6 Fine-tuning for classification

This chapter covers

- Introducing different LLM fine-tuning approaches
- Preparing a dataset for text classification
- Modifying a pretrained LLM for fine-tuning
- Fine-tuning an LLM to identify spam messages
- Evaluating the accuracy of a fine-tuned LLM classifier
- Using a fine-tuned LLM to classify new data

So far, we have coded the LLM architecture, pretrained it, and learned how to import pretrained weights from an external source, such as OpenAI, into our model. Now we will reap the fruits of our labor by fine-tuning the LLM on a specific target task, such as classifying text. The concrete example we examine is classifying text messages as "spam" or "not spam." Figure 6.1 highlights the two main ways of fine-tuning an LLM: fine-tuning for classification (step 8) and fine-tuning to follow instructions (step 9).

6 微调用于分类

本章涵盖

- 介绍不同的LLM微调方法
- 准备用于文本分类的数据集
- 修改预训练的LLM以微调
- 微调LLM以识别垃圾邮件
- 评估微调后的LLM分类器的准确性
- 使用微调的LLM来分类新数据

到目前为止，我们已经编码了LLM架构，对其进行了预训练，并学习了如何从外部来源，如 OpenAI，将预训练的权重导入我们的模型。现在，我们将通过在特定目标任务上微调LLM来收获我们的劳动成果，例如对文本进行分类。我们考察的具体例子是将短信消息分类为“垃圾邮件”或“非垃圾邮件”。图 6.1 突出了微调LLM的两种主要方式：用于分类的微调（步骤 8）和用于遵循指令的微调（步骤 9）。

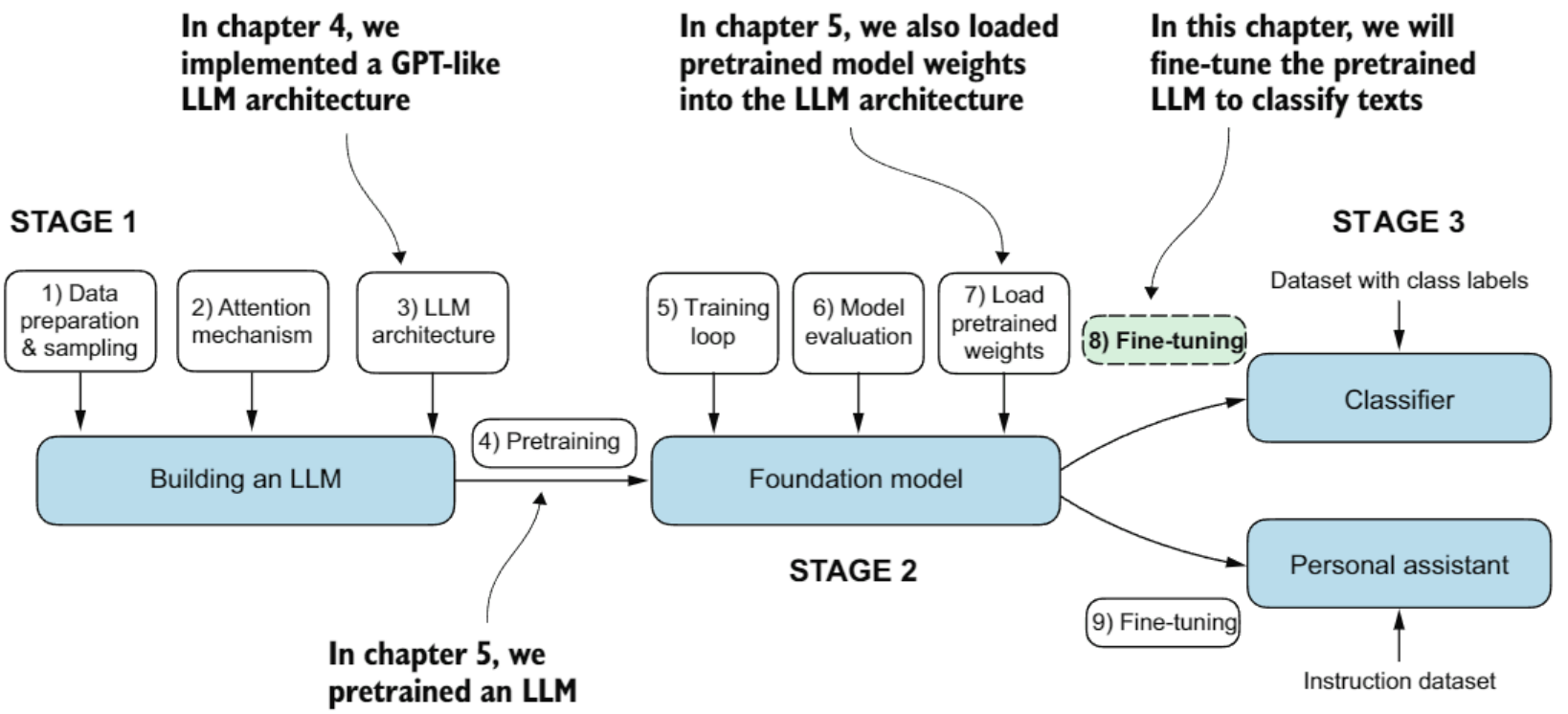


Figure 6.1 The three main stages of coding an LLM. This chapter focus on stage 3 (step 8): fine-tuning a pretrained LLM as a classifier.

6.1 Different categories of fine-tuning

The most common ways to fine-tune language models are *instruction fine-tuning* and *classification fine-tuning*. Instruction fine-tuning involves training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts, as illustrated in figure 6.2.

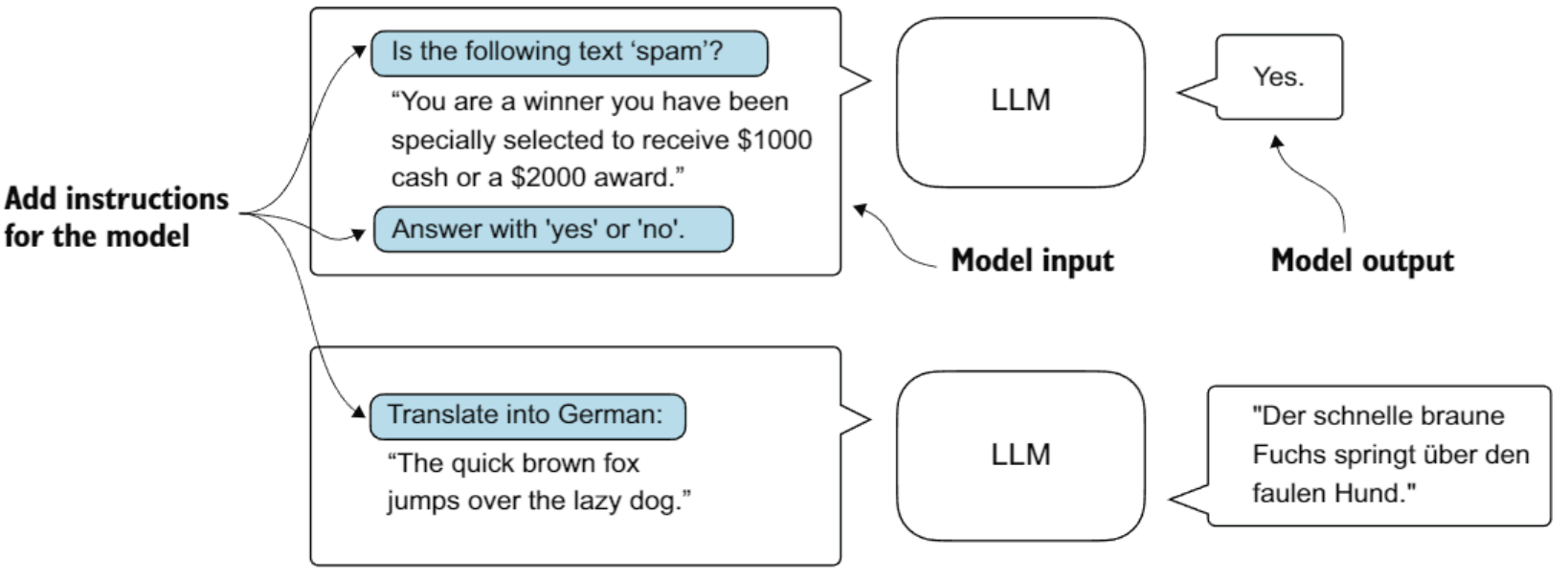


Figure 6.2 Two different instruction fine-tuning scenarios. At the top, the model is tasked with determining whether a given text is spam. At the bottom, the model is given an instruction on how to translate an English sentence into German.

In classification fine-tuning, a concept you might already be acquainted with if you have a background in machine learning, the model is trained to recognize a specific

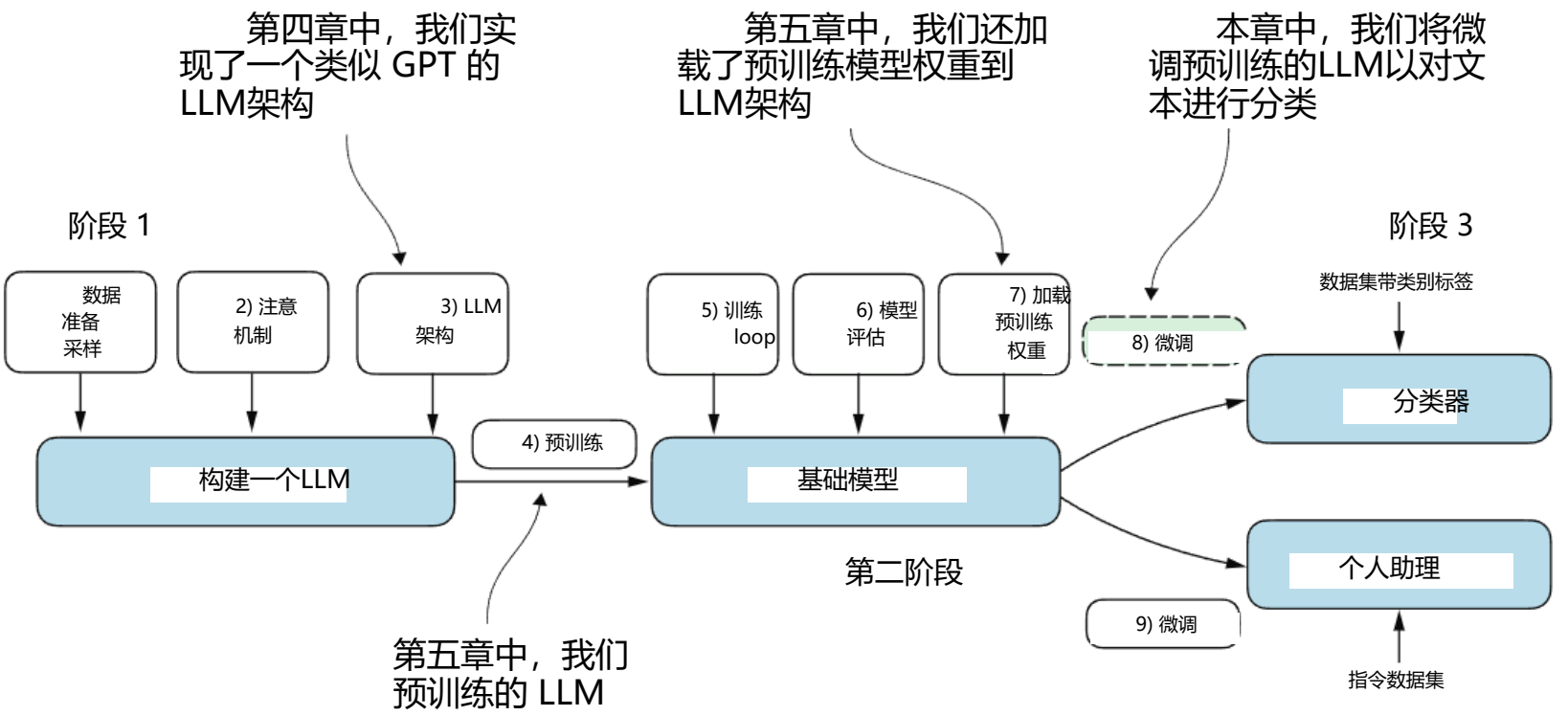


图 6.1 LLM编码的三个主要阶段。本章重点介绍第 3 阶段（步骤 8）：将预训练的LLM微调为分类器。

6.1 不同类别的高精度调整

最常见调整语言模型的方法是指令微调和分类微调。指令微调涉及使用特定指令在一系列任务上训练语言模型，以提高其理解和执行自然语言提示中描述的任务的能力，如图 6.2 所示。

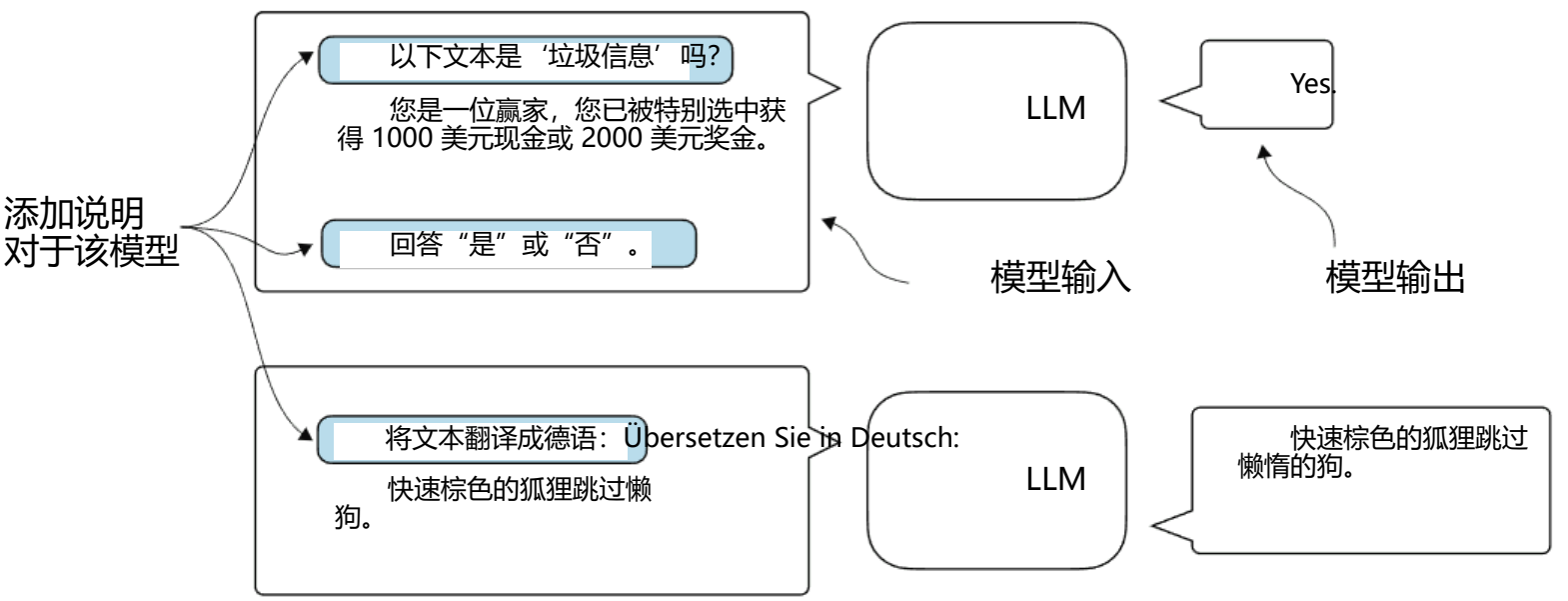


图 6.2 两种不同的指令微调场景。在顶部，模型的任务是确定给定的文本是否为垃圾邮件。在底部，模型被给出如何将英语句子翻译成德语的指令。

在分类微调中，如果你有机器学习的背景，你可能已经熟悉的一个概念，模型被训练来识别特定的

set of class labels, such as "spam" and "not spam." Examples of classification tasks extend beyond LLMs and email filtering: they include identifying different species of plants from images; categorizing news articles into topics like sports, politics, and technology; and distinguishing between benign and malignant tumors in medical imaging.

The key point is that a classification fine-tuned model is restricted to predicting classes it has encountered during its training. For instance, it can determine whether something is "spam" or "not spam," as illustrated in figure 6.3, but it can't say anything else about the input text.

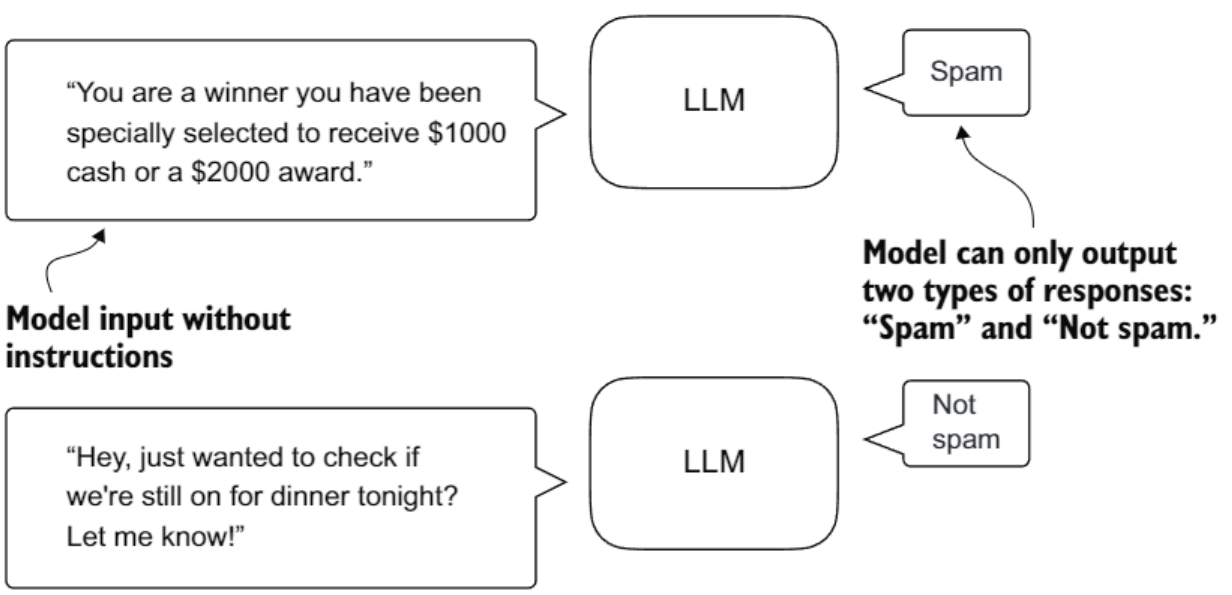


Figure 6.3 A text classification scenario using an LLM. A model fine-tuned for spam classification does not require further instruction alongside the input. In contrast to an instruction fine-tuned model, it can only respond with "spam" or "not spam."

In contrast to the classification fine-tuned model depicted in figure 6.3, an instruction fine-tuned model typically can undertake a broader range of tasks. We can view a classification fine-tuned model as highly specialized, and generally, it is easier to develop a specialized model than a generalist model that works well across various tasks.

Choosing the right approach

Instruction fine-tuning improves a model's ability to understand and generate responses based on specific user instructions. Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. Classification fine-tuning is ideal for projects requiring precise categorization of data into predefined classes, such as sentiment analysis or spam detection.

