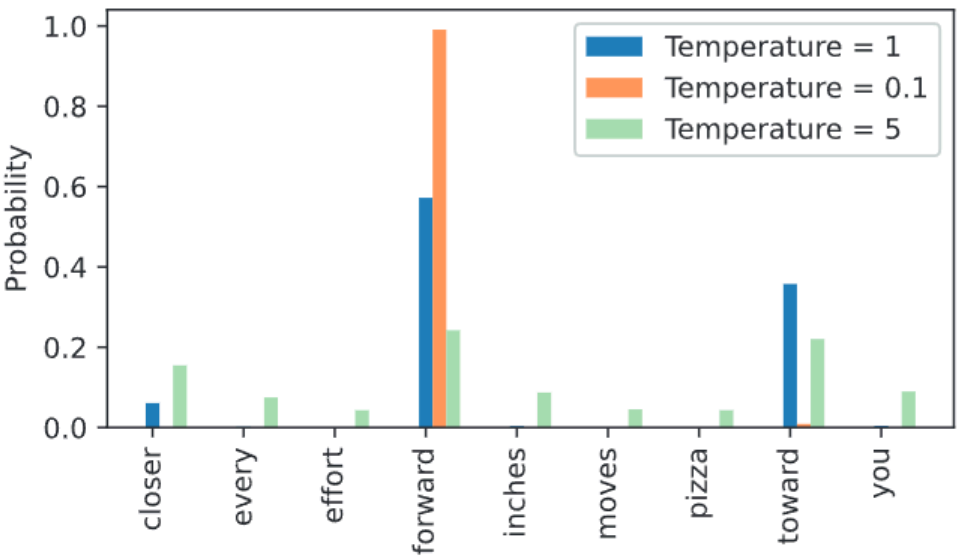

图 5.14 温度为 1 表示词汇表中每个标记的未缩放概率分数。将温度降低到 0.1 会锐化分布，因此最可能的标记（此处为“forward”）将具有更高的概率分数。同样，将温度提高到 5 会使分布更加均匀。

此外，如图 5.14 所示，应用非常小的温度，如 0.1，将导致分布更加尖锐，使得多项式函数的行为几乎 100%选择最可能的标记（此处为“forward”），接近 argmax 函数的行为。同样，温度为 5 会导致分布更加均匀，其他标记被选中的频率更高。这可以为生成的文本增加更多多样性，但同时也更经常导致无意义的文本。例如，使用温度为 5 的结果是文本如“每一点努力都推动你”

披萨 大约 4%的时间。

练习 5.1

使用 print_sampled_tokens 函数打印带有图 5.14 所示温度缩放的 softmax 概率的采样频率。在每种情况下，单词 pizza 被采样的频率是多少？你能想到一种更快更准确的方法来确定单词 pizza 的采样频率吗？

5.3.2 顶级采样

我们现在实现了一种结合温度缩放的概率采样方法，以增加输出的多样性。我们发现，较高的温度值导致下一个标记的概率分布更加均匀，这减少了模型反复选择最可能标记的可能性，从而产生更多样化的输出。这种方法允许在生成过程中探索不太可能但可能更有趣和创造性的路径。然而，这种方法的一个缺点是，有时会导致语法错误或完全无意义的输出，例如“每个努力都让你披萨移动”。

Top-k sampling, when combined with probabilistic sampling and temperature scaling, can improve the text generation results. In top-k sampling, we can restrict the sampled tokens to the top-k most likely tokens and exclude all other tokens from the selection process by masking their probability scores, as illustrated in figure 5.15.

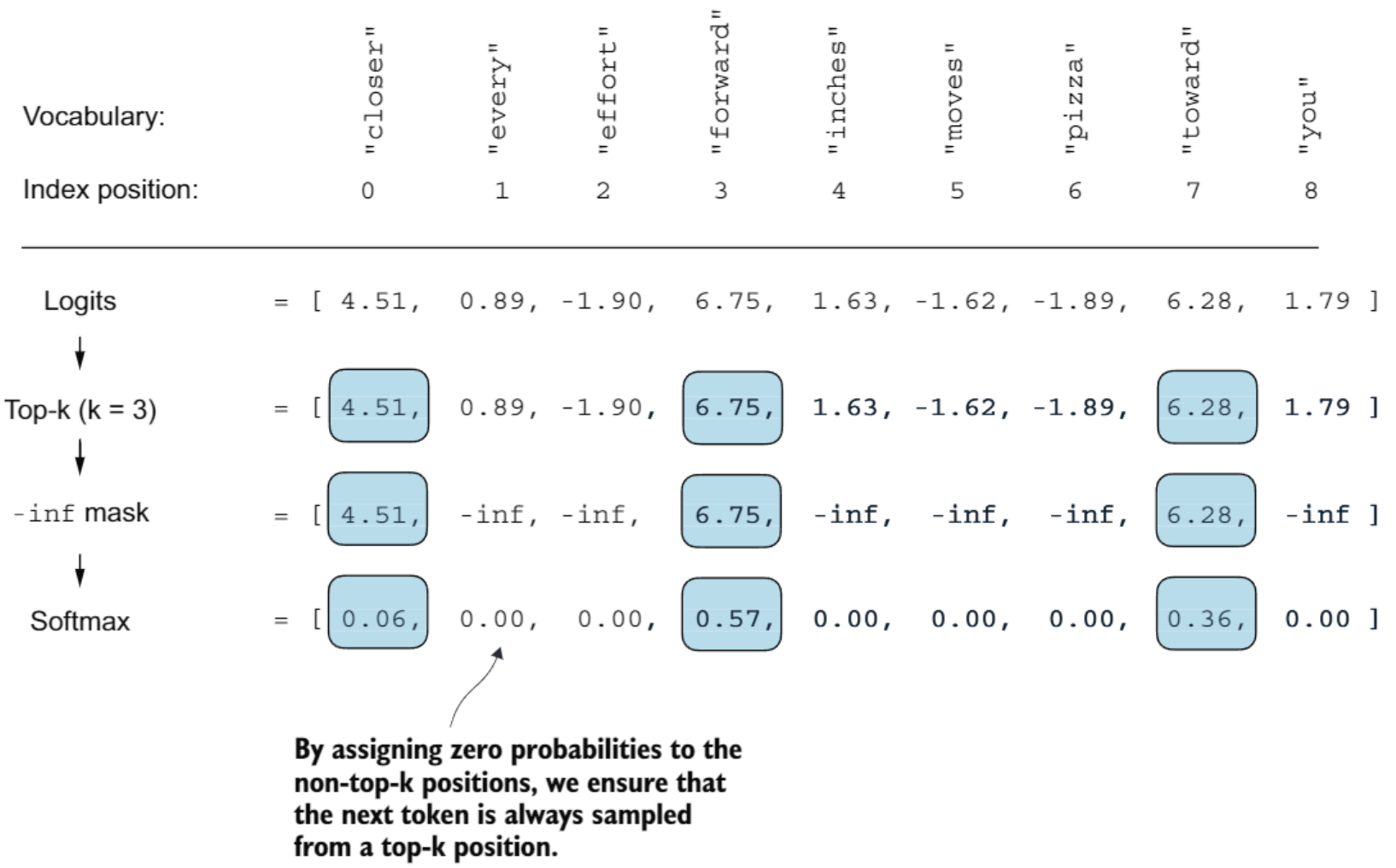

Figure 5.15 Using top-k sampling with k = 3, we focus on the three tokens associated with the highest logits and mask out all other tokens with negative infinity (`-inf`) before applying the `softmax` function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in the "Softmax" row should add up to 1.0.)

The top-k approach replaces all nonselected logits with negative infinity value (`-inf`), such that when computing the softmax values, the probability scores of the non-top-k tokens are 0, and the remaining probabilities sum up to 1. (Careful readers may remember this masking trick from the causal attention module we implemented in chapter 3, section 3.5.1.)

In code, we can implement the top-k procedure in figure 5.15 as follows, starting with the selection of the tokens with the largest logit values:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

顶-k 采样，结合概率采样和温度缩放，可以提高文本生成结果。在顶-k 采样中，我们可以将采样的标记限制为最可能的顶-k 标记，并通过掩码它们的概率分数，排除所有其他标记的选择过程，如图 5.15 所示。

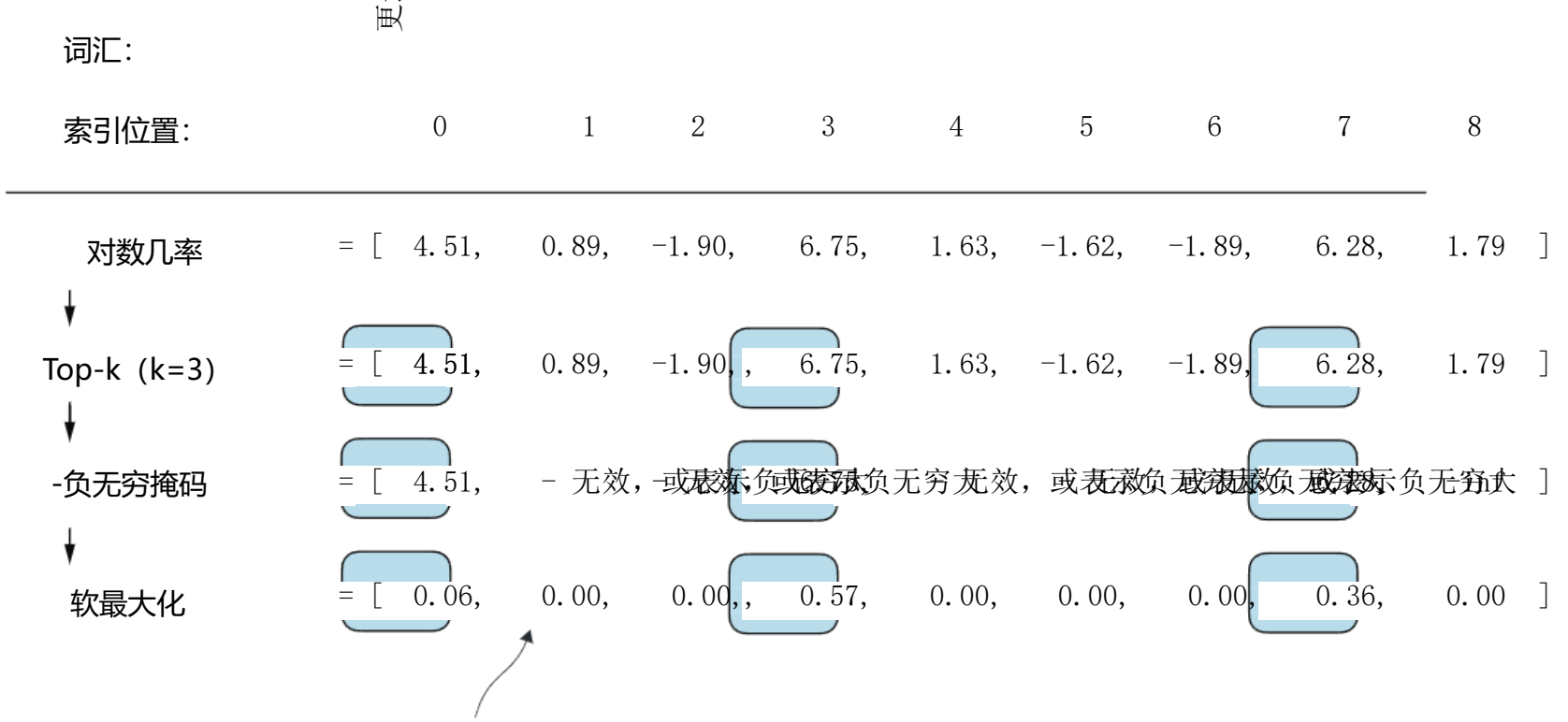

Figure 5.15 Using top-k sampling with k = 3, we focus on the three tokens associated with the highest logits and mask out all other tokens with negative infinity (–inf) before applying the softmax function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in the "Softmax" row should add up to 1.0.)

将非选中 logits 替换为负无穷大值（-inf），使得在计算 softmax 值时，非 top-k 标记的概率得分为 0，剩余概率之和为 1。（仔细的读者可能会记得我们在第 3 章第 3.5.1 节中实现的因果注意力模块中的这个掩码技巧。）

在代码中，我们可以按照图 5.15 所示实现 top-k 过程，从选择具有最大 logit 值的标记开始：

```
    top_k = 3 top_logits, top_pos = torch.topk(next_token_logits,
top_k) 打印("Top logits:", top_logits) 打印("Top positions:", top_pos)
```

The logits values and token IDs of the top three tokens, in descending order, are

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Subsequently, we apply PyTorch's `where` function to set the logit values of tokens that are below the lowest logit value within our top-three selection to negative infinity (`-inf`):

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float('-inf')),
    other=next_token_logits
)
print(new_logits)
```

Identifies logits less than the minimum in the top 3

Assigns –inf to these lower logits

Retains the original logits for all other tokens

The resulting logits for the next token in the nine-token vocabulary are

```
tensor([4.5100,   -inf,   -inf, 6.7500,   -inf,   -inf,   -inf, 6.2800,
     -inf])
```

Lastly, let's apply the `softmax` function to turn these into next-token probabilities:

```
topk_probas = torch.softmax(new_logits, dim=0)
print(topk_probas)
```

As we can see, the result of this top-three approach are three non-zero probability scores:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610,
        0.0000])
```

We can now apply the temperature scaling and multinomial function for probabilistic sampling to select the next token among these three non-zero probability scores to generate the next token. We do this next by modifying the text generation function.

5.3.3 Modifying the text generation function

Now, let's combine temperature sampling and top-k sampling to modify the `generate_text_simple` function we used to generate text via the LLM earlier, creating a new `generate` function.

Listing 5.4 A modified text generation function with more diversity

```
def generate(model, idx, max_new_tokens, context_size,
             temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
```

The for loop is the same as before: gets logits and only focuses on the last time step.

下一行文本的简体中文翻译为：
三个最高 token 的 logits 值和 token ID，按降序排列
顶部对数概率： 张量([6.7500, 6.2800, 4.5100])
顶级位置：tensor([3, 7, 0])

随后，我们使用 PyTorch 的 where 函数将我们前三项选择中低于最低 logit 值的 token 的 logit 值设置为负无穷大（-inf）：

```
new_logits = torch.where(
condition=下一_token_概率 < 最高_概率[-1],
input=torch.tensor(float('-inf')), other=下一_token_概率 )
打印(new_logits)
```

识别出前 3 个中小于最小值的 logits

分配-无穷大值给这些较低的 logits

保留所有其他标记的原始 logits

下一个九元词汇中下一个标记的结果对数是

```
张量([4.5100,  - 无效，或表示负无穷大,  6.7500,  - 无效，或表示负无穷大,  - 无效，或表示负无穷大,  - 无效，或表示负无穷大,  - 无效，或表示负无穷大,  - 无效，或表示负无穷大,
      [-inf])
```

最后，让我们应用 softmax 函数将这些转换为下一个标记的概率：

```
topk_probas = torch.softmax(new_logits, dim=0) 打印
(topk_probas)
```

我们可以看到，这种前三名方法的结果是三个非零概率分数：

```
张量([0.0615, 0.0000,])  0.0000, 0.5775,  0.0000, 0.0000, 0.0000,  0.3610,
      0.0000])
```