While instruction fine-tuning is more versatile, it demands larger datasets and greater computational resources to develop models proficient in various tasks. In contrast, classification fine-tuning requires less data and compute power, but its use is confined to the specific classes on which the model has been trained.

类别标签集合，例如“垃圾邮件”和“非垃圾邮件。”分类任务的例子不仅限于LLMs和电子邮件过滤：还包括从图像中识别不同种类的植物；将新闻文章分类到体育、政治和技术等主题；以及在医学影像中区分良性肿瘤和恶性肿瘤。

关键点在于，一个经过微调的分类模型只能预测其在训练过程中遇到的类别。例如，它可以确定某事物是“垃圾邮件”还是“非垃圾邮件”，如图 6.3 所示，但它对输入文本的其他内容无法做出任何说明。

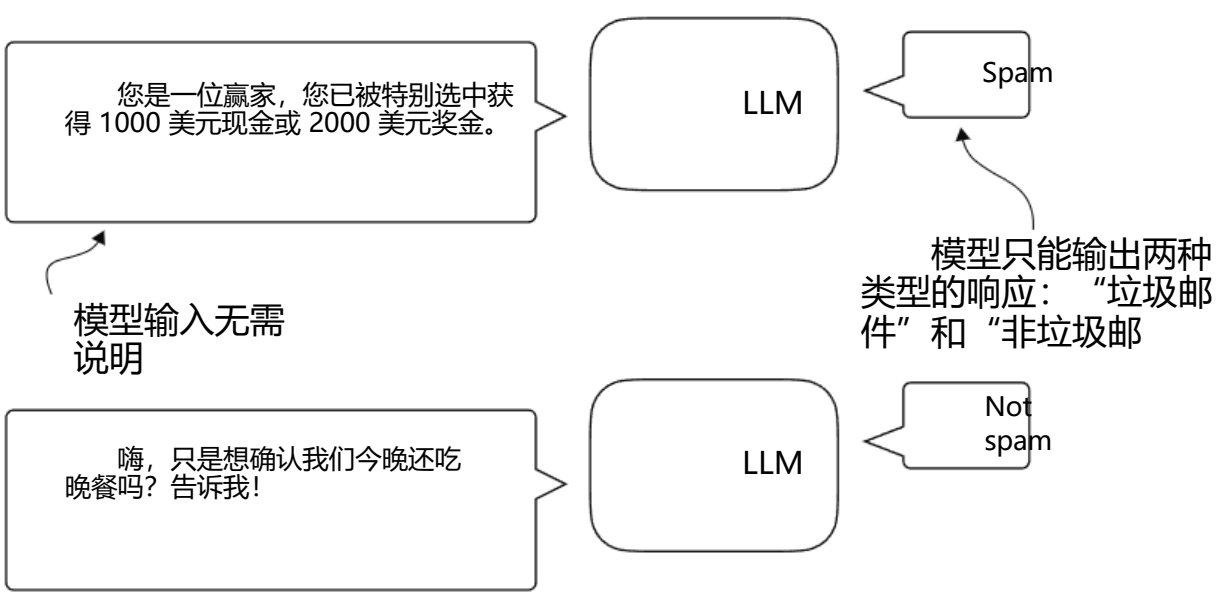


图 6.3 使用LLM的文本分类场景。针对垃圾邮件分类微调的模型不需要进一步指令即可对输入进行分类。与指令微调模型相比，它只能回答“垃圾邮件”或“非垃圾邮件”。

与图 6.3 中描述的分类微调模型相比，指令微调模型通常可以承担更广泛的任务范围。我们可以将分类微调模型视为高度专业化的，通常来说，开发一个适用于各种任务的通用模型比开发一个专业模型更容易。

选择正确的方法

指令微调提高了模型理解和生成基于特定用户指令的响应的能力。指令微调最适合需要根据复杂用户指令处理各种任务的模型，提高灵活性和交互质量。分类微调适用于需要将数据精确分类到预定义类别中的项目，例如情感分析或垃圾邮件检测。

虽然指令微调更加灵活，但它需要更大的数据集和更多的计算资源来开发擅长各种任务的模型。相比之下，分类微调需要的数据和计算能力较少，但其应用范围仅限于模型已训练的特定类别。

6.2 Preparing the dataset

We will modify and classification fine-tune the GPT model we previously implemented and pretrained. We begin by downloading and preparing the dataset, as highlighted in figure 6.4. To provide an intuitive and useful example of classification fine-tuning, we will work with a text message dataset that consists of spam and non-spam messages.

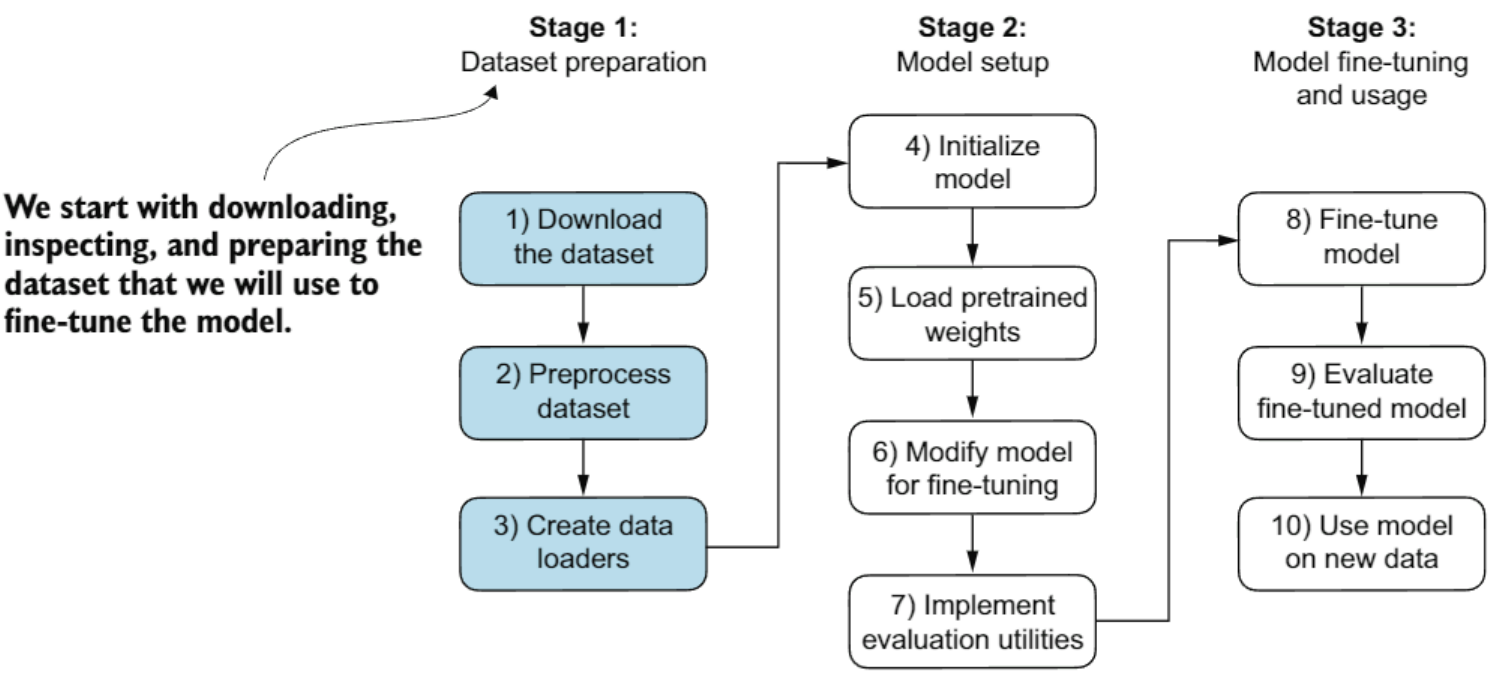


Figure 6.4 The three-stage process for classification fine-tuning an LLM. Stage 1 involves dataset preparation. Stage 2 focuses on model setup. Stage 3 covers fine-tuning and evaluating the model.

> **NOTE** Text messages typically sent via phone, not email. However, the same steps also apply to email classification, and interested readers can find links to email spam classification datasets in appendix B.

The first step is to download the dataset.

Listing 6.1 Downloading and unzipping the dataset

```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"


def download_and_unzip_spam_data(
        url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
```

6.2 准备数据集

我们将修改和分类微调之前实现和预训练的 GPT 模型。我们首先下载并准备数据集，如图 6.4 所示。为了提供一个直观且有用的分类微调示例，我们将使用一个包含垃圾邮件和非垃圾邮件的消息数据集进行工作。

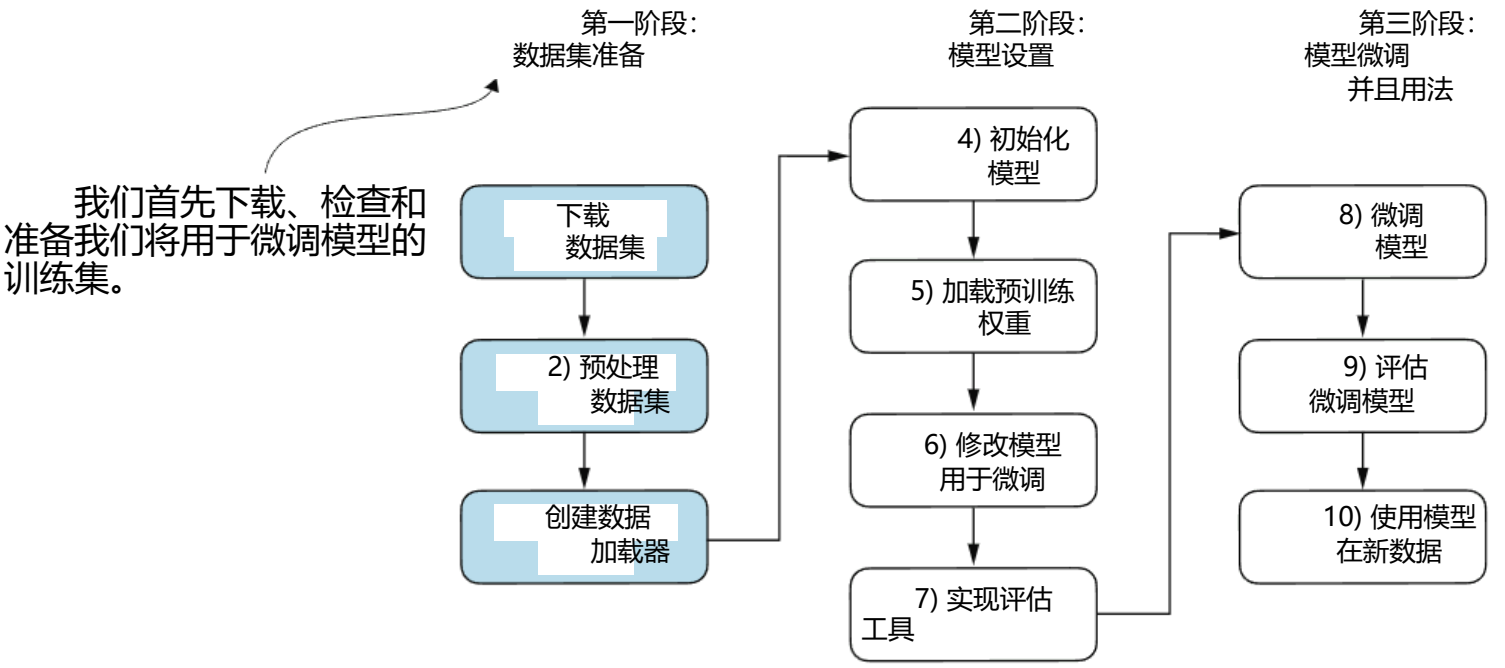


图 6.4 对分类微调的三阶段过程 LLM。第一阶段涉及数据集准备。第二阶段侧重于模型设置。第三阶段涵盖微调和评估模型。

短信通常通过电话发送，而不是电子邮件。然而，相同的步骤也适用于电子邮件分类，感兴趣的读者可以在附录 B 中找到电子邮件垃圾邮件分类数据集的链接。

第一步是下载数据集。

列表 6.1 下载和解压数据集

```
    import urllib.request
import zipfile
import os
from pathlib import Path


    url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip" zip_path =
"sms_spam_collection.zip" extracted_path = "sms_spam_collection" data_file_path =
Path(extracted_path) / "SMSSpamCollection.tsv"




    def  下载并解压垃圾数据
              url, zip 路径, 解压路径, 数据文件路径):
         if  data_file_path.exists(): 文件路径存在检查
```

```
        print(f"{data_file_path} already exists. Skipping download "
              "and extraction."
        )
        return

    with urllib.request.urlopen(url) as response:
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    with zipfile.ZipFile(zip_path, "r") as zip_ref:
        zip_ref.extractall(extracted_path)

    original_file_path = Path(extracted_path) / "SMSSpamCollection"
    os.rename(original_file_path, data_file_path)
    print(f"File downloaded and saved as {data_file_path}")

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)
```

Downloads the file

Unzips the file

Adds a .tsv file extension

After executing the preceding code, the dataset is saved as a tab-separated text file, `SMSSpamCollection.tsv`, in the `sms_spam_collection` folder. We can load it into a pandas `DataFrame` as follows:

```
import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df
```

Renders the data frame in a Jupyter notebook. Alternatively, use print(df).

Figure 6.5 shows the resulting data frame of the spam dataset.

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Figure 6.5 Preview of the `SMSSpamCollection` dataset in a pandas `DataFrame`, showing class labels ("ham" or "spam") and corresponding text messages. The dataset consists of 5,572 rows (text messages and labels).

Let's examine the class label distribution:

```
print(df["Label"].value_counts())
```

```
        该文件路径已存在。跳过下载
               "and  提取。
        )
        返回

    使用 urllib.request.urlopen(url) 作为 response:
        with open(压缩文件路径, "wb") as 输出文件:
            out_file.write(response.read())

    使用 zipfile.ZipFile(zip_path, "r") 作为 zip_ref:
        zip_ref 提取所有文件到 extracted_path

    原始文件路径 = Path(提取路径) / "SMSSpamCollection"
os.rename(original_file_path, data_file_path) 打印(f"文件已下载并保存为
{data_file_path}")

存在(): download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)
```

下载文件

解压文件

添加 .tsv 文件文件扩展名

执行上述代码后，数据集被保存为制表符分隔的文本文件，SMSSpamCollection.tsv，在 sms_spam_collection 文件夹中。我们可以将其加载到 pandas DataFrame 如下：

```
    导入 pandas 库作为
pd, df = pd.read_csv(
    data_file_path, 分隔符="\t", 标题=None, 名称=["标签", "文本"] ) df
```

渲染数据框在 Jupyter 笔记本中。或者使用 print(df)。

图 6.5 显示了垃圾邮件数据集的结果数据框。

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

图 6.5 预览 SMSSpamCollection 数据集 在一个 pandas 数据框中显示类别标签（“ham”或“spam”）和相应的文本消息。数据集包含 5,572 行（文本消息和标签）。

让我们检查类别标签分布：

```
打印(df["标签"].value_counts())
```

Executing the previous code, we find that the data contains "ham" (i.e., not spam) far more frequently than "spam":

```
Label
ham     4825
spam     747
Name: count, dtype: int64
```

For simplicity, and because we prefer a small dataset (which will facilitate faster fine-tuning of the LLM), we choose to undersample the dataset to include 747 instances from each class.

> **NOTE** There are several other methods to handle class imbalances, but these are beyond the scope of this book. Readers interested in exploring methods for dealing with imbalanced data can find additional information in appendix B.

We can use the code in the following listing to undersample and create a balanced dataset.

Listing 6.2 Creating a balanced dataset

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.concat([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

Counts the instances of "spam"

Randomly samples "ham" instances to match the number of "spam" instances

Combines ham subset with "spam"

After executing the previous code to balance the dataset, we can see that we now have equal amounts of spam and non-spam messages:

```
Label
ham     747
spam    747
Name: count, dtype: int64
```

Next, we convert the "string" class labels `"ham"` and `"spam"` into integer class labels 0 and 1, respectively:

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

This process is similar to converting text into token IDs. However, instead of using the GPT vocabulary, which consists of more than 50,000 words, we are dealing with just two token IDs: 0 and 1.

执行上一段代码，我们发现数据中“ham”（即非垃圾邮件）的出现频率远高于“spam”：

```
标签
火腿 4825
垃圾邮件 747
姓名： count, 数据类型：int64
```

为了简化，并且因为我们更喜欢小数据集（这将有助于更快地微调LLM），我们选择对数据集进行下采样，包括每个类别 747 个实例。

注意：处理类别不平衡的方法还有几种，但这些超出了本书的范围。对探索处理不平衡数据方法感兴趣的读者可以在附录 B 中找到更多信息。

我们可以使用以下列表中的代码进行下采样并创建一个平衡的数据集。

列表 6.2 创建一个平衡的数据集

```
def 创建平衡数据集(df):
    num_spam = df[df["标签"] == "垃圾邮件"].shape[0]
ham_subset = df[df["标签"] == "正常邮件"].sample(
    num_spam, 随机状态=123) 平衡_df =
    pd.concat([
        ham_subset, df[df["标签"] == "垃圾邮件"]
    return balanced_df


balanced_df = 创建平衡数据集(df) 打印
(balanced_df["标签"].value_counts())
```

统计“垃圾邮件”的实例数量

随机采样“ham”实例以匹配“spam”实例的数量

结合火腿子集与“垃圾邮件”

执行之前的代码平衡数据集后，我们现在可以看到垃圾邮件和非垃圾邮件的数量相等：

```
标签
火腿 747
垃圾邮件 747
姓名： count, 数据类型：int64
```

接下来，我们将“string”类标签“ham”和“spam”分别转换为整数类标签 0 和 1：

```
平衡后的数据框["标签"]  = balanced_df["Label"].map({"ham": "正常",0,"spam": "垃圾邮件"1})
```

这个过程类似于将文本转换为标记 ID。然而，我们处理的是只有两个标记 ID：0 和 1，而不是使用包含超过 50,000 个单词的 GPT 词汇表。

Next, we create a `random_split` function to split the dataset into three parts: 70% for training, 10% for validation, and 20% for testing. (These ratios are common in machine learning to train, adjust, and evaluate models.)

Listing 6.3 Splitting the dataset

```
def random_split(df, train_frac, validation_frac):

    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1)
```

Shuffles the entire DataFrame

Calculates split indices

Splits the DataFrame

Test size is implied to be 0.2 as the remainder.

Let's save the dataset as CSV (comma-separated value) files so we can reuse it later:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Thus far, we have downloaded the dataset, balanced it, and split it into training and evaluation subsets. Now we will set up the PyTorch data loaders that will be used to train the model.

6.3 Creating data loaders

We will develop PyTorch data loaders conceptually similar to those we implemented while working with text data. Previously, we utilized a sliding window technique to generate uniformly sized text chunks, which we then grouped into batches for more efficient model training. Each chunk functioned as an individual training instance. However, we are now working with a spam dataset that contains text messages of varying lengths. To batch these messages as we did with the text chunks, we have two primary options:

- Truncate all messages to the length of the shortest message in the dataset or batch.
- Pad all messages to the length of the longest message in the dataset or batch.