我们现在可以应用温度缩放和多项式函数进行概率采样，从这三个非零概率分数中选择下一个标记来生成下一个标记。我们接下来通过修改文本生成函数来实现这一点。

5.3.3 修改文本生成函数

现在，让我们结合温度采样和 top-k 采样来修改 generate_文本简单生成函数。我们之前使用LLM生成文本的功能，创建了一个新的函数。

列表 5.4 修改后的文本生成函数，具有更多样性

```
def generate(模型, 索引, 最大新令牌数, 上下文大小,)
                temperature=0.0, top_k=None, eos_id=None): 温度=0.0, top_k=None, eos_id=None):
    for _ in 范围(最大新令牌数):
        idx_cond = idx[:, -context_size:]
     with torch.no_grad():
        logits = 模型(idx_cond) logits
     = logits[:, -1, :]
```

for 循环与之前相同：获取 logits，仅关注最后一个时间步。

```
        if top_k is not None:
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )
        if temperature > 0.0:
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        if idx_next == eos_id:
            break
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

Filters logits with top_k sampling

Applies temperature scaling

Carries out greedy next-token selection as before when temperature scaling is disabled

Stops generating early if end-of-sequence token is encountered

Let's now see this new `generate` function in action:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is

```
Output text:
 Every effort moves you stand to work on surprise, a one of us had gone
 with random-
```

As we can see, the generated text is very different from the one we previously generated via the `generate_simple` function in section 5.3 (`"Every effort moves you know," was one of the axioms he laid...!`), which was a memorized passage from the training set.

Exercise 5.2

Play around with different temperatures and top-k settings. Based on your observations, can you think of applications where lower temperature and top-k settings are desired? Likewise, can you think of applications where higher temperature and top-k settings are preferred? (It's recommended to also revisit this exercise at the end of the chapter after loading the pretrained weights from OpenAI.)

```
        如果 top_k 不为 None：
            top_logits, _ = torch.topk(logits, top_k)
         min_val = top_logits[:, -1]
         logits = torch.where(
        logits < min_val, torch.tensor(float('-
    inf')).to(logits.device), logits) 如果 temperature > 0.0:
        logits = logits / temperature
    probs = torch.softmax(logits, dim=-1)
    idx_next = torch.multinomial(probs, num_samples=1)
    else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        如果 idx_next 等于 eos_id：
            break
        中断
    idx = torch.cat((idx, idx_next), dim=1) return
idx
```

过滤对数几率
顶级采样

适用
温度
缩放

执行
贪婪的下一个
令牌选择
之前一样
温度
缩放是
禁用

如果遇到序列结束标记则提前停止生成

现在让我们看看这个新生成函数的实际应用：

```
    torch 手动设置随机种子
(123) token_ids =
generate()
    model=model, idx=文本_to_token_ids("Every effort moves you", tokenizer),
最大新令牌数=15, 上下文大小=GPT_CONFIG_124M["context_length"], top_k=25, 温
度=1.4 ) 打印("输出")
```

```
文本: ", 将 token_ids 转换为文本(token_ids_to_text(token_ids, tokenizer))"
```

生成的文本是

```
输出文本：
 每一步努力都让你站在工作的惊喜之中，我们中的一员曾与随机-
```

如您所见，生成的文本与我们之前生成的文本非常不同
通过第 5.3 节中的 generate_simple 函数进行翻译（“每一步努力，你都知道，”）
他是其中一条公理……！），这是从训练集中记忆的一段话。

练习 5.2

尝试不同的温度和 top-k 设置。根据你的观察，你能想到哪些需要较低温度和 top-k 设置的应用场景吗？同样，你能想到哪些需要较高温度和 top-k 设置的应用场景吗？（建议在章节结束时，在加载 OpenAI 预训练权重后，也回顾一下这个练习。）

Exercise 5.3

What are the different combinations of settings for the `generate` function to force deterministic behavior, that is, disabling the random sampling such that it always produces the same outputs similar to the `generate_simple` function?

5.4 Loading and saving model weights in PyTorch

Thus far, we have discussed how to numerically evaluate the training progress and pretrain an LLM from scratch. Even though both the LLM and dataset were relatively small, this exercise showed that pretraining LLMs is computationally expensive. Thus, it is important to be able to save the LLM so that we don't have to rerun the training every time we want to use it in a new session.

So, let's discuss how to save and load a pretrained model, as highlighted in figure 5.16. Later, we will load a more capable pretrained GPT model from OpenAI into our `GPTModel` instance.

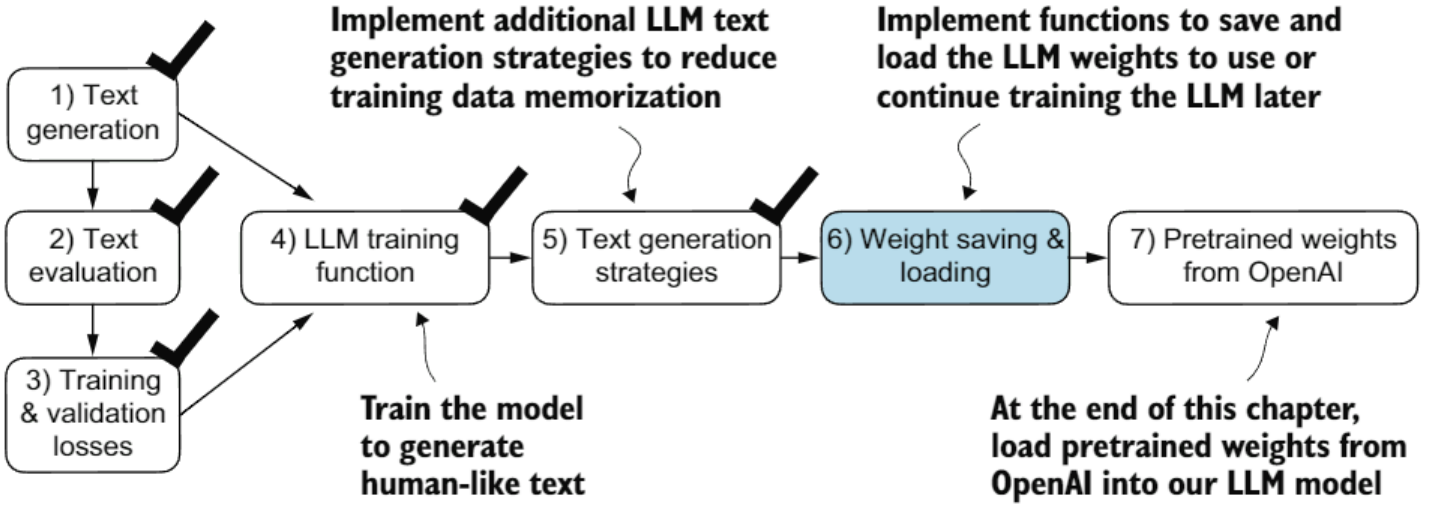

Figure 5.16 After training and inspecting the model, it is often helpful to save the model so that we can use or continue training it later (step 6).

Fortunately, saving a PyTorch model is relatively straightforward. The recommended way is to save a model's `state_dict`, a dictionary mapping each layer to its parameters, using the `torch.save` function:

```
torch.save(model.state_dict(), "model.pth")
```

`"model.pth"` is the filename where the `state_dict` is saved. The `.pth` extension is a convention for PyTorch files, though we could technically use any file extension.

Then, after saving the model weights via the `state_dict`, we can load the model weights into a new `GPTModel` model instance:

练习 5.3

生成函数的不同设置组合，以强制确定性行为，即禁用随机采样，使其始终产生与 generate_simple 函数类似的相同输出？

5.4 加载和保存 PyTorch 中的模型权重

截至目前，我们已讨论了如何数值评估训练进度以及从头开始预训练LLM。尽管LLM和数据集相对较小，这项练习表明预训练LLMs是计算密集型的。因此，能够保存LLM非常重要，这样我们就不必每次在新会话中使用它时都重新运行训练。

因此，让我们讨论如何保存和加载一个预训练模型，如图 5.16 所示。稍后，我们将从 OpenAI 加载一个更强大的预训练 GPT 模型到我们的 GPTModel 实例中。

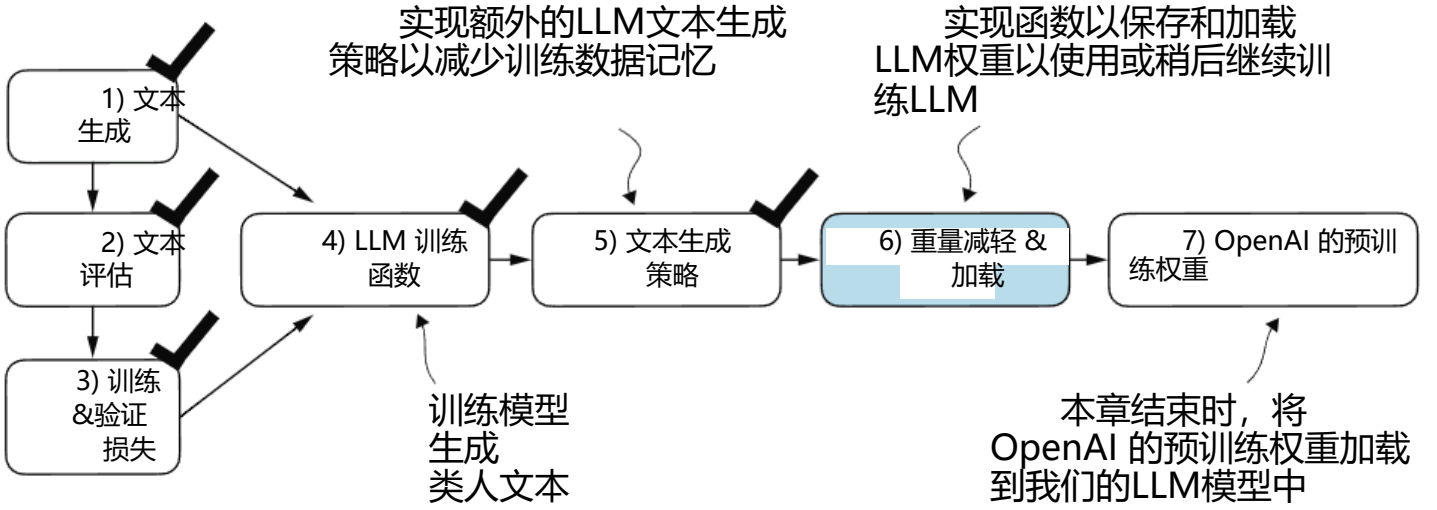

图 5.16 在训练和检查模型后，通常有助于保存模型，以便我们稍后使用或继续训练它（步骤 6）。

幸运的是，保存 PyTorch 模型相对简单。推荐的方法是使用 torch.save 函数保存模型的状态字典，这是一个将每个层映射到其参数的字典：

```
torch.save(model.state_dict(),) 保存模型的状态字典，不进行翻译。
    "model.pth"
```

model.pth 是保存 state_dict 的文件名。.pth 扩展名是 PyTorch 文件的惯例，尽管技术上我们可以使用任何文件扩展名。

然后，通过 state_dict 保存模型权重后，我们可以将模型权重加载到一个新的 GPTModel 模型实例中：

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

As discussed in chapter 4, dropout helps prevent the model from overfitting to the training data by randomly "dropping out" of a layer's neurons during training. However, during inference, we don't want to randomly drop out any of the information the network has learned. Using `model.eval()` switches the model to evaluation mode for inference, disabling the dropout layers of the `model`. If we plan to continue pretraining a model later—for example, using the `train_model_simple` function we defined earlier in this chapter—saving the optimizer state is also recommended.

Adaptive optimizers such as AdamW store additional parameters for each model weight. AdamW uses historical data to adjust learning rates for each model parameter dynamically. Without it, the optimizer resets, and the model may learn suboptimally or even fail to converge properly, which means it will lose the ability to generate coherent text. Using `torch.save`, we can save both the model and optimizer `state_dict` contents:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

Then we can restore the model and optimizer states by first loading the saved data via `torch.load` and then using the `load_state_dict` method:

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Exercise 5.4

After saving the weights, load the model and optimizer in a new Python session or Jupyter notebook file and continue pretraining it for one more epoch using the `train_model_simple` function.

5.5 Loading pretrained weights from OpenAI

Previously, we trained a small GPT-2 model using a limited dataset comprising a short-story book. This approach allowed us to focus on the fundamentals without the need for extensive time and computational resources.

```
    模型 = GPTModel(GPT_CONFIG_124M) 模型.load_state_dict(torch.load("model.pth",
map_location=device)) 模型.eval()
```

如第 4 章所述，dropout 通过在训练过程中随机“丢弃”层中的神经元来帮助防止模型过度拟合训练数据。然而，在推理过程中，我们不想随机丢弃网络学习到的任何信息。使用 model.eval()将模型切换到评估模式进行推理，禁用模型的 dropout 层。如果我们计划稍后继续预训练模型——例如，使用本章前面定义的 train_model_simple 函数——保存优化器状态也是推荐的。

自适应优化器如 AdamW 为每个模型权重存储额外的参数。AdamW 使用历史数据动态调整每个模型参数的学习率。没有它，优化器会重置，模型可能学习不佳甚至无法正确收敛，这意味着它将失去生成连贯文本的能力。使用 torch.save，我们可以保存模型和优化器 state_dict 的内容：

```
    torch.save({})
    "model_state_dict": model.state_dict(),
"optimizer_state_dict": optimizer.state_dict(), },
"model_and_optimizer.pth" )
```

然后我们可以通过首先通过加载保存的数据来恢复模型和优化器状态 torch.load 然后使用 load_state_dict 方法：

```
    checkpoint = 加载模型和优化器.pth 文件，并将其映射到指定设备
model = 创建 GPTModel 实例，配置为 GPT_CONFIG_124M
model.load_state_dict(checkpoint["model_state_dict"]) # 加载模型状态字典
optimizer = 创建 AdamW 优化器，参数学习率为 5e-4，权重衰减为 0.1
optimizer.load_state_dict(checkpoint["optimizer_state_dict"]) # 加载优化器状态字典
model.train() # 将模型设置为训练模式
```

练习 5.4

保存权重后，在新 Python 会话或 Jupyter 笔记本文件中加载模型和优化器，并继续进行一个 epoch 的预训练

训练模型简单函数。

5.5 从 OpenAI 加载预训练权重

之前，我们使用包含短篇小说书籍的有限数据集训练了一个小型 GPT-2 模型。这种方法使我们能够专注于基础，无需大量时间和计算资源。

Fortunately, OpenAI openly shared the weights of their GPT-2 models, thus eliminating the need to invest tens to hundreds of thousands of dollars in retraining the model on a large corpus ourselves. So, let's load these weights into our `GPTModel` class and use the model for text generation. Here, *weights* refer to the weight parameters stored in the `.weight` attributes of PyTorch's `Linear` and `Embedding` layers, for example. We accessed them earlier via `model.parameters()` when training the model. In chapter 6, will reuse these pretrained weights to fine-tune the model for a text classification task and follow instructions similar to ChatGPT.

Note that OpenAI originally saved the GPT-2 weights via TensorFlow, which we have to install to load the weights in Python. The following code will use a progress bar tool called `tqdm` to track the download process, which we also have to install.

You can install these libraries by executing the following command in your terminal:

```
pip install tensorflow>=2.15.0  tqdm>=4.66
```

The download code is relatively long, mostly boilerplate, and not very interesting. Hence, instead of devoting precious space to discussing Python code for fetching files from the internet, we download the `gpt_download.py` Python module directly from this chapter's online repository:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Next, after downloading this file to the local directory of your Python session, you should briefly inspect the contents of this file to ensure that it was saved correctly and contains valid Python code.

We can now import the `download_and_load_gpt2` function from the `gpt_download.py` file as follows, which will load the GPT-2 architecture settings (`settings`) and weight parameters (`params`) into our Python session:

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

Executing this code downloads the following seven files associated with the `124M` parameter GPT-2 model:

```
checkpoint: 100%|███████████████████████████| 77.0/77.0 [00:00<00:00,
                                                          63.9kiB/s]
encoder.json: 100%|████████████████████████| 1.04M/1.04M [00:00<00:00,
                                                           2.20MiB/s]
```

幸运的是，OpenAI 公开分享了他们的 GPT-2 模型的权重，从而消除了我们自己在大型语料库上重新训练模型所需的数十万至数百万美元的投资。因此，让我们将这些权重加载到我们的 GPTModel 类中，并使用该模型进行文本生成。在这里，权重指的是存储在 PyTorch 的 Linear 和 Embedding 层的.weight 属性中的权重参数，例如。我们之前在训练模型时通过 model.parameters()访问了它们。在第 6 章中，我们将重用这些预训练的权重来微调模型以进行文本分类任务，并遵循类似于 ChatGPT 的说明。

请注意，OpenAI 最初通过 TensorFlow 保存了 GPT-2 的权重，我们必须安装 TensorFlow 才能在 Python 中加载权重。以下代码将使用名为 tqdm 的进度条工具来跟踪下载过程，我们同样需要安装它。

您可以通过在终端中执行以下命令来安装这些库：

```
pip install   tensorflow>=2.15.0     tqdm >= 4.66
```

下载代码相对较长，大部分是样板代码，不太有趣。因此，我们不必花费宝贵空间来讨论从互联网上获取文件的 Python 代码，而是直接从本章的在线仓库下载 gpt_download.py Python 模块：

```
    import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/" "LLMs-
from-scratch/main/ch05/" "01_main-chapter-
code/gpt_download.py" ) 文件名 = url.split('/')[-1]
urllib.request.urlretrieve(url, 文件名)
```

接下来，在将此文件下载到您的 Python 会话的本地目录后，您应简要检查此文件的内容，以确保其正确保存且包含有效的 Python 代码。

现在可以从 gpt_download.py 文件中导入 download_and_load_gpt2 函数，如下所示，这将把 GPT-2 架构设置（settings）和权重参数（params）加载到我们的 Python 会话中：

```
    从 gpt_download 导入 download_and_load_gpt2 函数，
settings, params = download_and_load_gpt2()
    model_size="124M", models_dir="gpt2"
```

执行此代码将下载以下七个与该程序相关的文件 124M 参数 GPT-2 模型：

```
检查点          100%|██████████████████████████████████| 77.0/77.0 [00:00<00:00,
                                                    63.9 千字节每秒
encoder.json: 编码器%j|██████████████████████████████| 1.04M/1.04M [00:00<00:00,
                                                    2.20MiB/s
```

```
hprams.json: 100%|████████████████████████| 90.0/90.0 [00:00<00:00,
                                                        78.3kiB/s]
model.ckpt.data-00000-of-00001: 100%|███████| 498M/498M [01:09<00:00,
                                                        7.16MiB/s]
model.ckpt.index: 100%|█████████████████████| 5.21k/5.21k [00:00<00:00,
                                                          3.24MiB/s]
model.ckpt.meta: 100%|██████████████████████| 471k/471k [00:00<00:00,
                                                        2.46MiB/s]
vocab.bpe: 100%|████████████████████████████| 456k/456k [00:00<00:00,
                                                        1.70MiB/s]
```

NOTE If the download code does not work for you, it could be due to intermittent internet connection, server problems, or changes in how OpenAI shares the weights of the open-source GPT-2 model. In this case, please visit this chapter's online code repository at https://github.com/rasbt/LLMs-from-scratch for alternative and updated instructions, and reach out via the Manning Forum for further questions.

Assuming the execution of the previous code has completed, let's inspect the contents of `settings` and `params`:

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

The contents are

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12,
           'n_layer': 12}
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

Both `settings` and `params` are Python dictionaries. The `settings` dictionary stores the LLM architecture settings similarly to our manually defined `GPT_CONFIG_124M` settings. The `params` dictionary contains the actual weight tensors. Note that we only printed the dictionary keys because printing the weight contents would take up too much screen space; however, we can inspect these weight tensors by printing the whole dictionary via `print(params)` or by selecting individual tensors via the respective dictionary keys, for example, the embedding layer weights:

```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].shape)
```

The weights of the token embedding layer are

```
[[-0.11010301 ... -0.1363697   0.01506208   0.04531523]
 [ 0.04034033 ...  0.08605453  0.00253983   0.04318958]
 [-0.12746179  ...  0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ...   0.10435229  0.09783269 -0.06952604]
 [ 0.1860082  ...  -0.09625227  0.07847701 -0.02245961]
```

```
20MiB/s] hprams.json: 100%|████████████████████████████████████████| 90.0/90.0
                                                                    78.3 千字节每秒
model.ckpt.data-00000-of-00001：       100%|████████| 498M/498M [01:09<00:00,
                                                                    7.16MiB/s
model.ckpt.index: 模型100%|t████████████████████████| 5.21k/5.21k [00:00<00:00,
                                                                    3.24MiB/s
model.ckpt.meta：100%|████████████████████████| 471k/471k [00:00<00:00,
                                                                    2.46MiB/s]
vocab.bpe: 词汇100%|████████████████████████| 456k/456k [00:00<00:00,
                                                                    1.70MiB/s
```

注意：如果下载代码对您不起作用，可能是由于间歇性互联网连接、服务器问题或 OpenAI 分享开源 GPT-2 模型权重方式的变化。在这种情况下，请访问此章节的在线代码仓库 https://github.com/rasbt/LLMsfrom-scratch 以获取替代和更新说明，并通过 Manning 论坛寻求进一步问题解答。

假设上一段代码已执行完成，让我们检查内容
设置和参数：

```
打印("设置:", 设置) 打印("参数字典")                    keys: ", params.keys())"
```

内容

```
设置：{'n_vocab': 50257, 'n_ctx': 1024，}          'n_embd':  768,  'n_head': 12,
'n_layer': 12}
参数字典                          keys: 字典键(['blocks', 'b', 'g', 'wpe', 'wte'])
```

设置和参数都是 Python 字典。设置字典存储类似于我们手动定义的 GPT_CONFIG_124M 设置的LLM架构设置。参数字典包含实际的权重张量。请注意，我们只打印了字典键，因为打印权重内容会占用太多的屏幕空间；然而，我们可以通过打印整个字典（print(params)）或通过选择单个张量（例如，嵌入层权重）来检查这些权重张量。

```
打印(params["wte"]) 打印("标记嵌入")                重量    张量    尺寸:",          params["wte"].shape) # 翻译为：
                                                                  params["wte"].形状)
```

令牌嵌入层的权重

```
[[-0.11010301 ... -0.1363697 0.01506208 0.04531523] [0.04034033
... 0.08605453 0.00253983 0.04318958] [-0.12746179 ... 0.08991534
-0.12972379 -0.08785918] ...]
```

```
[-0.04453601 ... 0.10435229 0.09783269 -0.06952604] [0.1860082
... -0.09625227 0.07847701 -0.02245961]
```