The first option is computationally cheaper, but it may result in significant information loss if shorter messages are much smaller than the average or longest messages,

接下来，我们创建一个 random_split 函数来将数据集分为三部分：70%用于训练，10%用于验证，20%用于测试。（这些比例在机器学习中很常见，用于训练、调整和评估模型。）

列表 6.3 分割数据集

```
def 随机分割(df,          train_frac, validation_frac): 训练分数，验证分数
    df = df.sample()
    frac=1, 随机状态=123 ).重置索引(删除
=True) 训练结束 = int(len(df) * 训练比例) 验
证结束 = 训练结束
                                    + int(数据帧长度)* 验证分数

    train_df = df[:训练结束] validation_df = df[训练结
束:验证结束] test_df = df[验证结束:]

    返回 train_df,       validation_df,    测试_df

train_df, validation_df, test_df = 随机分割(
    平衡的_df       0.7, 0.1)
```

洗牌整个数据框

计算分割索引

将 DataFrame 分割

测试大小隐含为剩余的 0.2。

让我们将数据集保存为 CSV（逗号分隔值）文件，以便以后可以重用：

```
train_df 保存为"train.csv"，不包含索引 validation_df 保存
为"validation.csv"，不包含索引 test_df 保存为"test.csv"，不包
含索引
```

截至目前，我们已经下载了数据集，进行了平衡，并将其分为训练集和评估集。现在我们将设置用于训练模型的 PyTorch 数据加载器。

6.3 创建数据加载器

我们将开发与我们在处理文本数据时实现的类似概念的 PyTorch 数据加载器。之前，我们使用了滑动窗口技术来生成均匀大小的文本块，然后将这些块分组到批次中以提高模型训练效率。每个块作为一个独立的训练实例。然而，我们现在正在处理一个包含不同长度文本消息的垃圾邮件数据集。为了将这些消息批量处理，就像我们处理文本块一样，我们有两种主要选择：

- 截断所有消息至数据集或批次中最短消息的长度。
- 将所有消息填充到数据集或批次中最长消息的长度。

第一个选项计算成本更低，但如果短消息远小于平均或最长消息，可能会导致信息损失较大

potentially reducing model performance. So, we opt for the second option, which preserves the entire content of all messages.

To implement batching, where all messages are padded to the length of the longest message in the dataset, we add padding tokens to all shorter messages. For this purpose, we use `"<|endoftext|>"` as a padding token.

However, instead of appending the string `"<|endoftext|>"` to each of the text messages directly, we can add the token ID corresponding to `"<|endoftext|>"` to the encoded text messages, as illustrated in figure 6.6. `50256` is the token ID of the padding token `"<|endoftext|>"`. We can double-check whether the token ID is correct by encoding the `"<|endoftext|>"` using the *GPT-2 tokenizer* from the `tiktoken` package that we used previously:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

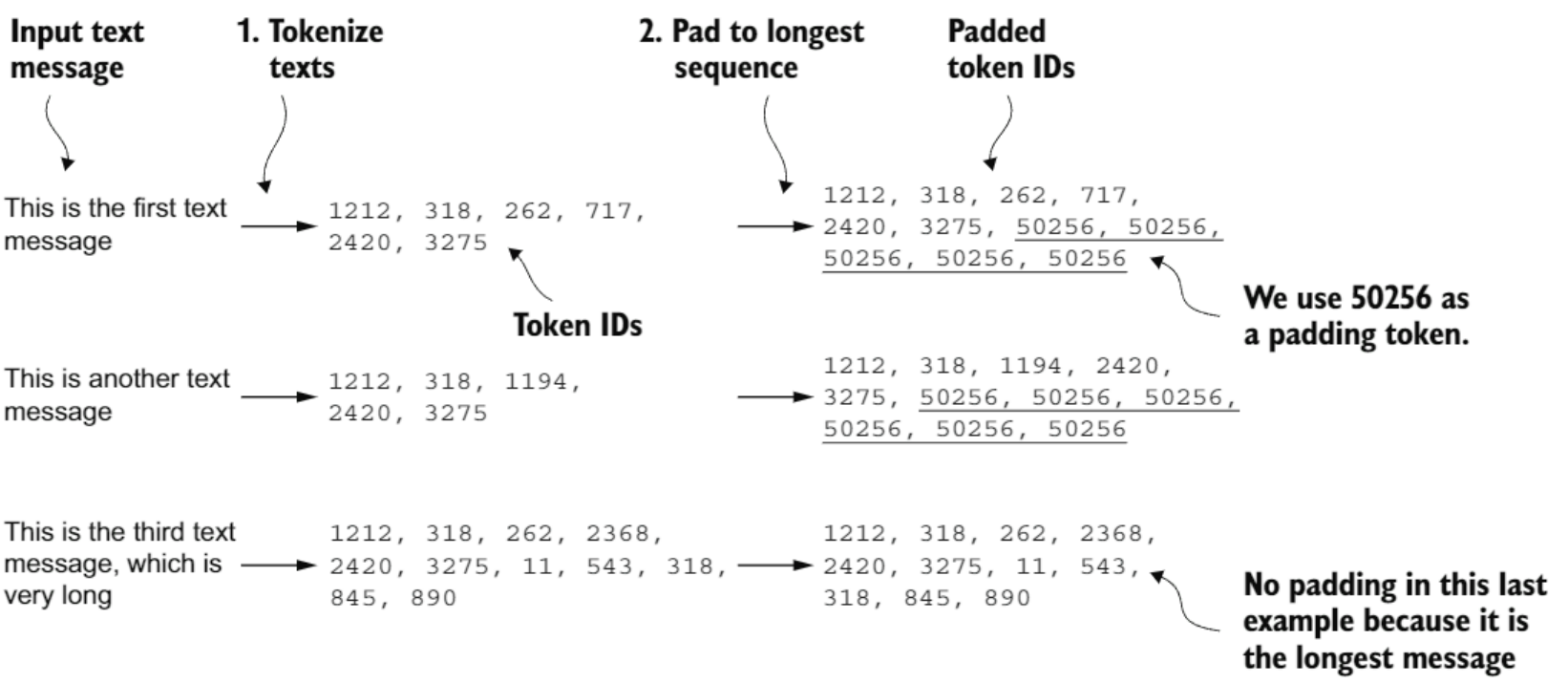


Figure 6.6 The input text preparation process. First, each input text message is converted into a sequence of token IDs. Then, to ensure uniform sequence lengths, shorter sequences are padded with a padding token (in this case, token ID 50256) to match the length of the longest sequence.

Indeed, executing the preceding code returns `[50256]`.

We first need to implement a PyTorch `Dataset`, which specifies how the data is loaded and processed before we can instantiate the data loaders. For this purpose, we define the `SpamDataset` class, which implements the concepts in figure 6.6. This `SpamDataset` class handles several key tasks: it identifies the longest sequence in the training dataset, encodes the text messages, and ensures that all other sequences are padded with a *padding token* to match the length of the longest sequence.

可能降低模型性能。因此，我们选择第二个选项，保留所有消息的全部内容。

为了实现批处理，将所有消息填充到数据集中最长消息的长度，我们在所有较短的消息中添加填充标记。为此，我们使用"[[pad]]"作为填充标记。

然而，我们不是直接将字符串"}}"附加到每条文本消息上，而是可以将对应于"}}"的标记 ID 添加到编码后的文本消息中，如图 6.6 所示。50256 是填充标记"}}"的标记 ID。我们可以通过使用之前使用的 tiktoken 包中的 GPT-2 标记器对"}}"进行编码来双重检查标记 ID 是否正确：

```
    import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("",
                                                    允许的特殊字符={}))
```

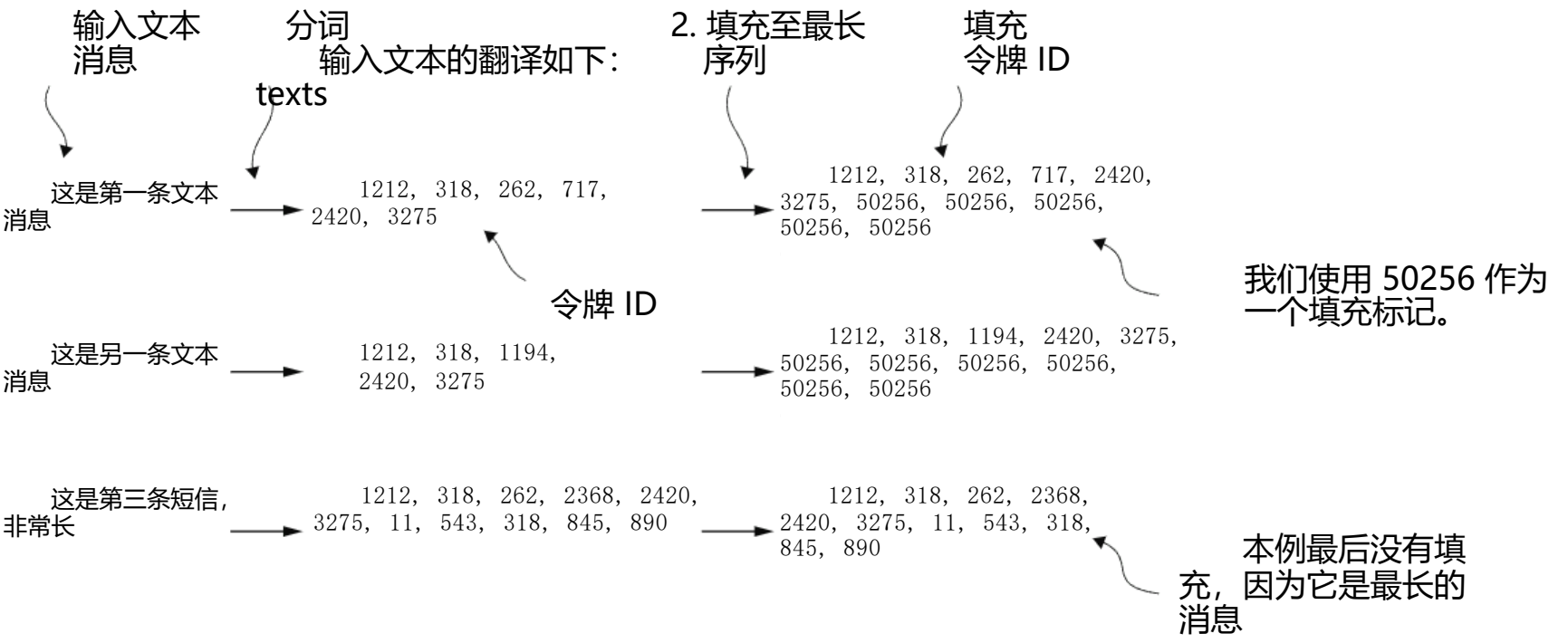


图 6.6 输入文本准备过程。首先，将每个输入文本消息转换为一系列标记 ID。然后，为了确保序列长度一致，将较短的序列用填充标记（在这种情况下，标记 ID 50256）填充，以匹配最长序列的长度。

确实，执行前面的代码返回 [50256]。

首先需要实现一个 PyTorch 数据集，它指定了在实例化数据加载器之前如何加载数据和处理数据。为此，我们定义了 SpamDataset 类，该类实现了图 6.6 中的概念。这个 SpamDataset 类处理几个关键任务：它识别训练数据集中的最长序列，对文本消息进行编码，并确保所有其他序列都使用填充标记进行填充，以匹配最长序列的长度。

Listing 6.4 Setting up a Pytorch `Dataset` class

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

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                 pad_token_id=50256):
        self.data = pd.read_csv(csv_file)

        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length

            self.encoded_texts = [
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]

        self.encoded_texts = [
            encoded_text + [pad_token_id] *
            (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]


    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
        return len(self.data)

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length
```

Pretokenizes texts

Truncates sequences if they are longer than max_length

Pads sequences to the longest sequence

列表 6.4 设置 Pytorch 数据集类

```
import torch
from  torch.utils.data 导入 Dataset


class SpamDataset(数据集):
     def __init__(self, csv_file, tokenizer, 最大长度=None,
          pad_token_id=50256):
      self.data =
      pd.read_csv(csv_file)
                                                  <-- 预处理文本
          self.encoded_texts = [
          tokenizer.encode(text) for text in self.data["Text"] ]
      翻译为：tokenizer.encode（text）for text in self.data["Text"] ]


          如果 max_length 是 None：
          self.max_length = self._longest_encoded_length() else:


               self.max_length     最大长度

                                                  <-- 截断长度超过
                                                      max_length 的序列
       最大长度
               self.encoded_texts = [
               encoded_text[:self.max_length] for
      encoded_text in self.encoded_texts ]
      翻译：encoded_text[:self.max_length] 对于
      self.encoded_texts 中的 encoded_text ]


                                                  <-- 将序列填充至最
                                                      长序列长度
          self.encoded_texts = [
          encoded_text + [填充令牌 ID] *
     (self.max_length - len(encoded_text)) for
      encoded_text in self.encoded_texts ]



def __getitem__(self, 索引):
     encoded = self.encoded_texts[index] 标签 =
 self.data.iloc[index]["Label"] 返回 (


     torch.tensor(encoded, dtype=torch.long),
 torch.tensor(label, dtype=torch.long) ) ->
 torch.tensor(编码, dtype=torch.long),
 torch.tensor(标签, dtype=torch.long) )

def __len__(self): # 获取长度
     返回 self.data 的长度


def _最长编码长度(self):
     最大长度 = 0
     for encoded_text in self.encoded_texts:
          encoded_length = len(encoded_text) 如
      果 encoded_length > max_length:
     max_length = 编码长度 return
 max_length
```

The `SpamDataset` class loads data from the CSV files we created earlier, tokenizes the text using the GPT-2 tokenizer from `tiktoken`, and allows us to *pad* or *truncate* the sequences to a uniform length determined by either the longest sequence or a predefined maximum length. This ensures each input tensor is of the same size, which is necessary to create the batches in the training data loader we implement next:

```
train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)
```

The longest sequence length is stored in the dataset's `max_length` attribute. If you are curious to see the number of tokens in the longest sequence, you can use the following code:

```
print(train_dataset.max_length)
```

The code outputs `120`, showing that the longest sequence contains no more than 120 tokens, a common length for text messages. The model can handle sequences of up to 1,024 tokens, given its context length limit. If your dataset includes longer texts, you can pass `max_length=1024` when creating the training dataset in the preceding code to ensure that the data does not exceed the model's supported input (context) length.

Next, we pad the validation and test sets to match the length of the longest training sequence. Importantly, any validation and test set samples exceeding the length of the longest training example are truncated using `encoded_text[:self.max_length]` in the `SpamDataset` code we defined earlier. This truncation is optional; you can set `max_length=None` for both validation and test sets, provided there are no sequences exceeding 1,024 tokens in these sets:

```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
```

SpamDataset 类从我们之前创建的 CSV 文件中加载数据，使用 tiktoken 中的 GPT-2 分词器对文本进行分词，并允许我们将序列填充或截断到由最长序列或预定义的最大长度确定的统一长度。这确保了每个输入张量的大小相同，这对于创建我们接下来实现的训练数据加载器中的批次是必要的：

```
    train_dataset = SpamDataset(
    csv_file="train.csv",
max_length=None,
tokenizer=tokenizer)
```

数据集中最长序列长度存储在 max_length 属性中。如果您想查看最长序列中的标记数量，可以使用以下代码：

```
    打印(train_dataset 的最大长度)
```

代码输出 120，表示最长序列不超过 120 个标记，这是短信的常见长度。模型可以处理长达 1,024 个标记的序列，考虑到其上下文长度限制。如果你的数据集包含更长的文本，可以在创建训练数据集时将 max_length=1024 传递给前述代码，以确保数据不超过模型支持的输入（上下文）长度。

接下来，我们将验证集和测试集填充到最长训练序列的长度。重要的是，任何超过最长训练示例长度的验证集和测试集样本都会使用我们之前定义的 SpamDataset 代码中的 encoded_text[:self.max_length] 进行截断。这种截断是可选的；如果这两个集合中没有超过 1,024 个标记的序列，你可以将 max_length 设置为 None：

```
    val_dataset = SpamDataset(
    csv_file="validation.csv",
max_length=train_dataset.max_length,
tokenizer=tokenizer) test_dataset = SpamDataset(
    csv_file="test.csv",
max_length=train_dataset.max_length,
tokenizer=tokenizer)
```

Exercise 6.1 Increasing the context length

Pad the inputs to the maximum number of tokens the model supports and observe how it affects the predictive performance.

Using the datasets as inputs, we can instantiate the data loaders similarly to when we were working with text data. However, in this case, the targets represent class labels rather than the next tokens in the text. For instance, if we choose a batch size of 8, each batch will consist of eight training examples of length 120 and the corresponding class label of each example, as illustrated in figure 6.7.

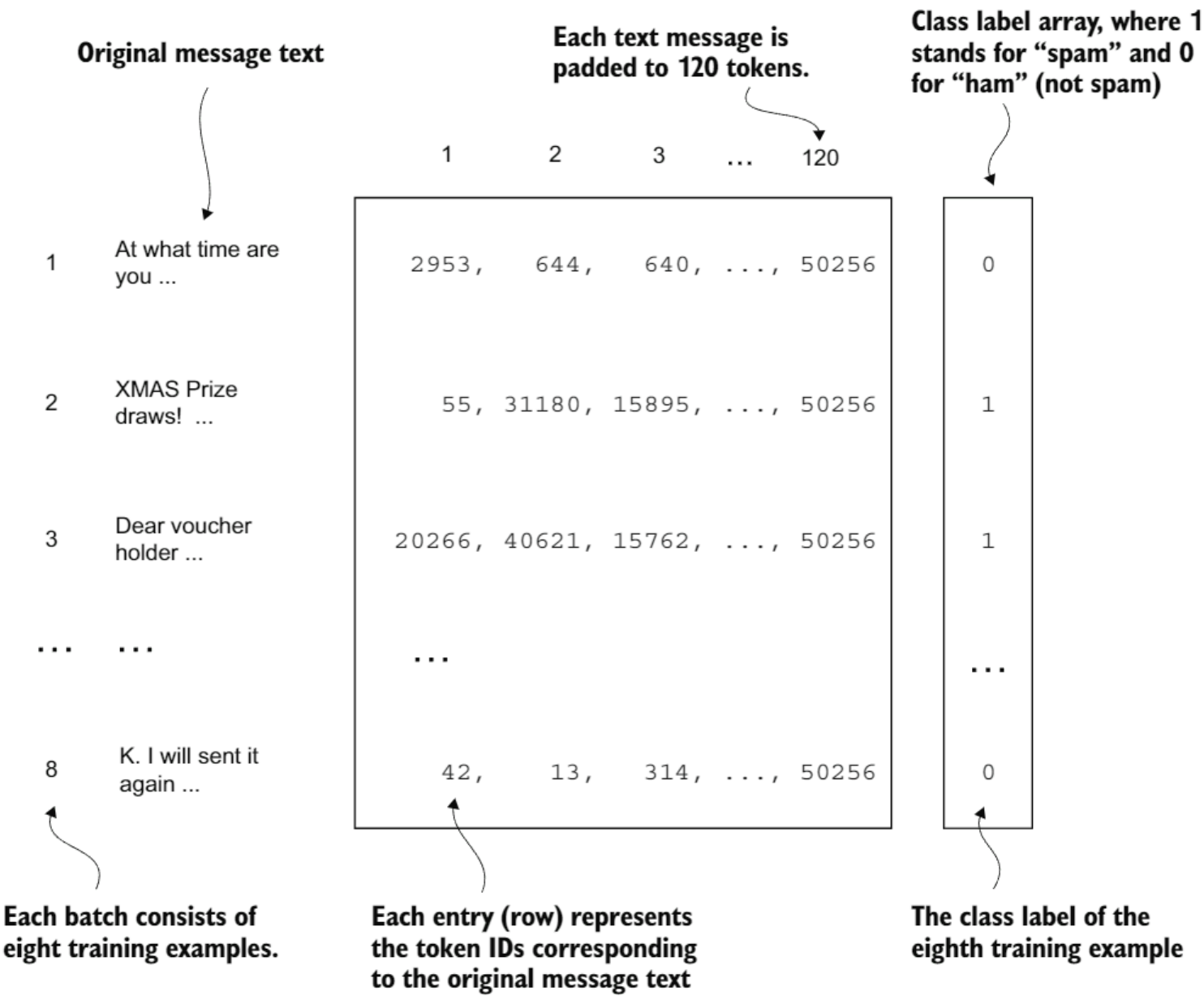


Figure 6.7 A single training batch consisting of eight text messages represented as token IDs. Each text message consists of 120 token IDs. A class label array stores the eight class labels corresponding to the text messages, which can be either `0` ("not spam") or `1` ("spam").

练习 6.1 增加上下文长度

填充输入以符合模型支持的最大令牌数，并观察这对预测性能的影响。

使用数据集作为输入，我们可以像处理文本数据时一样实例化数据加载器。然而，在这种情况下，目标表示的是类别标签，而不是文本中的下一个标记。例如，如果我们选择批大小为 8，每个批次将包含 8 个长度为 120 的训练示例及其对应的类别标签，如图 6.7 所示。

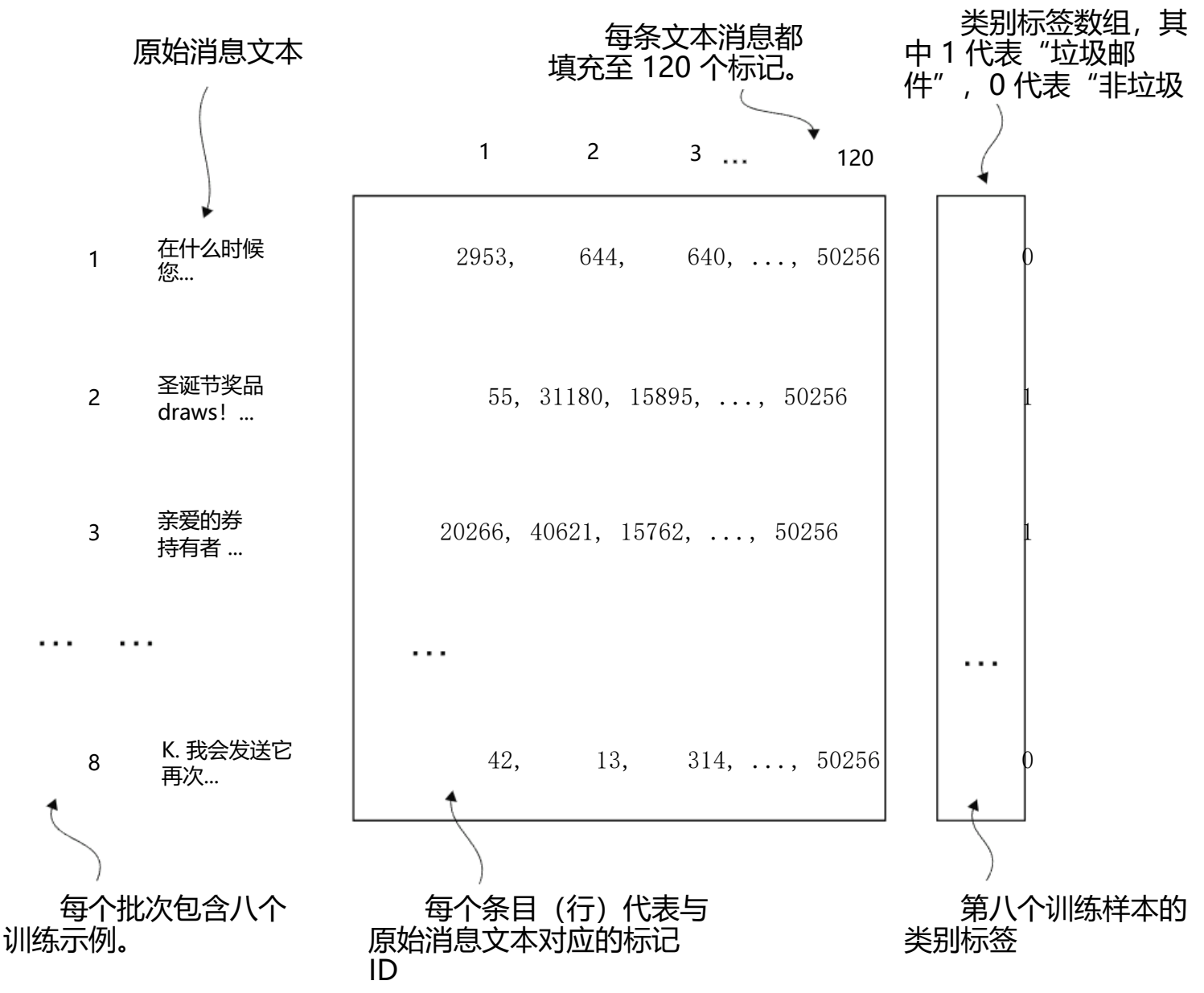


图 6.7 一个包含八个文本消息的训练批次，这些文本消息以标记 ID 表示。每个文本消息由 120 个标记 ID 组成。一个类别标签数组存储与文本消息对应的八个类别标签，可以是 0（“非垃圾邮件”）或 1（“垃圾邮件”）。

The code in the following listing creates the training, validation, and test set data loaders that load the text messages and labels in batches of size 8.

Listing 6.5 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8
torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)
val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
```

This setting ensures compatibility with most computers.

```
    num_workers=num_workers,
    drop_last=False,
)
```

To ensure that the data loaders are working and are, indeed, returning batches of the expected size, we iterate over the training loader and then print the tensor dimensions of the last batch:

```
for input_batch, target_batch in train_loader:
    pass
print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

The output is

```
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

As we can see, the input batches consist of eight training examples with 120 tokens each, as expected. The label tensor stores the class labels corresponding to the eight training examples.

Lastly, to get an idea of the dataset size, let's print the total number of batches in each dataset:

以下代码创建了训练、验证和测试数据加载器，这些加载器以 8 个批次的容量加载文本消息和标签。

列表 6.5 创建 PyTorch 数据加载器

```
    from torch.utils.data 导入 DataLoader

    num_workers = 0
    batch_size = 8
torch.manual_seed(123)

    train_loader = DataLoader(
        dataset=train_dataset,
     批大小=batch_size, 打乱顺序
     =True,
    num_workers=num_workers,
drop_last=True, ) val_loader =
DataLoader(
    dataset=val_dataset,
batch_size=batch_size,
num_workers=num_workers,
drop_last=False, ) 测试加载器 =
DataLoader(
    dataset=test_dataset, 批大小
=batch_size, 工作进程数
=num_workers, 不丢弃最后=False, )
```

此设置确保与大多数计算机兼容。

为确保数据加载器正在运行并且确实返回了预期大小的批次，我们遍历训练加载器，然后打印最后一个批次的张量维度：

```
    for 输入批次，目标批次 in 训练加载器：
    通过 print("输入批次维度:", input_batch.shape) print("标签
批次维度", target_batch.shape)
```

输出结果

```
    输入批次维度：torch.Size([8, 120]) 标签批次维度
torch.Size([8])
```

如您所见，输入批次由八个训练示例组成，每个示例有 120 个标记，正如预期的那样。标签张量存储了与八个训练示例对应的类别标签。

最后，为了了解数据集的大小，让我们打印出每个数据集中的批次数总量：

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

The number of batches in each dataset are

```
130 training batches
19 validation batches
38 test batches
```

Now that we've prepared the data, we need to prepare the model for fine-tuning.

6.4 Initializing a model with pretrained weights

We must prepare the model for classification fine-tuning to identify spam messages. We start by initializing our pretrained model, as highlighted in figure 6.8.

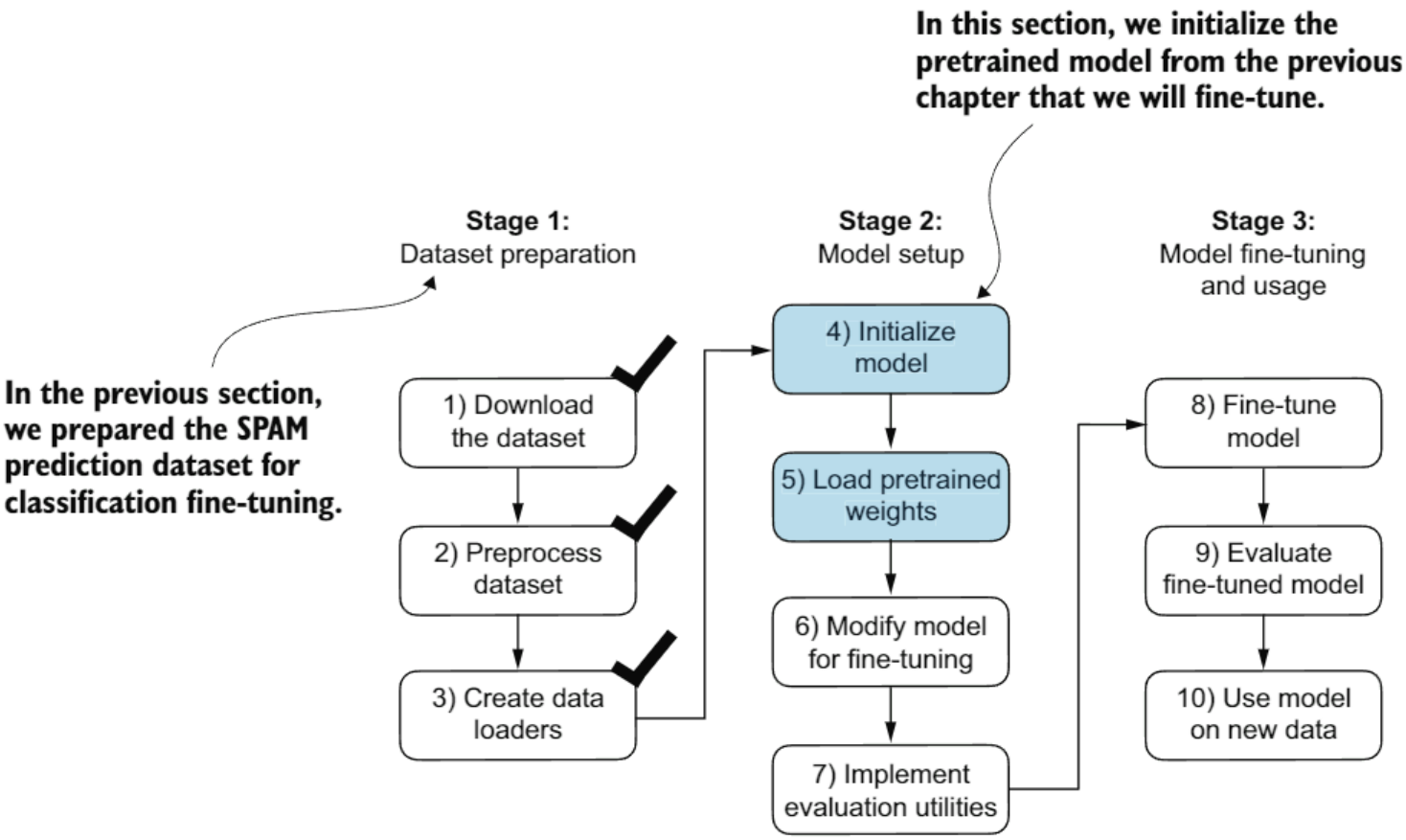


Figure 6.8 The three-stage process for classification fine-tuning the LLM. Having completed stage 1, preparing the dataset, we now must initialize the LLM, which we will then fine-tune to classify spam messages.

To begin the model preparation process, we employ the same configurations we used to pretrain unlabeled data:

```
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
```

```
    打印(f"{len(train_loader)} 训练批次") 打印(f"
{len(val_loader)} 验证批次") 打印(f"{len(test_loader)}
测试批次")
```

每个数据集中的批次数量

```
130 训练批次
19 验证批次
38 测试批次
```

现在我们已经准备好了数据，我们需要为微调准备模型。

6.4 初始化一个带有预训练权重的模型

我们必须为分类微调准备模型，以识别垃圾邮件。

我们首先初始化我们的预训练模型，如图 6.8 所示。

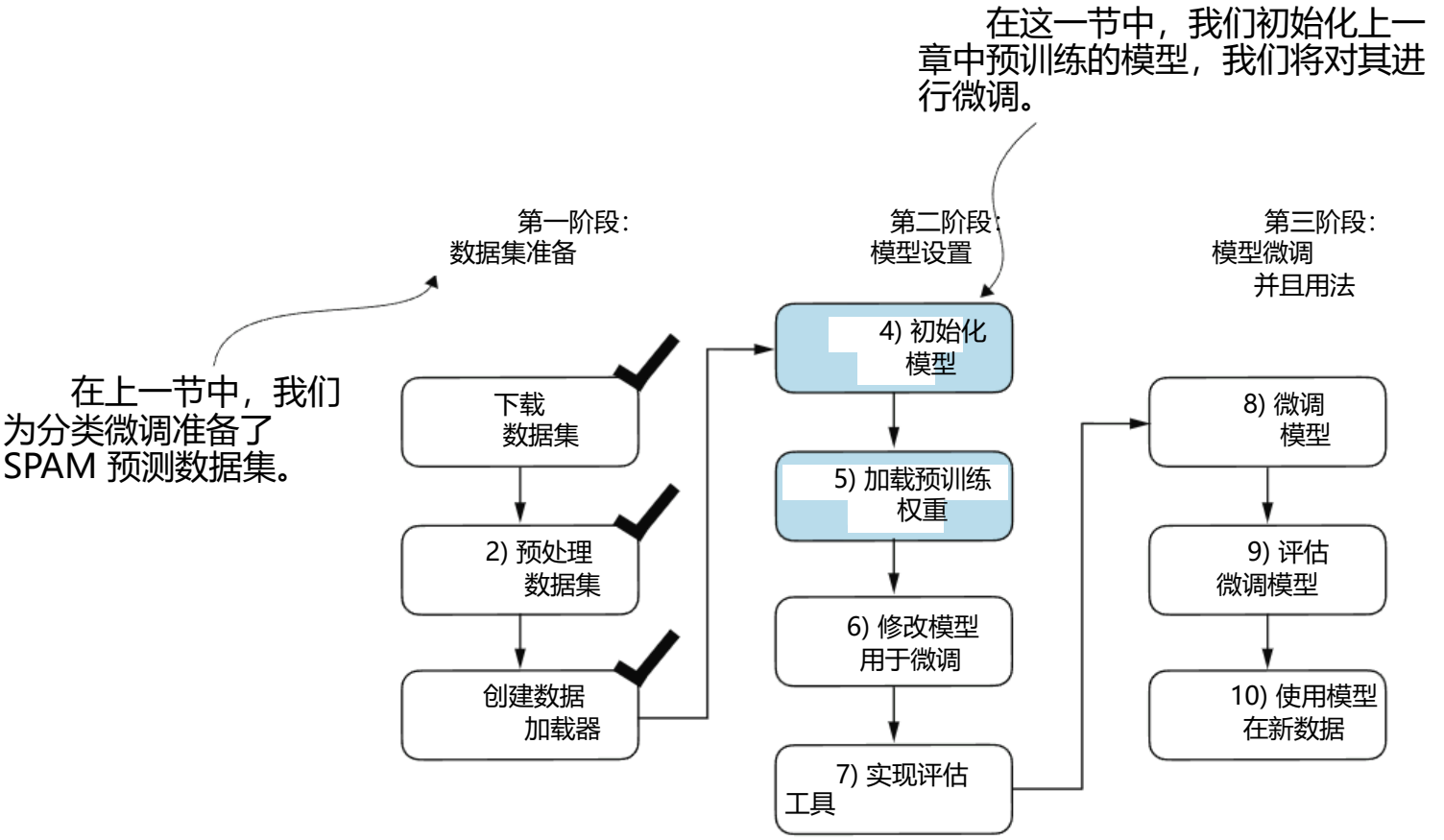


图 6.8 对分类微调的三阶段过程。完成第一阶段，准备数据集后，我们现在必须初始化LLM，然后对其进行微调以分类垃圾邮件。

开始模型准备过程时，我们使用与预训练未标记数据相同的配置：

```
    选择模型 = "gpt2-small (124M)" 输入提示
= "每一步努力"
```

```
BASE_CONFIG = {
    "vocab_size": 50257,          # Vocabulary size
    "context_length": 1024,       # Context length
    "drop_rate": 0.0,             # Dropout rate
    "qkv_bias": True              # Query-key-value bias
}
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},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])
```

Next, we import the `download_and_load_gpt2` function from the `gpt_download.py` file and reuse the `GPTModel` class and `load_weights_into_gpt` function from pretraining (see chapter 5) to load the downloaded weights into the GPT model.

Listing 6.6 Loading a pretrained GPT model

```
from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()
```

After loading the model weights into the `GPTModel`, we reuse the text generation utility function from chapters 4 and 5 to ensure that the model generates coherent text:

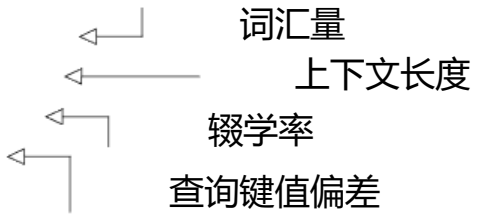
```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

The following output shows the model generates coherent text, which is indicates that the model weights have been loaded correctly:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

```
    BASE_CONFIG = {
    "vocab_size": 50257,
"context_length": 1024,
"drop_rate": 0.0, "qkv_bias": 真
确 } 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}, } BASE_CONFIG.update(model_configs[CHOOSE_MODEL])
```

接下来，我们从 gpt_download.py 导入 download_and_load_gpt2 函数

文件并重用 GPTModel 类和 load_weights_into_gpt 函数从预训练中（见第 5 章）将下载的权重加载到 GPT 模型中。

列表 6.6 加载预训练的 GPT 模型

```
    从 gpt_download 导入 download_and_load_gpt2 从
chapter05 导入 GPTModel,                    加载权重到 GPT

    模型大小 = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")") 设置，参数
= download_and_load_gpt2(
    model_size=model_size, 模型目录="gpt2" )

    模型 = GPTModel(BASE_CONFIG) 将权重加载
到 gpt(model, params) 模型评估()
```

在将模型权重加载到 GPTModel 后，我们重用第 4 章和第 5 章中的文本生成实用函数，以确保模型生成连贯的文本：

```
    从第四章导入 generate_text_simple 函数，从第五章导入
text_to_token_ids 和 token_ids_to_text 函数

    每个努力都让你前进

    model=model, idx=将 text_1 转换为 token_ids(text_1,
tokenizer), 最大新 token 数=15, 上下文大小
=BASE_CONFIG["context_length"] ) 打印(token_ids 转换为
text(token_ids, tokenizer))
```

以下输出显示模型生成了连贯的文本，这表明模型权重已正确加载：

```
    每一步努力都让你前进。第一步是
                              理解          重要性              您的作品
```

Before we start fine-tuning the model as a spam classifier, let's see whether the model already classifies spam messages by prompting it with instructions:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

The model output is

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
you have been specially selected to receive $1000 cash
or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
```

Based on the output, it's apparent that the model is struggling to follow instructions. This result is expected, as it has only undergone pretraining and lacks instruction fine-tuning. So, let's prepare the model for classification fine-tuning.