```
 [ 0.05135201 ...   0.00704835  0.15519823  0.12067825]]
Token embedding weight tensor dimensions: (50257, 768)
```

We downloaded and loaded the weights of the smallest GPT-2 model via the `download_and_load_gpt2(model_size="124M", ...)` setting. OpenAI also shares the weights of larger models: `355M`, `774M`, and `1558M`. The overall architecture of these differently sized GPT models is the same, as illustrated in figure 5.17, except that different

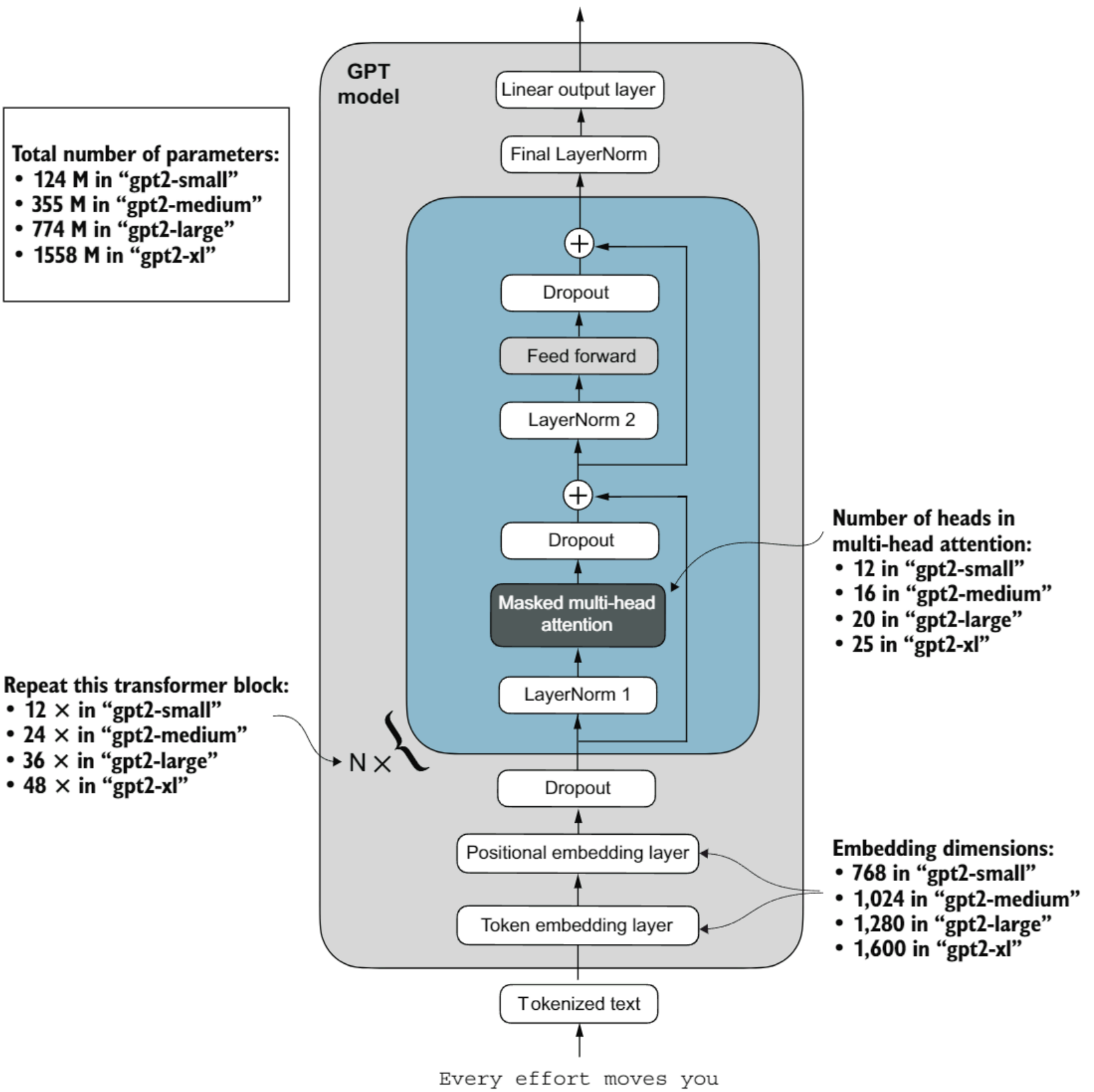

Figure 5.17 GPT-2 LLMs come in several different model sizes, ranging from 124 million to 1,558 million parameters. The core architecture is the same, with the only difference being the embedding sizes and the number of times individual components like the attention heads and transformer blocks are repeated.

[0.05135201 ... 0.00704835 0.15519823 0.12067825] 令牌嵌入权重张量维度：(50257, 768)

我们通过 download_下载并加载了最小 GPT-2 模型的权重 and_load_gpt2（模型大小="124M"，...）设置。OpenAI 还分享了更大模型的权重：355M、774M 和 1558M。这些不同大小的 GPT 模型的总体架构相同，如图 5.17 所示，除了不同的

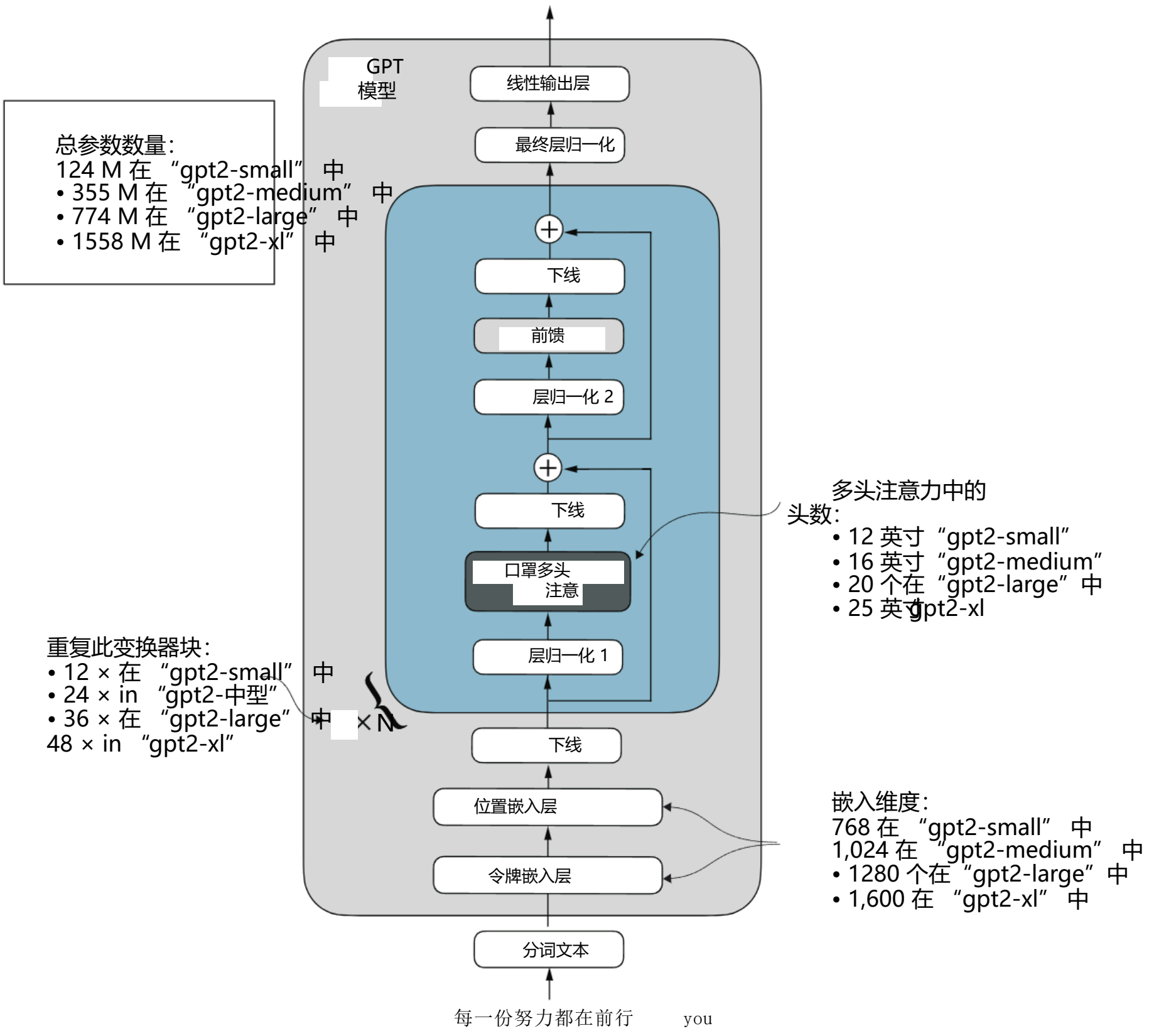

图 5.17 GPT-2 LLMs 有几种不同的模型大小，参数量从 1.24 亿到 15.58 亿不等。核心架构相同，唯一的区别在于嵌入大小以及注意力头和 Transformer 块等个别组件重复的次数。

architectural elements are repeated different numbers of times and the embedding size differs. The remaining code in this chapter is also compatible with these larger models.

After loading the GPT-2 model weights into Python, we still need to transfer them from the `settings` and `params` dictionaries into our `GPTModel` instance. First, we create a dictionary that lists the differences between the different GPT model sizes in figure 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
```