6.5 Adding a classification head

We must modify the pretrained LLM to prepare it for classification fine-tuning. To do so, we replace the original output layer, which maps the hidden representation to a vocabulary of 50,257, with a smaller output layer that maps to two classes: `0` ("not spam") and `1` ("spam"), as shown in figure 6.9. We use the same model as before, except we replace the output layer.

Output layer nodes

We could technically use a single output node since we are dealing with a binary classification task. However, it would require modifying the loss function, as I discuss in "Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch" (https://mng.bz/NRZ2). Therefore, we choose a more general approach, where the number of output nodes matches the number of classes. For example, for a three-class problem, such as classifying news articles as "Technology," "Sports," or "Politics," we would use three output nodes, and so forth.

在我们开始将模型微调为垃圾邮件分类器之前，让我们通过提示指令来看看模型是否已经能够对垃圾邮件进行分类：

```
    text_2 = (
    以下文本是否为“垃圾邮件”？请回答“是”或“否”：’您是赢家，您已被特别
选中，可领取 1000 美元现金或 2000 美元奖金。’) token_ids =
generate_text_simple(


    model=model, idx=将 text_2 转换为 token_ids(text_2,
tokenizer), 最大新 token 数=23, 上下文大小
=BASE_CONFIG["context_length"] ) 打印(token_ids 转换为
text(token_ids, tokenizer))
```

模型输出是

```
    以下文本是垃圾邮件吗？回答“是”或“否”：’您是幸运儿，您被特别选中获得 1000 美元现金
或 2000 美元奖金。’以下文本是垃圾邮件吗？回答“是”或“否”：’您是幸运儿’
```

基于输出结果，很明显模型在遵循指令方面存在困难。这个结果是预期的，因为它只经过了预训练，缺乏指令微调。因此，让我们为模型准备分类微调。

6.5 添加分类头

我们必须修改预训练的LLM以准备进行分类微调。为此，我们用映射到两个类别（0：“非垃圾邮件”和 1：“垃圾邮件”）的较小输出层替换了原始输出层，该层将隐藏表示映射到 50,257 个词汇，如图 6.9 所示。我们使用与之前相同的模型，只是替换了输出层。

输出层节点

技术上我们可以使用单个输出节点，因为我们处理的是二元分类任务。然而，这需要修改损失函数，正如我在“损失函数学习——在 PyTorch 中优化负对数似然和交叉熵”一文中讨论的那样（https://mng.bz/NRZ2）。因此，我们选择了一种更通用的方法，其中输出节点的数量与类别的数量相匹配。例如，对于一个有三个类别的分类问题，比如将新闻文章分类为“科技”、“体育”或“政治”，我们会使用三个输出节点，依此类推。

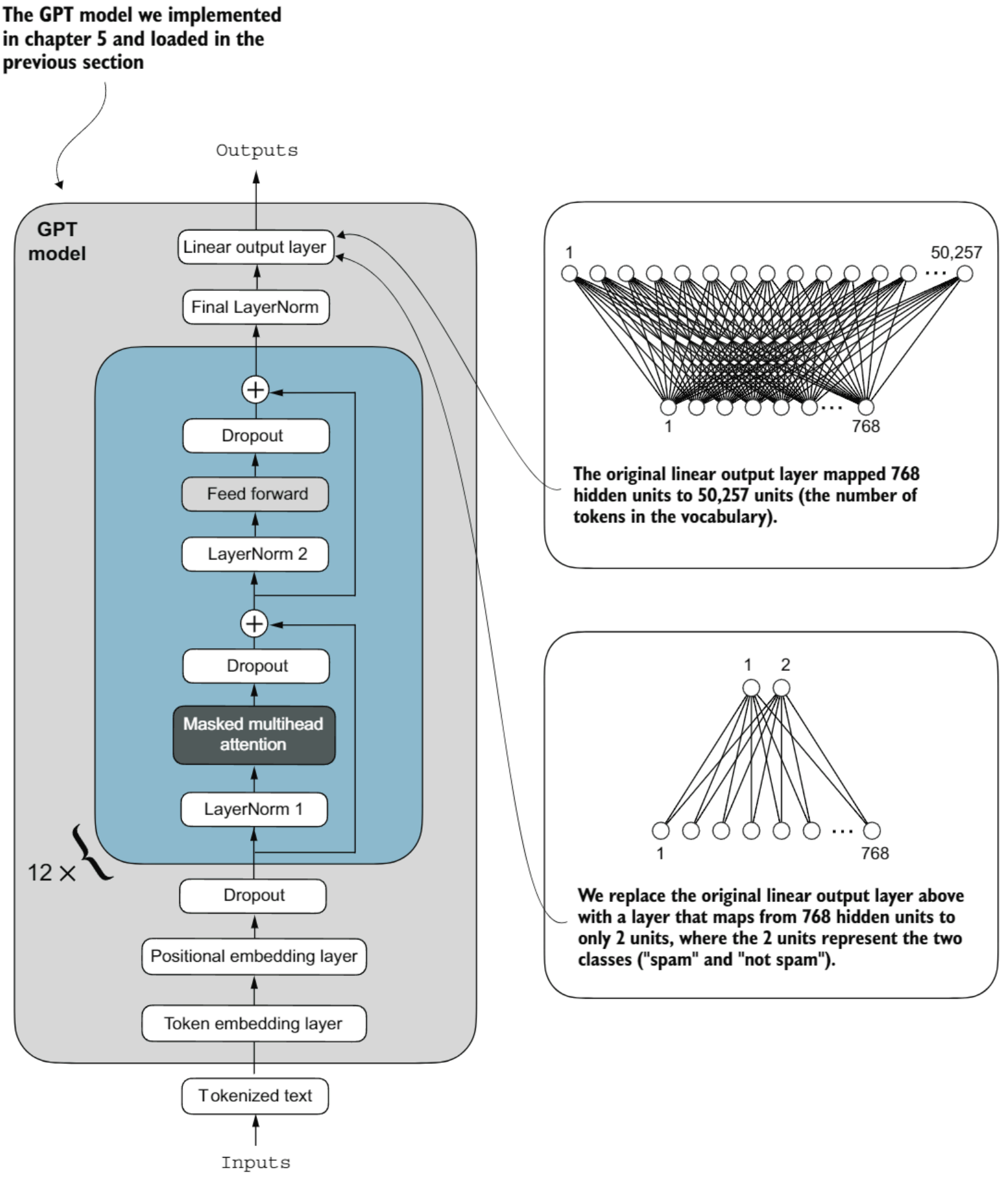


Figure 6.9 Adapting a GPT model for spam classification by altering its architecture. Initially, the model's linear output layer mapped 768 hidden units to a vocabulary of 50,257 tokens. To detect spam, we replace this layer with a new output layer that maps the same 768 hidden units to just two classes, representing "spam" and "not spam."

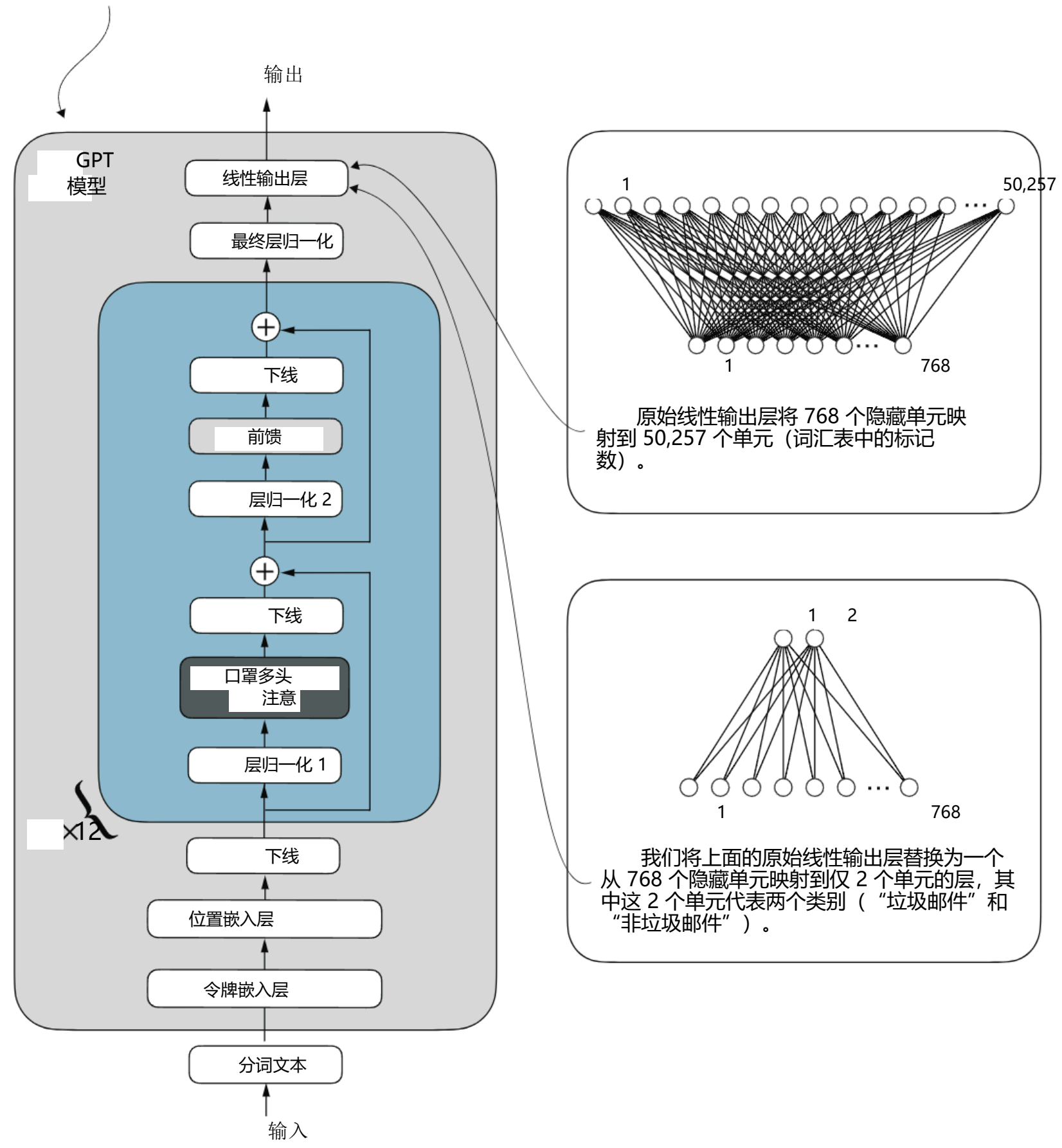


图 6.9 通过改变其架构来调整 GPT 模型以进行垃圾邮件分类。最初，该模型的线性输出层将 768 个隐藏单元映射到 50,257 个标记的词汇表。为了检测垃圾邮件，我们用一个新的输出层替换了这个层，该层将相同的 768 个隐藏单元映射到仅两个类别，代表“垃圾邮件”和“非垃圾邮件”。

Before we attempt the modification shown in figure 6.9, let's print the model architecture via `print(model)`:

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): Linear(in_features=768, out_features=768, bias=True)
        (W_key): Linear(in_features=768, out_features=768, bias=True)
        (W_value): Linear(in_features=768, out_features=768, bias=True)
        (out_proj): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU()
          (2): Linear(in_features=3072, out_features=768, bias=True)
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
  )
  (final_norm): LayerNorm()
  (out_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

This output neatly lays out the architecture we laid out in chapter 4. As previously discussed, the `GPTModel` consists of embedding layers followed by 12 identical *transformer blocks* (only the last block is shown for brevity), followed by a final `LayerNorm` and the output layer, `out_head`.

Next, we replace the `out_head` with a new output layer (see figure 6.9) that we will fine-tune.

Fine-tuning selected layers vs. all layers

Since we start with a pretrained model, it's not necessary to fine-tune all model layers. In neural network-based language models, the lower layers generally capture basic language structures and semantics applicable across a wide range of tasks and datasets. So, fine-tuning only the last layers (i.e., layers near the output), which are more specific to nuanced linguistic patterns and task-specific features, is often sufficient to adapt the model to new tasks. A nice side effect is that it is computationally more efficient to fine-tune only a small number of layers. Interested readers can find more information, including experiments, on which layers to fine-tune in appendix B.

在我们尝试修改图 6.9 所示的修改之前，让我们先打印出模型架构 true via 打印（模型）：

```
    GPT 模型（（tok_emb）：嵌入层（50257，
768）（pos_emb）：嵌入层（1024，768））


      (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(

  (丢弃嵌入): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (11): TransformerBlock( (att):
    多头注意力(

        (W_query): 线性层（输入特征=768，输出特征=768，偏置=True）(W_key): 线性层
        （输入特征=768，输出特征=768，偏置=True）(W_value): 线性层（输入特征=768，输出特
        征=768，偏置=True）(out_proj): 线性层（输入特征=768，输出特征=768，偏置=True）
        (dropout): Dropout（概率=0.0，就地=False））(ff): FeedForward（（layers）：
        Sequential（




        (0): 线性层(in_features=768, out_features=3072, bias=True) (1): GELU()
        (2): 线性层(in_features=3072, out_features=768, bias=True) ) ) (norm1): 层
        归一化() (norm2): 层归一化()




    (dropout_resid): Dropout(p=0.0, inplace=False) ) ) (final_norm): 层归一化()
(out_head): 线性(in_features=768, out_features=50257, bias=False) )
```

本输出清晰地展示了我们在第 4 章中阐述的架构。如前所述，GPTModel 由嵌入层组成，随后是 12 个相同的 transformer 块（为了简洁，仅展示了最后一个块），然后是最终的 LayerNorm 层和输出层，即 out_head。

接下来，我们将 out_head 替换为一个新的输出层（见图 6.9），我们将对其进行微调。

微调选定层与所有层

由于我们从预训练模型开始，因此不需要微调所有模型层。在基于神经网络的语料库模型中，底层通常捕获适用于广泛任务和数据集的基本语言结构和语义。因此，仅微调最后一层（即靠近输出的层），这些层更具体于细微的语言模式和特定任务的特性，通常足以适应新任务。一个很好的副作用是，仅微调少量层在计算上更高效。感兴趣的读者可以在附录 B 中找到更多关于要微调哪些层的信息，包括实验。

To get the model ready for classification fine-tuning, we first *freeze* the model, meaning that we make all layers nontrainable:

```
for param in model.parameters():
    param.requires_grad = False
```

Then, we replace the output layer (`model.out_head`), which originally maps the layer inputs to 50,257 dimensions, the size of the vocabulary (see figure 6.9).

Listing 6.7 Adding a classification layer

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

To keep the code more general, we use `BASE_CONFIG["emb_dim"]`, which is equal to 768 in the `"gpt2-small (124M)"` model. Thus, we can also use the same code to work with the larger GPT-2 model variants.

This new `model.out_head` output layer has its `requires_grad` attribute set to `True` by default, which means that it's the only layer in the model that will be updated during training. Technically, training the output layer we just added is sufficient. However, as I found in experiments, fine-tuning additional layers can noticeably improve the predictive performance of the model. (For more details, refer to appendix B.) We also configure the last transformer block and the final `LayerNorm` module, which connects this block to the output layer, to be trainable, as depicted in figure 6.10.

To make the final `LayerNorm` and last transformer block trainable, we set their respective `requires_grad` to `True`:

```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Exercise 6.2 Fine-tuning the whole model

Instead of fine-tuning just the final transformer block, fine-tune the entire model and assess the effect on predictive performance.

Even though we added a new output layer and marked certain layers as trainable or nontrainable, we can still use this model similarly to how we have previously. For

为了将模型准备好进行分类微调，我们首先冻结模型，这意味着我们将所有层设置为不可训练：

```
for param in 模型参数():
    param.requires_grad = False
参数.需要梯度 = False
```

然后，我们替换输出层（model.out_head），它原本将层输入映射到 50,257 维，即词汇表的大小（见图 6.9）。

列表 6.7 添加分类层

```
torch.manual_seed(123) num_classes =
2 model.out_head = torch.nn.Linear(


in_features=BASE_CONFIG["emb_dim"],
out_features=类别数)
```

为了使代码更加通用，我们使用 BASE_CONFIG["emb_dim"]，在"gpt2-small (124M)"模型中等于 768。因此，我们也可以使用相同的代码来处理更大的 GPT-2 模型变体。

此新模型.out_head 输出层其 requires_grad 属性被设置为

默认为 True，这意味着它是模型中唯一会在训练期间更新的层。从技术上讲，训练我们刚刚添加的输出层就足够了。然而，正如我在实验中发现的那样，微调额外的层可以显著提高模型的预测性能。（更多详情请参阅附录 B。）我们还配置了最后一个 transformer 块和最终的 LayerNorm 模块，该模块将此块连接到输出层，如图 6.10 所示，使其可训练。

为了使最终的 LayerNorm 和最后一个 transformer 块可训练，我们将它们的相应的 requires_grad 设置为 True：

```
for param in 模型.trf_blocks[-1].参数():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

参数.需要梯度 = True

参数.需要梯度 = True

练习 6.2 微调整个模型

而不是仅微调最终的 Transformer 块，微调整个模型并评估其对预测性能的影响。

尽管我们添加了一个新的输出层并将某些层标记为可训练或不可训练，我们仍然可以像以前一样使用这个模型。

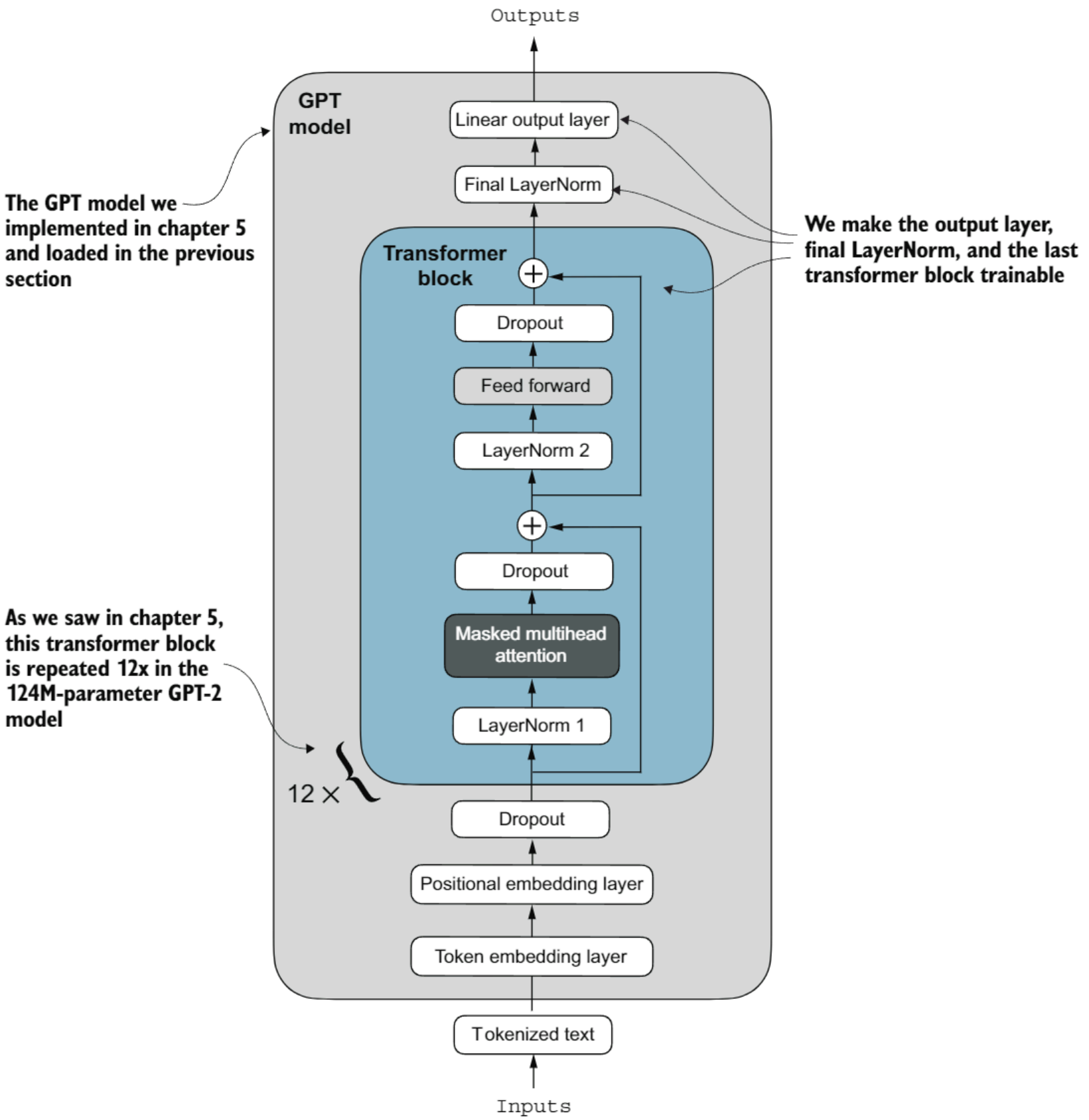


Figure 6.10 The GPT model includes 12 repeated transformer blocks. Alongside the output layer, we set the final `LayerNorm` and the last transformer block as trainable. The remaining 11 transformer blocks and the embedding layers are kept nontrainable.

instance, we can feed it an example text identical to our previously used example text:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape)
```

shape: (batch_size, num_tokens)

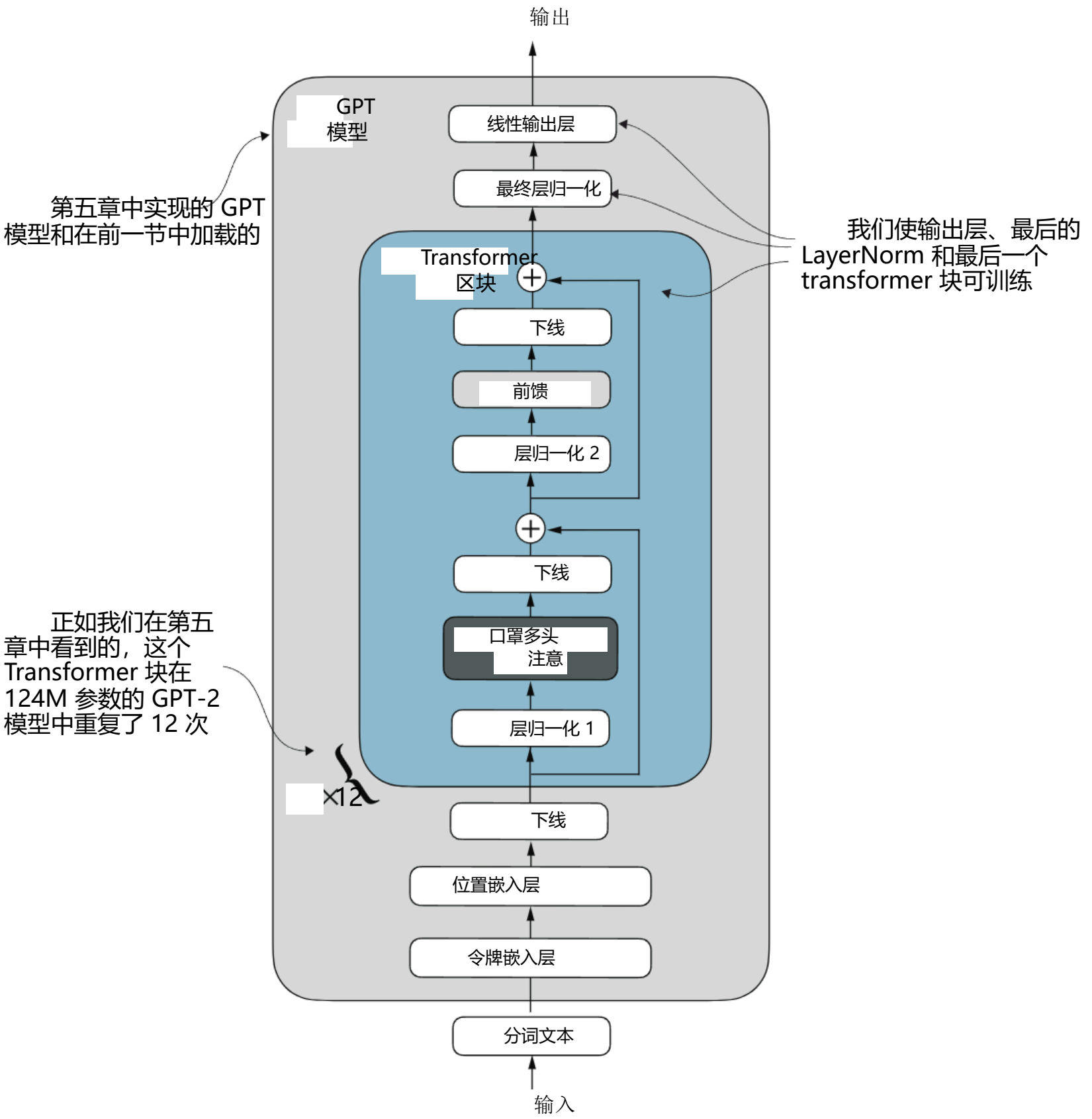


图 6.10 GPT 模型包含 12 个重复的 Transformer 块。在输出层旁边，我们将最后的 LayerNorm 和最后一个 Transformer 块设置为可训练。其余的 11 个 Transformer 块和嵌入层保持不可训练。

实例，我们可以给它一个与之前使用的示例文本相同的示例文本：

```
    inputs = tokenizer.encode("你有时间吗") inputs =
torch.tensor(inputs).unsqueeze(0) 打印("Inputs:",
inputs) 打印("Inputs 维度:", inputs.shape)
```

形状: (batch_size, num_tokens)

The print output shows that the preceding code encodes the inputs into a tensor consisting of four input tokens:

```
Inputs: tensor([[5211,  345,  423,  640]])
Inputs dimensions: torch.Size([1, 4])
```

Then, we can pass the encoded token IDs to the model as usual:

```
with torch.no_grad():
    outputs = model(inputs)
print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape)
```

The output tensor looks like the following:

```
Outputs:
 tensor([[[-1.5854,  0.9904],
          [-3.7235,  7.4548],
          [-2.2661,  6.6049],
          [-3.5983,  3.9902]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

A similar input would have previously produced an output tensor of `[1, 4, 50257]`, where `50257` represents the vocabulary size. The number of output rows corresponds to the number of input tokens (in this case, four). However, each output's embedding dimension (the number of columns) is now 2 instead of 50,257 since we replaced the output layer of the model.

Remember that we are interested in fine-tuning this model to return a class label indicating whether a model input is "spam" or "not spam." We don't need to fine-tune all four output rows; instead, we can focus on a single output token. In particular, we will focus on the last row corresponding to the last output token, as shown in figure 6.11.

To extract the last output token from the output tensor, we use the following code:

```
print("Last output token:", outputs[:, -1, :])
```

This prints

```
Last output token: tensor([[-3.5983,  3.9902]])
```

We still need to convert the values into a class-label prediction. But first, let's understand why we are particularly interested in the last output token only.

We have already explored the attention mechanism, which establishes a relationship between each input token and every other input token, and the concept of a *causal attention mask*, commonly used in GPT-like models (see chapter 3). This mask restricts a

打印输出显示，前面的代码将输入编码成一个包含四个输入标记的张量：

```
输入：tensor([[5211,          345,    423,    640]])
输入     维度：torch.Size([1, 4])
```

然后，我们可以像往常一样将编码后的令牌 ID 传递给模型：

```
使用 torch.no_grad():
outputs = 模型(inputs) 打印("输出:\n", outputs)
打印("输出维度:", outputs.shape)
```

输出张量看起来如下：

```
输出：
 张量([[[-1.5854,          0.9904]
            [-3.7235, 7.4548]
            [-2.2661, 6.6049]
3.9902]]]) 辅-3.5983,]
出维度：                                          4, 2])
torch.Size([1,])
```

类似输入之前会产生一个输出张量[1, 4, 50257]，其中 50257 代表词汇量大小。输出行数对应输入标记的数量（在这种情况下，四个）。然而，由于我们替换了模型的输出层，每个输出的嵌入维度（列数）现在是 2 而不是 50257。

记住，我们感兴趣的是微调此模型以返回一个类别标签，指示模型输入是“垃圾邮件”还是“非垃圾邮件”。我们不需要微调所有四行输出；相反，我们可以专注于单个输出标记。特别是，我们将关注最后一行，对应于最后一个输出标记，如图 6.11 所示。

从输出张量中提取最后一个输出标记，我们使用以下代码：

```
打印("最后输出令牌：")            outputs[:, -1, :])
```

这会打印

```
上一输出标记：tensor([[-3.5983,]]              3.9902]])
```

我们仍然需要将值转换为类别预测。但首先，让我们了解为什么我们特别关注最后一个输出标记。

我们已经探讨了注意力机制，它建立了每个输入标记与所有其他输入标记之间的关系，以及因果注意力掩码的概念，这在类似 GPT 的模型中常用（见第 3 章）。此掩码限制了

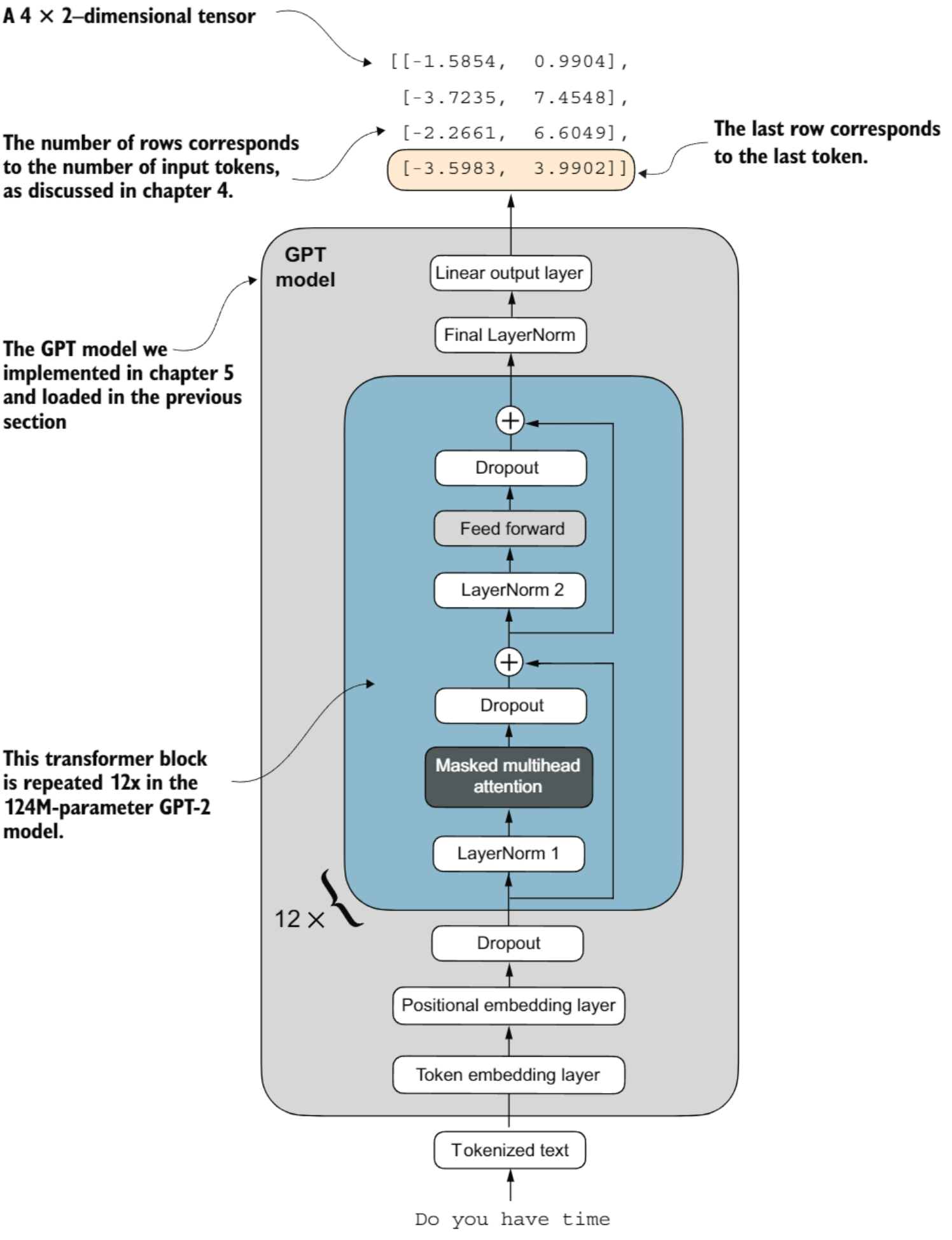


Figure 6.11 The GPT model with a four-token example input and output. The output tensor consists of two columns due to the modified output layer. We are only interested in the last row corresponding to the last token when fine-tuning the model for spam classification.

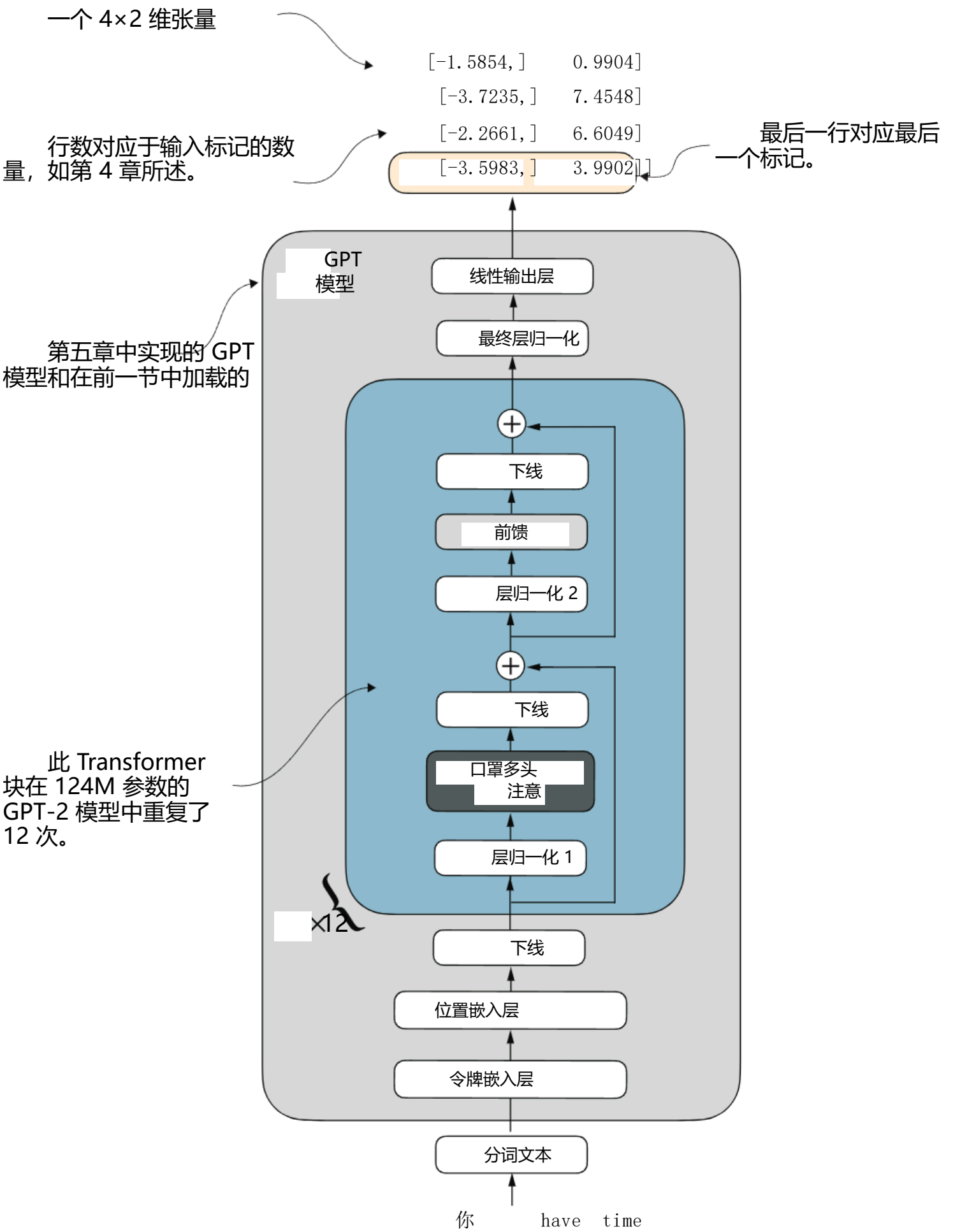


图 6.11 具有四个标记示例输入和输出的 GPT 模型。由于修改后的输出层，输出张量包含两列。在微调模型进行垃圾邮件分类时，我们只对最后一行感兴趣，对应于最后一个标记。

token's focus to its current position and the those before it, ensuring that each token can only be influenced by itself and the preceding tokens, as illustrated in figure 6.12.