Suppose we are interested in loading the smallest model, `"gpt2-small (124M)"`. We can use the corresponding settings from the `model_configs` table to update our full-length `GPT_CONFIG_124M` we defined and used earlier:

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

Careful readers may remember that we used a 256-token length earlier, but the original GPT-2 models from OpenAI were trained with a 1,024-token length, so we have to update the `NEW_CONFIG` accordingly:

```
NEW_CONFIG.update({"context_length": 1024})
```

Also, OpenAI used bias vectors in the multi-head attention module's linear layers to implement the query, key, and value matrix computations. Bias vectors are not commonly used in LLMs anymore as they don't improve the modeling performance and are thus unnecessary. However, since we are working with pretrained weights, we need to match the settings for consistency and enable these bias vectors:

```
NEW_CONFIG.update({"qkv_bias": True})
```

We can now use the updated `NEW_CONFIG` dictionary to initialize a new `GPTModel` instance:

```
gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

建筑元素重复的次数不同，嵌入大小也不同。本章剩余的代码也与这些更大的模型兼容。

在将 GPT-2 模型权重加载到 Python 后，我们仍需要将它们从设置和 params 字典中转移到我们的 GPTModel 实例中。首先，我们创建一个字典，列出图 5.17 中不同 GPT 模型大小之间的差异。

```
    model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12}, "gpt2-medium
(355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16}, "gpt2-large (774M)":
{"emb_dim": 1280, "n_layers": 36, "n_heads": 20}, "gpt2-xl (1558M)": {"emb_dim": 1600,
"n_layers": 48, "n_heads": 25}, }
```

假设我们感兴趣的是加载最小的模型，“gpt2-small (124M)”。我们可以使用模型_configs 表中对应的设置来更新我们之前定义和使用的全长度 GPT_CONFIG_124M：

```
    model_name = "gpt2-small (124M)" NEW_CONFIG =
GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

仔细的读者可能还记得我们之前使用过 256 个 token 的长度，但 OpenAI 的原始 GPT-2 模型是用 1,024 个 token 的长度进行训练的，因此我们必须相应地更新 NEW_CONFIG：

```
    NEW_CONFIG.update({"context_length": })      1024})
```

此外，OpenAI 在多头注意力模块的线性层中使用了偏置向量来实现查询、键和值矩阵的计算。偏置向量在LLMs中不再常用，因为它们不会提高建模性能，因此是多余的。然而，由于我们正在使用预训练的权重，我们需要匹配设置以保持一致性，并启用这些偏置向量：

```
    NEW_CONFIG.update({"qkv_bias": True})
```

我们现在可以使用更新的 NEW_CONFIG 字典来初始化一个新的 GPTModel 实例：

```
    gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

By default, the `GPTModel` instance is initialized with random weights for pretraining. The last step to using OpenAI's model weights is to override these random weights with the weights we loaded into the `params` dictionary. For this, we will first define a small `assign` utility function that checks whether two tensors or arrays (`left` and `right`) have the same dimensions or shape and returns the right tensor as trainable PyTorch parameters:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Shape mismatch. Left: {left.shape}, "
                         "Right: {right.shape}"
        )
    return torch.nn.Parameter(torch.tensor(right))
```

Next, we define a `load_weights_into_gpt` function that loads the weights from the `params` dictionary into a `GPTModel` instance `gpt`.

Listing 5.5 Loading OpenAI weights into our GPT model code

```
import numpy as np

def load_weights_into_gpt(gpt, params):
    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe'])
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])):
        q_w, k_w, v_w = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)
```

Sets the model's positional and token embedding weights to those specified in params.

The np.split function is used to divide the attention and bias weights into three equal parts for the query, key, and value components.

Iterates over each transformer block in the model

默认情况下，GPTModel 实例使用随机权重进行预训练。使用 OpenAI 模型权重的最后一步是用我们加载到 params 字典中的权重覆盖这些随机权重。为此，我们首先定义一个小的赋值实用函数，该函数检查两个张量或数组（左侧和右侧）是否相等。具有相同的维度或形状，并返回与可训练的 PyTorch 参数相同的张量：

```
def assign(左, 右):
    如果 left.shape 不等于 right.shape:
        抛出    值错误（"形状不匹配。"）
                                {left.shape}, "右:
                                {right.shape}"
    返回 torch.nn.Parameter(torch.tensor(right))
```

接下来，我们定义一个 load_weights_into_gpt 函数，该函数将 params 字典中的权重加载到 GPTModel 实例 gpt 中。

列表 5.5 加载 OpenAI 权重到我们的 GPT 模型代码

```
导入    numpy 作为 np

def 将权重加载到 gpt(gpt, 参数):
    gpt.pos_emb.weight = 分配(gpt.pos_emb.weight, params['wpe'])
gpt.tok_emb.weight = 分配(gpt.tok_emb.weight, params['wte'])

for b 在 range(len(params["blocks"])) 中:
    q_w, k_w, v_w = np.split(
    (params["blocks"][b]["attn"]["c_attn"])["w"], 3, 轴=-1)
gpt.trf_blocks[b].att.W_query.weight = 赋值(
    gpt.trf_blocks[b].att.W_query.weight, q_w.T)
gpt.trf_blocks[b].att.W_key.weight = assign(
    gpt.trf_blocks[b].att.W_key.weight, k_w.T)
gpt.trf_blocks[b].att.W_value.weight = 分配(
        gpt.trf_blocks[b].att.W_value.weight          v_w.T)

    q_b, k_b, v_b = np.split(
    (params["blocks"][b]["attn"]["c_attn"])["b"], 3, 轴=-1)
gpt.trf_blocks[b].att.W_query.bias = 赋值(
    gpt.trf_blocks[b].att.W_query.bias, q_b)
gpt.trf_blocks[b].att.W_key.bias = 分配(
    gpt.trf_blocks[b].att.W_key 偏差, k_b)
gpt.trf_blocks[b].att.W_value 偏差 = 赋值(
        gpt.trf_blocks[b].att.W_value.bias, v_b)

    gpt.trf_blocks[b].att.out_proj.weight = assign(分配)
        gpt.trf_blocks[b].att.out_proj.weight,
     params["blocks"][b]["attn"]["c_proj"]["w"].T)
```

设置模型的定位和标记嵌入权重为 params 中指定的权重。

np.split 函数用于将注意力权重和偏置权重分为三等份，用于查询、键和值组件。

遍历模型中的每个 Transformer 块

```
    gpt.trf_blocks[b].att.out_proj.bias = assign(
        gpt.trf_blocks[b].att.out_proj.bias,
        params["blocks"][b]["attn"]["c_proj"]["b"])

    gpt.trf_blocks[b].ff.layers[0].weight = assign(
        gpt.trf_blocks[b].ff.layers[0].weight,
        params["blocks"][b]["mlp"]["c_fc"]["w"].T)
    gpt.trf_blocks[b].ff.layers[0].bias = assign(
        gpt.trf_blocks[b].ff.layers[0].bias,
        params["blocks"][b]["mlp"]["c_fc"]["b"])
    gpt.trf_blocks[b].ff.layers[2].weight = assign(
        gpt.trf_blocks[b].ff.layers[2].weight,
        params["blocks"][b]["mlp"]["c_proj"]["w"].T)
    gpt.trf_blocks[b].ff.layers[2].bias = assign(
        gpt.trf_blocks[b].ff.layers[2].bias,
        params["blocks"][b]["mlp"]["c_proj"]["b"])

    gpt.trf_blocks[b].norm1.scale = assign(
        gpt.trf_blocks[b].norm1.scale,
        params["blocks"][b]["ln_1"]["g"])
    gpt.trf_blocks[b].norm1.shift = assign(
        gpt.trf_blocks[b].norm1.shift,
        params["blocks"][b]["ln_1"]["b"])
    gpt.trf_blocks[b].norm2.scale = assign(
        gpt.trf_blocks[b].norm2.scale,
        params["blocks"][b]["ln_2"]["g"])
    gpt.trf_blocks[b].norm2.shift = assign(
        gpt.trf_blocks[b].norm2.shift,
        params["blocks"][b]["ln_2"]["b"])

    gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
    gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
    gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"])
```

The original GPT-2 model by OpenAI reused the token embedding weights in the output layer to reduce the total number of parameters, which is a concept known as weight tying.