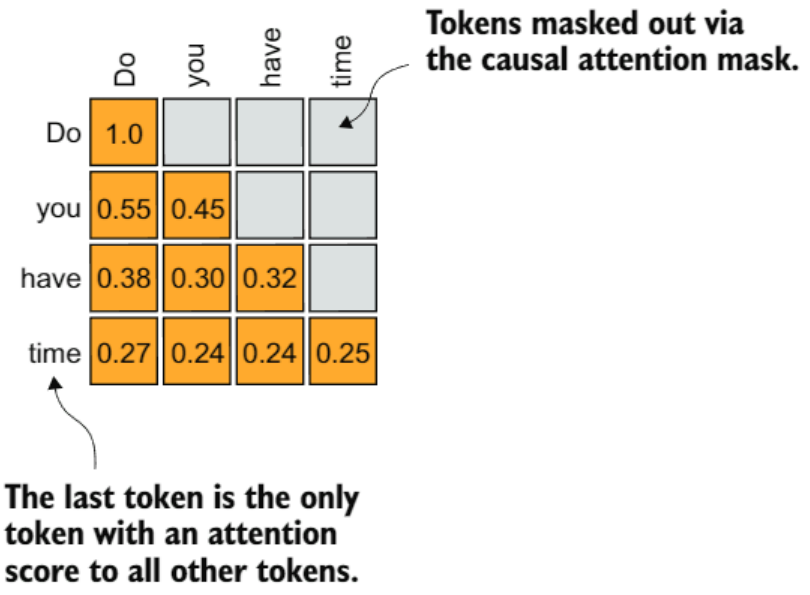


Figure 6.12 The causal attention mechanism, where the attention scores between input tokens are displayed in a matrix format. The empty cells indicate masked positions due to the causal attention mask, preventing tokens from attending to future tokens. The values in the cells represent attention scores; the last token, `time`, is the only one that computes attention scores for all preceding tokens.

Given the causal attention mask setup in figure 6.12, the last token in a sequence accumulates the most information since it is the only token with access to data from all the previous tokens. Therefore, in our spam classification task, we focus on this last token during the fine-tuning process.

We are now ready to transform the last token into class label predictions and calculate the model's initial prediction accuracy. Subsequently, we will fine-tune the model for the spam classification task.

> **Exercise 6.3 Fine-tuning the first vs. last token**
> Try fine-tuning the first output token. Notice the changes in predictive performance compared to fine-tuning the last output token.

6.6 Calculating the classification loss and accuracy

Only one small task remains before we fine-tune the model: we must implement the model evaluation functions used during fine-tuning, as illustrated in figure 6.13.

Before implementing the evaluation utilities, let's briefly discuss how we convert the model outputs into class label predictions. We previously computed the token ID of the next token generated by the LLM by converting the 50,257 outputs into probabilities via the `softmax` function and then returning the position of the highest probability via the `argmax` function. We take the same approach here to calculate whether the model outputs a "spam" or "not spam" prediction for a given input, as shown in figure 6.14. The only difference is that we work with 2-dimensional instead of 50,257-dimensional outputs.

token 的焦点定位到其当前位置及其之前的那些位置，确保每个 token 只能受到自身和前面 token 的影响，如图 6.12 所示。

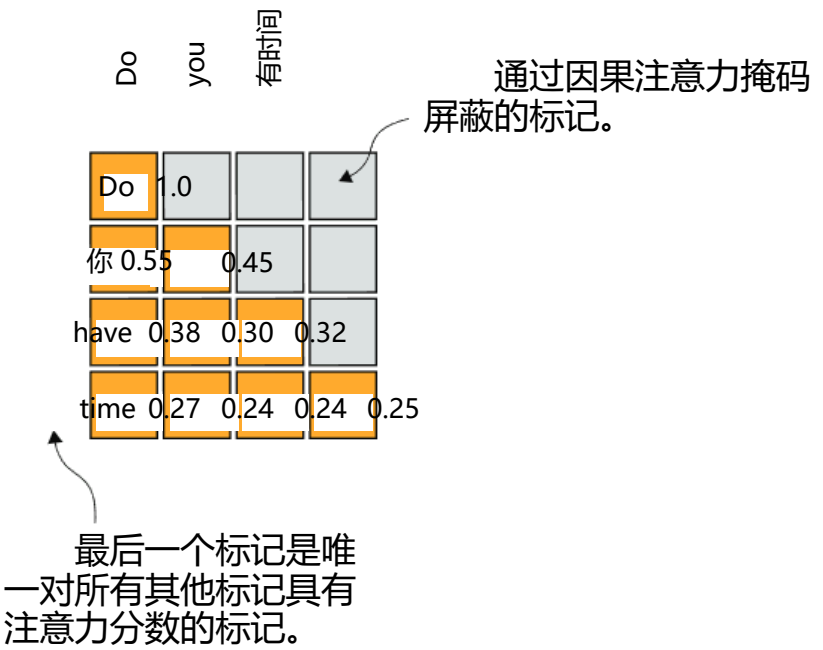


图 6.12 因果注意力机制，其中输入标记之间的注意力得分以矩阵格式显示。空单元格表示由于因果注意力掩码而屏蔽的位置，防止标记关注未来的标记。单元格中的值代表注意力得分；最后一个标记，时间，是唯一一个为所有先前标记计算注意力得分的标记。

鉴于图 6.12 中的因果注意力掩码设置，序列中的最后一个标记累积了最多的信息，因为它是有权访问所有先前标记数据的唯一标记。因此，在我们的垃圾邮件分类任务中，我们在微调过程中专注于这个最后一个标记。

我们现在准备将最后一个标记转换为类别标签预测，并计算模型的初始预测准确率。随后，我们将对模型进行微调以完成垃圾邮件分类任务。

练习 6.3 首个与最后一个标记的微调

尝试微调第一个输出标记。注意与微调最后一个输出标记相比，预测性能的变化。

6.6 计算分类损失和准确率

仅剩一个小任务需要在微调模型之前完成：我们必须实现如图 6.13 所示在微调过程中使用的模型评估函数。

在实现评估工具之前，让我们简要讨论如何将模型输出转换为类别标签预测。我们之前通过将 50,257 个输出转换为概率并通过 argmax 函数返回最高概率的位置，计算了由LLM生成的下一个标记的标记 ID。在这里，我们采取相同的方法来计算模型对给定输入的“垃圾邮件”或“非垃圾邮件”预测，如图 6.14 所示。唯一的区别是我们处理的是二维输出而不是 50,257 维输出。

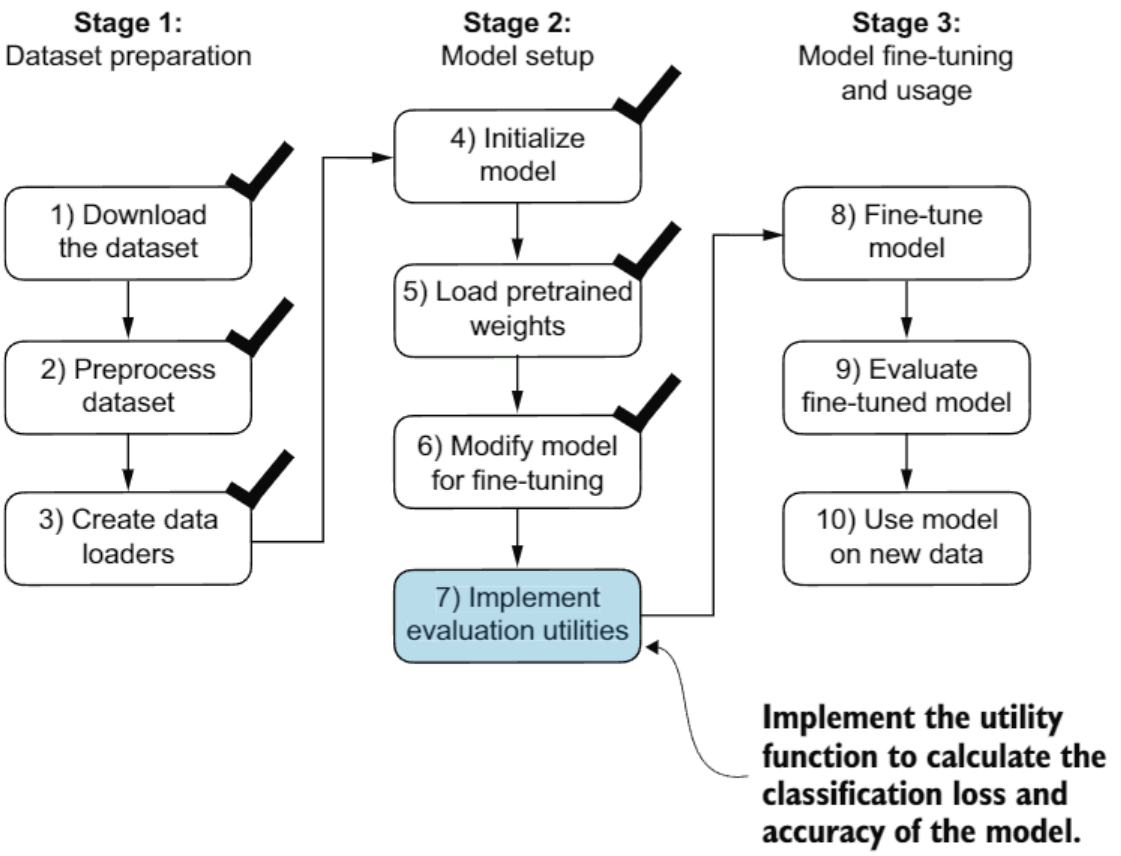


Figure 6.13 The three-stage process for classification fine-tuning the LLM. We've completed the first six steps. We are now ready to undertake the last step of stage 2: implementing the functions to evaluate the model's performance to classify spam messages before, during, and after the fine-tuning.

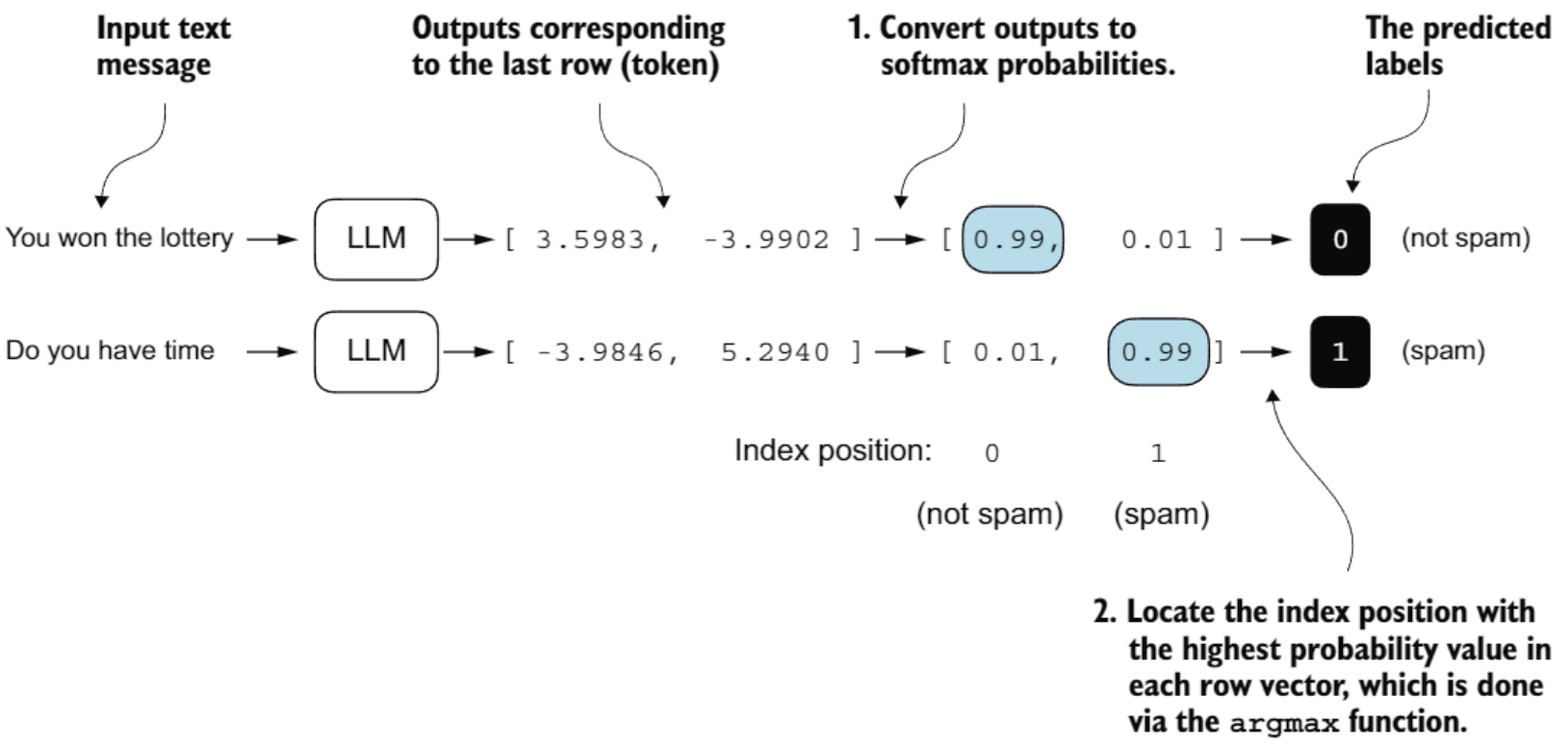


Figure 6.14 The model outputs corresponding to the last token are converted into probability scores for each input text. The class labels are obtained by looking up the index position of the highest probability score. The model predicts the spam labels incorrectly because it has not yet been trained.

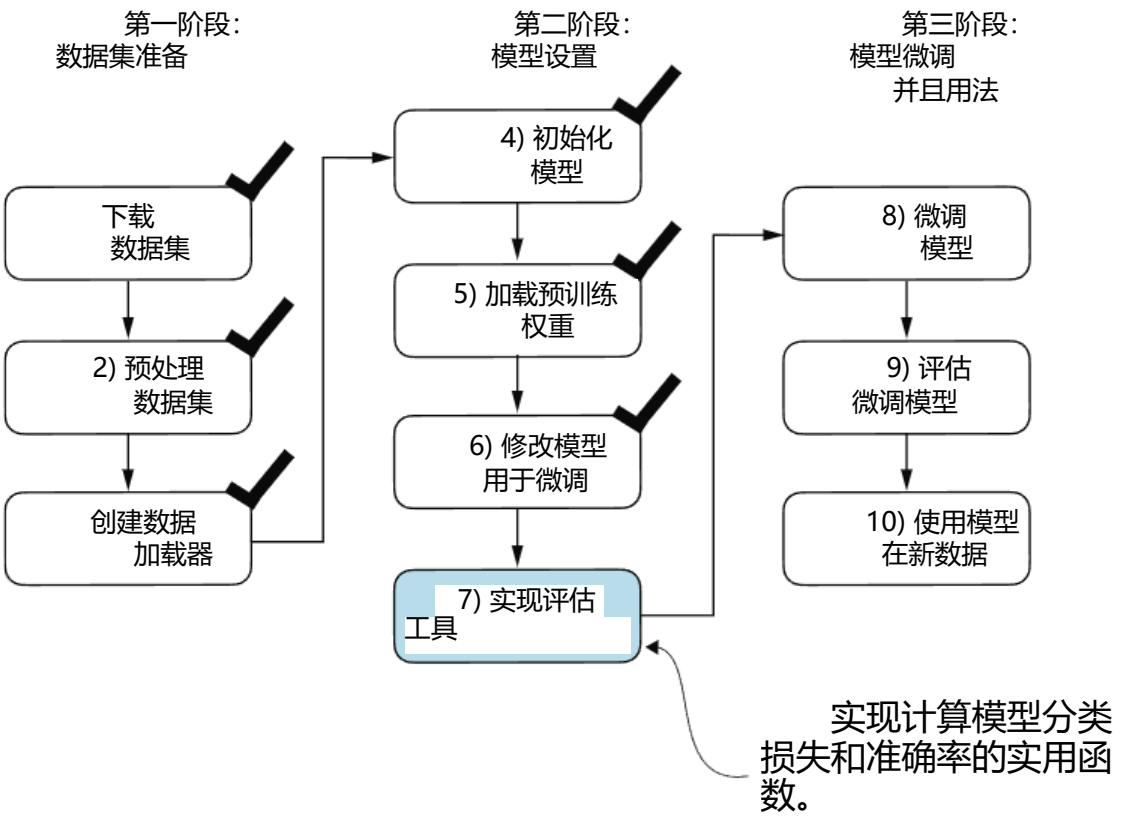


图 6.13 对分类微调的三个阶段过程。我们已完成前六个步骤。我们现在准备进行第二阶段最后一步：实现评估模型在微调前后对垃圾邮件进行分类的功能。

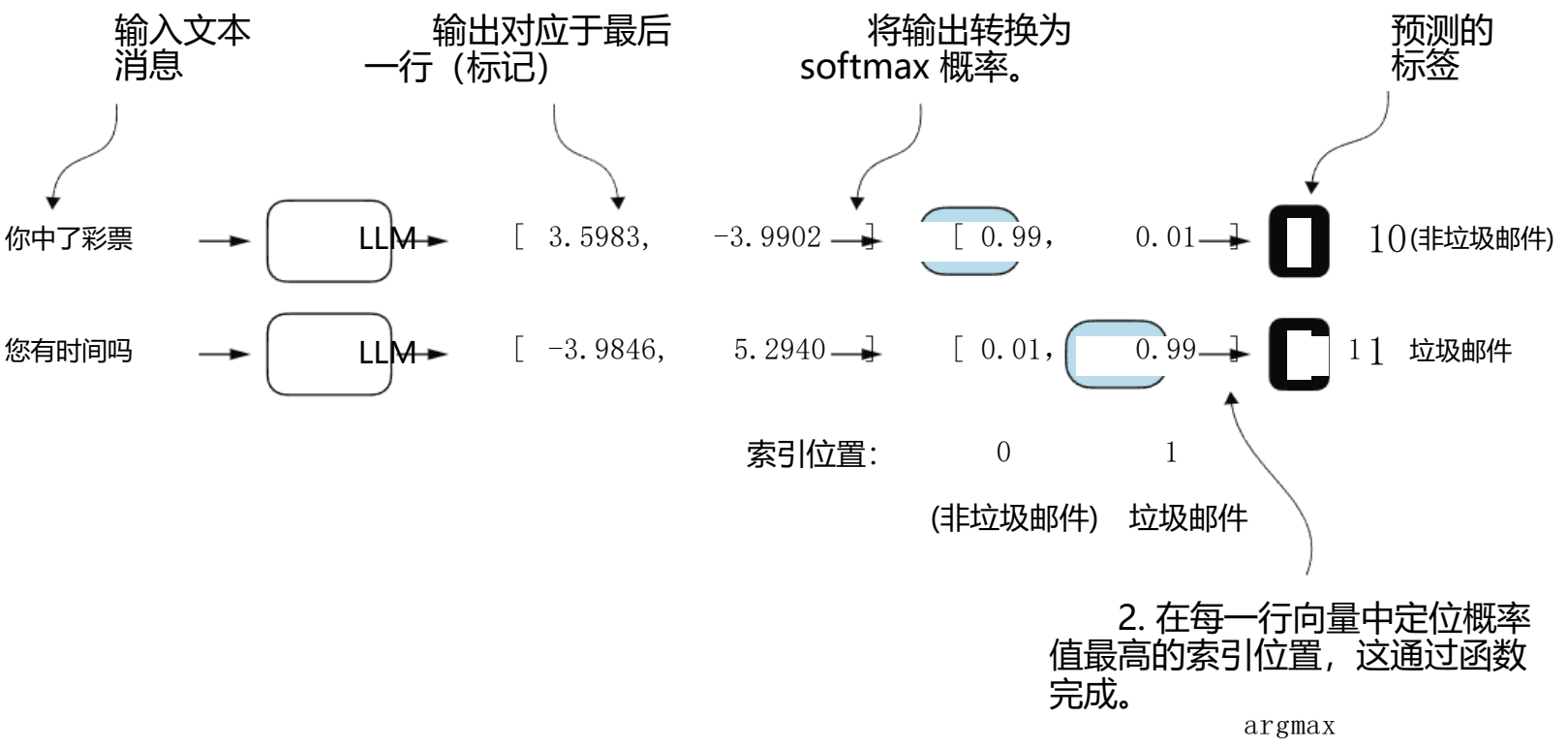


图 6.14 模型对最后一个标记的输出被转换为每个输入文本的概率分数。通过查找最高概率分数的索引位置来获得类别标签。由于尚未经过训练，模型预测垃圾邮件标签错误。

Let's consider the last token output using a concrete example:

```
print("Last output token:", outputs[:, -1, :])
```

The values of the tensor corresponding to the last token are

```
Last output token: tensor([[-3.5983,  3.9902]])
```

We can obtain the class label:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

In this case, the code returns `1`, meaning the model predicts that the input text is "spam." Using the `softmax` function here is optional because the largest outputs directly correspond to the highest probability scores. Hence, we can simplify the code without using softmax:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

This concept can be used to compute the classification accuracy, which measures the percentage of correct predictions across a dataset.