In the `load_weights_into_gpt` function, we carefully match the weights from OpenAI's implementation with our `GPTModel` implementation. To pick a specific example, OpenAI stored the weight tensor for the output projection layer for the first transformer block as `params["blocks"][0]["attn"]["c_proj"]["w"]`. In our implementation, this weight tensor corresponds to `gpt.trf_blocks[b].att.out_proj.weight`, where `gpt` is a `GPTModel` instance.

Developing the `load_weights_into_gpt` function took a lot of guesswork since OpenAI used a slightly different naming convention from ours. However, the `assign` function would alert us if we try to match two tensors with different dimensions. Also, if we made a mistake in this function, we would notice this, as the resulting GPT model would be unable to produce coherent text.

Let's now try the `load_weights_into_gpt` out in practice and load the OpenAI model weights into our `GPTModel` instance `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

```
    gpt.trf_blocks[b].att.out_proj.bias = assign(
        gpt.trf_blocks[b].att.out_proj.bias,
     params["blocks"][b]["attn"]["c_proj"]["b"]


    gpt.trf_blocks[b].ff.layers[0].weight = 分配(
    gpt.trf_blocks[b].ff.layers[0].权重,
params["blocks"][b]["mlp"]["c_fc"]["w"].转置)
gpt.trf_blocks[b].ff.layers[0].偏置 = assign(

    gpt.trf_blocks[b].ff.layers[0].偏置,
params["blocks"][b]["mlp"]["c_fc"]["b"])
gpt.trf_blocks[b].ff.layers[2].权重 = 赋值(

    gpt.trf_blocks[b].ff.layers[2].权重, params["blocks"]
[b]["mlp"]["c_proj"]["w"].T)
gpt.trf_blocks[b].ff.layers[2].偏置 = 赋值(

        gpt.trf_blocks[b].ff.layers[2].偏置,
     params["blocks"][b]["mlp"]["c_proj"]["b"])


    gpt.trf_blocks[b].norm1.scale = 分配(
    gpt.trf_blocks[b].norm1.缩放,
params["blocks"][b]["ln_1"]["g"])
gpt.trf_blocks[b].norm1.平移 = 分配(
    gpt.trf_blocks[b].norm1.shift,
params["blocks"][b]["ln_1"]["b"])
gpt.trf_blocks[b].norm2.scale = 分配(
    gpt.trf_blocks[b].norm2.scale,
params["blocks"][b]["ln_2"]["g"])
gpt.trf_blocks[b].norm2.shift = 分配(


        gpt.trf_blocks[b].norm2.shift,
     params["blocks"][b]["ln_2"]["b"])


    gpt.final_norm.scale = 赋值(gpt.final_norm.scale, params["g"])
gpt.final_norm.shift = 赋值(gpt.final_norm.shift, params["b"])
gpt.out_head.weight = 赋值(gpt.out_head.weight, params["wte"])
```

OpenAI 的原始 GPT-2 模型在输出层重用了标记嵌入权重以减少参数总数，这被称为权重绑定。

函数，我们仔细地将 OpenAI 实现中的权重与我们的 GPTModel 实现中的权重进行匹配。为了举一个具体的例子，OpenAI 存储了输出投影层的权重张量。

第一个 Transformer 块作为 params["blocks"][0]["attn"]["c_proj"]["w"]。在我们的实现，这个权重张量对应于 gpt.trf_blocks[b].att.out_proj 重量 ，其中 gpt 是一个 GPTModel 实例。

开发 load_weights_into_gpt 函数花费了很多猜测时间，因为 OpenAI 使用的命名约定与我们略有不同。然而，如果尝试匹配两个维度不同的张量，assign 函数会提醒我们。此外，如果我们在这个函数中出错，我们会注意到这一点，因为生成的 GPT 模型将无法生成连贯的文本。

现在让我们在实战中尝试将权重加载到 GPT 中，并将 OpenAI 模型权重加载到我们的 GPTModel 实例 gpt 中：

加载权重到 gpt(gpt, params) gpt 转移到设备(device)

If the model is loaded correctly, we can now use it to generate new text using our previous `generate` function:

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer).to(device),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The resulting text is as follows:

```
Output text:
 Every effort moves you toward finding an ideal new way to practice
     something!
What makes us want to be on top of that?
```

We can be confident that we loaded the model weights correctly because the model can produce coherent text. A tiny mistake in this process would cause the model to fail. In the following chapters, we will work further with this pretrained model and fine-tune it to classify text and follow instructions.

Exercise 5.5

Calculate the training and validation set losses of the `GPTModel` with the pretrained weights from OpenAI on the "The Verdict" dataset.

Exercise 5.6

Experiment with GPT-2 models of different sizes—for example, the largest 1,558 million parameter model—and compare the generated text to the 124 million model.

Summary

- When LLMs generate text, they output one token at a time.
- By default, the next token is generated by converting the model outputs into probability scores and selecting the token from the vocabulary that corresponds to the highest probability score, which is known as "greedy decoding."
- Using probabilistic sampling and temperature scaling, we can influence the diversity and coherence of the generated text.
- Training and validation set losses can be used to gauge the quality of text generated by LLM during training.

如果模型加载正确，我们现在可以使用它通过之前的生成函数生成新的文本：

```
    torch 手动设置随机种子
(123) token_ids = 生成()
    model=gpt, idx=将"Every effort moves you"转换为 token_id 的索引数组
(idx=text_to_token_ids("Every effort moves you", tokenizer).to(device)), 最大新 token 数量
=25, 上下文大小=NEW_CONFIG["context_length"], top_k=50, 温度=1.5) 打印("输出文本:\n", 将
token_id 数组转换为文本(token_ids_to_text(token_ids, tokenizer)))
```

结果文本如下：

```
输出文本：
 每一步努力都让你更接近找到一种理想的新方法来实践
      有些东西!
What  让我们     want  处于...状态那还用说？
```

我们可以确信我们已经正确加载了模型权重，因为模型可以生成连贯的文本。在这个过程中出现微小的错误会导致模型失败。在接下来的章节中，我们将进一步使用这个预训练模型，并对其进行微调以分类文本和遵循指令。

练习 5.5

计算使用 OpenAI 预训练权重的 GPTModel 在“The Verdict”数据集上的训练集和验证集损失。

练习 5.6

尝试不同大小的 GPT-2 模型——例如，最大的 15.58 亿参数模型——并将生成的文本与 1.24 亿参数模型进行比较。

摘要

- 当LLMs生成文本时，它们一次输出一个标记。
- 默认情况下，下一个标记是通过将模型输出转换为概率分数并选择与最高概率分数对应的词汇表中的标记来生成的，这被称为“贪婪解码”。
- 使用概率抽样和温度缩放，我们可以影响生成文本的多样性和连贯性。
- 训练和验证集损失可以用来衡量LLM在训练过程中生成的文本质量。

- Pretraining an LLM involves changing its weights to minimize the training loss.
- The training loop for LLMs itself is a standard procedure in deep learning, using a conventional cross entropy loss and AdamW optimizer.
- Pretraining an LLM on a large text corpus is time- and resource-intensive, so we can load openly available weights as an alternative to pretraining the model on a large dataset ourselves.

- 预训练一个LLM涉及调整其权重以最小化训练损失。
- 训练循环对于LLMs本身是深度学习中的一个标准程序，使用传统的交叉熵损失和 AdamW 优化器。
- 在大型文本语料库上预训练一个LLM既耗时又耗资源，因此我们可以加载公开可用的权重，作为我们自己在大数据集上预训练模型的替代方案。