To determine the classification accuracy, we apply the `argmax`-based prediction code to all examples in the dataset and calculate the proportion of correct predictions by defining a `calc_accuracy_loader` function.

Listing 6.8 Calculating the classification accuracy

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if 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:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)


            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]
            predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (
```

Logits of last output token

让我们通过一个具体的例子来考虑最后一个标记的输出：

```
打印("最后输出令牌：")            outputs[:, -1, :])
```

张量对应最后一个标记的值

```
上一输出标记：tensor([[-3.5983,]]              3.9902]])
```

我们可以获得类别标签：

```
    probas = torch.softmax(输出[:, -1, :], dim=-1) label =
torch.argmax(probas) 打印("类别标签：", label.item())
```

在这种情况下，代码返回 1，表示模型预测输入文本是“垃圾邮件”。在这里使用 softmax 函数是可选的，因为最大的输出直接对应最高的概率分数。因此，我们可以简化代码而不使用 softmax：

```
    logits = 输出[:, -1, :] 标签 =
torch.argmax(logits) 打印("类别标签：", 标
签.item())
```

这个概念可以用来计算分类准确率，它衡量的是数据集中正确预测的百分比。

为了确定分类准确率，我们将基于 argmax 的预测代码应用于数据集中的所有示例，并通过定义 calc_accuracy_loader 函数来计算正确预测的比例。

列表 6.8 计算分类准确率

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None): # 计算加载器准确度
    model.eval() 正确预测
                                num_examples =    0, 0

    如果 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:
            input_batch = input_batch.to(设备) 目标
批次                    = target_batch 转移到设备上


            使用 torch.no_grad():
            :] 预测logits = 逻辑输出(input_batch)[:,       -1,
标签 =
torch.argm
ax(输出，
dim=-1)
            num_examples += predicted_labels.shape[0]
            correct_predictions +=
```

输出 token 的 logits

```
                (predicted_labels == target_batch).sum().item()
            )

        else:
            break
    return correct_predictions / num_examples
```

Let's use the function to determine the classification accuracies across various datasets estimated from 10 batches for efficiency:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Via the `device` setting, the model automatically runs on a GPU if a GPU with Nvidia CUDA support is available and otherwise runs on a CPU. The output is

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

As we can see, the prediction accuracies are near a random prediction, which would be 50% in this case. To improve the prediction accuracies, we need to fine-tune the model.

However, before we begin fine-tuning the model, we must define the loss function we will optimize during training. Our objective is to maximize the spam classification accuracy of the model, which means that the preceding code should output the correct class labels: `0` for non-spam and `1` for spam.

Because classification accuracy is not a differentiable function, we use cross-entropy loss as a proxy to maximize accuracy. Accordingly, the `calc_loss_batch` function remains the same, with one adjustment: we focus on optimizing only the last token, `model(input_batch)[:, -1, :]`, rather than all tokens, `model(input_batch)`:

```
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)[:, -1, :]
```

Logits of last output token

```
            (predicted_labels == target_batch).sum().item() ) ->
        (预测标签等于目标批次的).求和().项() )

    否则：
break 返回 正确预测
                                / 示例数量
```

让我们使用该函数来确定从 10 个批次估计的各个数据集的分类准确率以提高效率：

```
    设备 = torch.device("cuda" if torch.cuda.is_available() else "cpu") 模型.to(设
备)

    torch 手动设置随机种子(123) 训练准确率 =
calc_accuracy_loader()
    train_loader, model, device, num_batches=10)
val_accuracy = calc_accuracy_loader()

    val_loader, model, device, num_batches=10) 测试准确
率 = calc_accuracy_loader()

    test_loader, 模型, 设备, num_batches=10 )

    打印训练准确率：{train_accuracy*100:.2f}% 打印验证准确率：
{val_accuracy*100:.2f}% 打印测试准确率：{test_accuracy*100:.2f}%
```

通过设备设置，如果可用带有 Nvidia CUDA 支持的 GPU，则模型自动在 GPU 上运行，否则在 CPU 上运行。输出是

```
    训练准确率：46.25% 验证准确
率：45.00% 测试准确率：48.75%
```

如您所见，预测准确率接近随机预测，在这种情况下将是 50%。为了提高预测准确率，我们需要微调模型。

然而，在我们开始微调模型之前，我们必须定义我们将在训练中优化的损失函数。我们的目标是最大化模型的垃圾邮件分类准确率，这意味着前面的代码应该输出正确的类别标签：0 表示非垃圾邮件，1 表示垃圾邮件。

因为分类准确率不是一个可微函数，我们使用交叉熵损失作为代理来最大化准确率。因此，calc_loss_batch 函数保持不变，只有一个调整：我们只专注于优化最后一个

```
token, 模型(input_batch)[:, -1, :], 而不是所有 token, 模型(input_batch):

def calc_loss_batch(输入批次, 目标批次, 模型, 设备):
     input_batch = input_batch 转换到设备
 target_batch = target_batch 转换到设备 logits
 = 模型(input_batch)[:, -1, :]
```

输出 token 的 logits

```
    loss = torch.nn.functional.cross_entropy(logits, target_batch)
    return loss
```

We use the `calc_loss_batch` function to compute the loss for a single batch obtained from the previously defined data loaders. To calculate the loss for all batches in a data loader, we define the `calc_loss_loader` function as before.

Listing 6.9 Calculating the classification 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
```

Ensures number of batches doesn't exceed batches in data loader

Similar to calculating the training accuracy, we now compute the initial loss for each data set:

```
with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

Disables gradient tracking for efficiency because we are not training yet

The initial loss values are

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Next, we will implement a training function to fine-tune the model, which means adjusting the model to minimize the training set loss. Minimizing the training set loss will help increase the classification accuracy, which is our overall goal.

```
    损失 = torch.nn.functional.cross_entropy(logits, target_batch) 返回 损
失
```

我们使用 calc_loss_batch 函数来计算从先前定义的数据加载器中获得的单个批次的损失。为了计算数据加载器中所有批次的损失，我们之前定义了 calc_loss_loader 函数。

列表 6.9 计算分类损失

```
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
```

确保批次数量不超过数据加载器中的批次数量

与计算训练准确度类似，我们现在计算每个数据集的初始损失：

```
使用 torch.no_grad():
    train_loss = calc_loss_loader(
训练加载器，模型，设备，批次数=5）验证损失 = calc_loss_loader(val_loader, 模型，设备，
批次数=5）测试损失 = calc_loss_loader(test_loader, 模型，设备，批次数=5）打印(f"训练损失:
{train_loss:.3f}") 打印(f"验证损失: {val_loss:.3f}") 打印(f"测试损失: {test_loss:.3f}")
```

禁用梯度跟踪以提高效率，因为我们尚未开始训练

初始损失值

```
训练损失：2.453 验证损
失：2.583 测试损失：2.322
```

接下来，我们将实现一个训练函数以微调模型，这意味着调整模型以最小化训练集损失。最小化训练集损失将有助于提高分类准确率，这是我们总体目标。

6.7 Fine-tuning the model on supervised data

We must define and use the training function to fine-tune the pretrained LLM and improve its spam classification accuracy. The training loop, illustrated in figure 6.15, is the same overall training loop we used for pretraining; the only difference is that we calculate the classification accuracy instead of generating a sample text to evaluate the model.

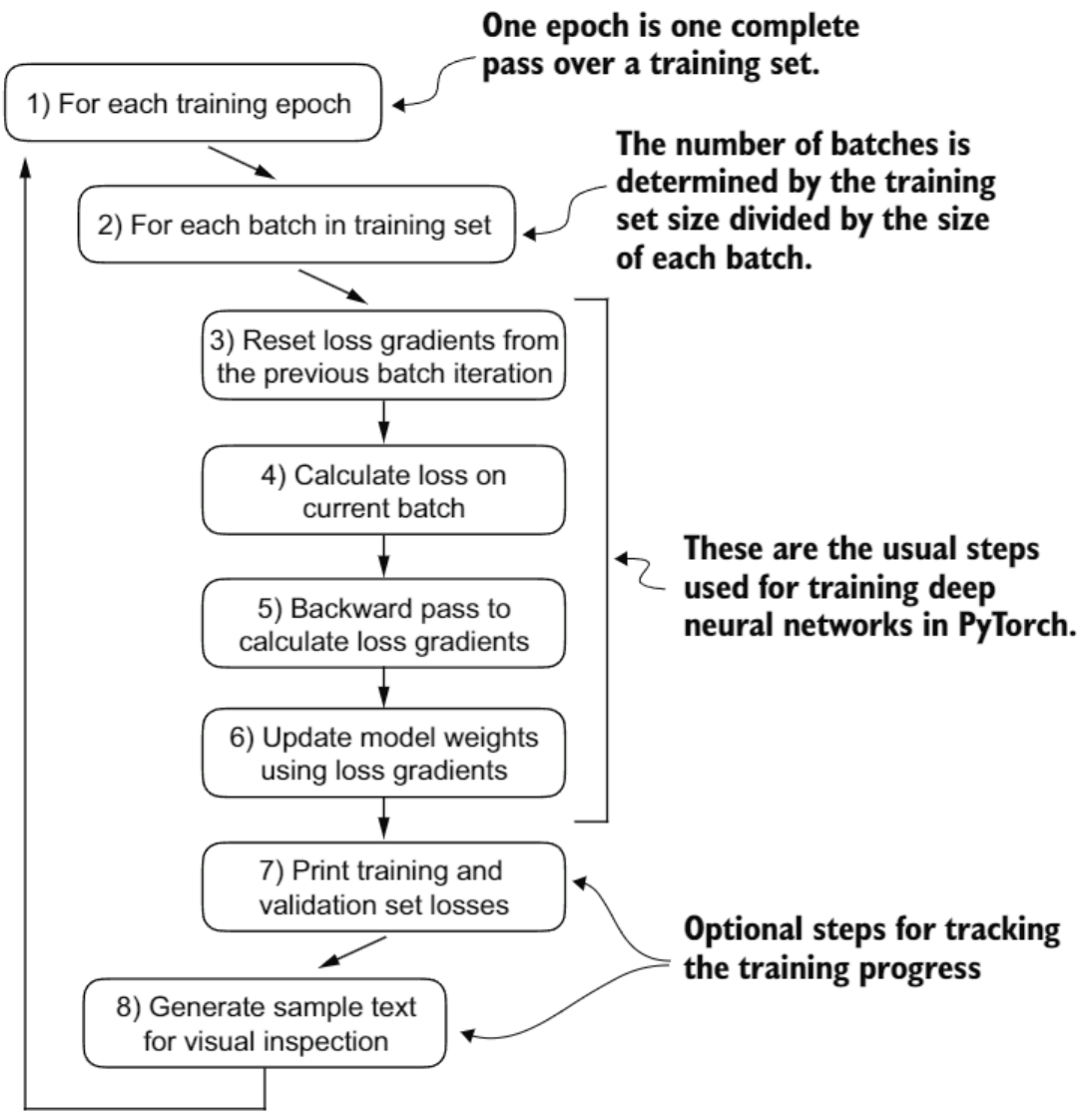


Figure 6.15 A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights to minimize the training set loss.

The training function implementing the concepts shown in figure 6.15 also closely mirrors the `train_model_simple` function used for pretraining the model. The only two distinctions are that we now track the number of training examples seen (`examples_seen`) instead of the number of tokens, and we calculate the accuracy after each epoch instead of printing a sample text.

6.7 微调模型在监督数据上

我们必须定义和使用训练函数来微调预训练的LLM并提高其垃圾邮件分类的准确性。如图 6.15 所示的训练循环，与我们用于预训练的整体训练循环相同；唯一的区别是我们计算分类准确性而不是生成样本文本来评估模型。

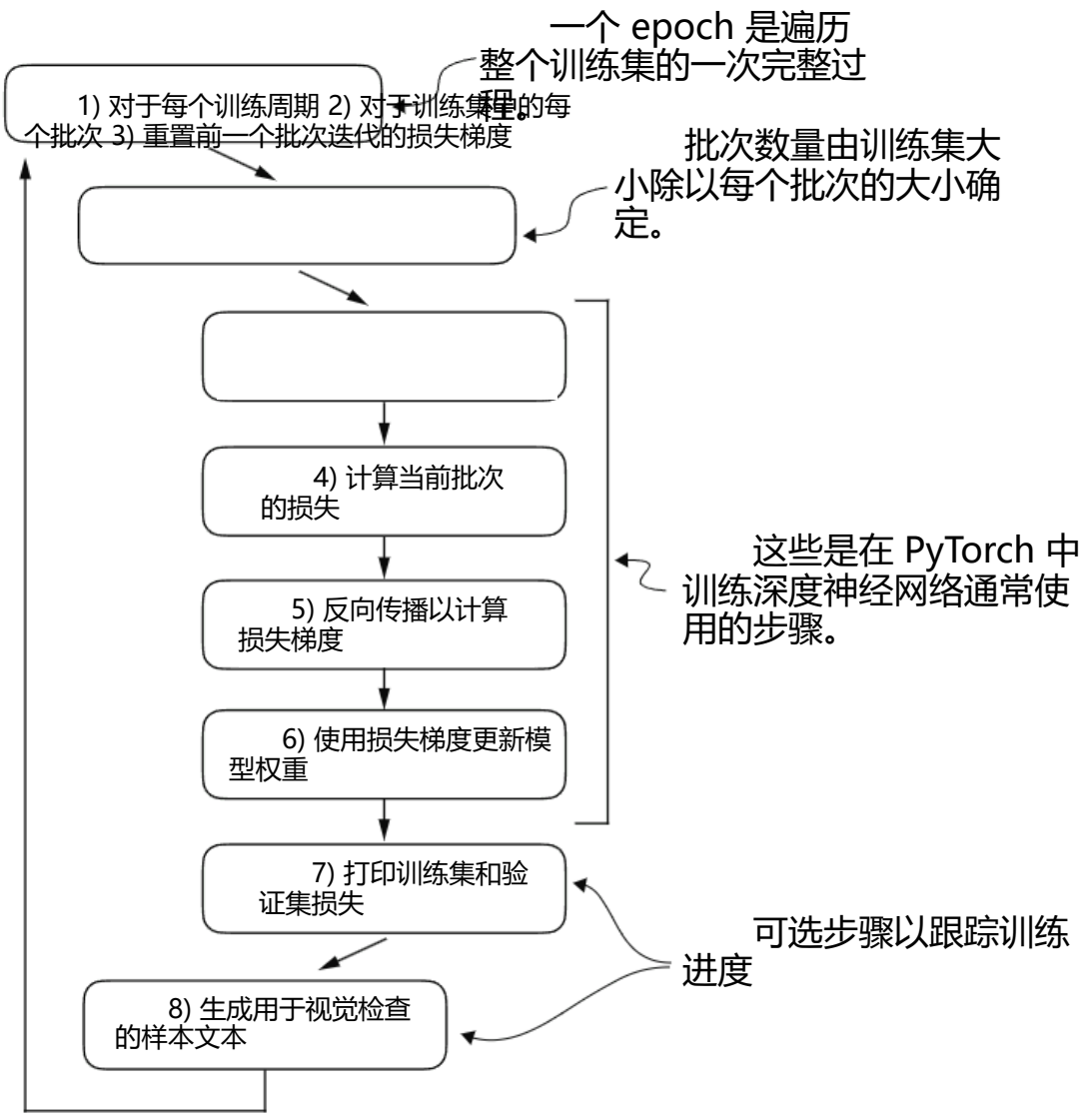


图 6.15 PyTorch 中训练深度神经网络的典型训练循环包括几个步骤，遍历训练集的批次进行多个 epoch。在每个循环中，我们计算每个训练集批次的损失以确定损失梯度，然后使用这些梯度来更新模型权重以最小化训练集损失。

训练函数实现了图 6.15 中所示的概念，也与用于预训练模型的 `train_model_simple` 函数非常相似。唯一的两个区别是，我们现在跟踪看到的训练示例数量（`examples_seen`）而不是标记数量，我们在每个 epoch 后计算准确率而不是打印样本文本。

Listing 6.10 Fine-tuning the model to classify spam

```
def train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs, eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_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()
            examples_seen += input_batch.shape[0]
            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)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        train_accuracy = calc_accuracy_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_accuracy = calc_accuracy_loader(
            val_loader, model, device, num_batches=eval_iter
        )

        print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
        print(f"Validation accuracy: {val_accuracy*100:.2f}%")
        train_accs.append(train_accuracy)
        val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen
```

Initialize lists to track losses and examples seen

Main training loop

Sets model to training mode

Resets loss gradients from the previous batch iteration

Calculates loss gradients

Updates model weights using loss gradients

New: tracks examples instead of tokens

Optional evaluation step

Calculates accuracy after each epoch

The `evaluate_model` function is identical to the one we used for pretraining:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
```

列表 6.10 微调模型以分类垃圾邮件

```
def  训练简单分类器
     模型，训练加载器，验证加载器，优化器，设备，训练轮数，评估频率，评估迭代次数：训练损失，验证损失，训练准确率，验证准确率 = [], [], [], [] 已见示例，全局步数 = 0, -1
```

初始化列表为跟踪损失示例所见

主训练循环

```
for epoch 在 range(num_epochs):
     model.train()
```

设置模型为训练模式

重置上一批迭代中的损失梯度

```
     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(). 步骤增加输入批次形状[0] 全局步骤增加 1
```

计算损失梯度

更新模型权重使用损失梯度

可选评估 step

新：跟踪示例而不是标记

```
          if global_step % eval_freq == 0: # 如果 global_step 对 eval_freq 取模等于 0：
               训练损失，验证损失 = 评估模型(
               模型，train_loader，val_loader，device，eval_iter) 将训练损失添加到 train_losses 中，将验证损失添加到 val_losses 中 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}"
```

计算每个 epoch 后的准确率

```
          训练准确率 = calc_accuracy_loader(
          train_loader, model, device, num_batches=eval_iter)
     val_accuracy = calc_accuracy_loader()

          val_loader, model, device, num_batches=评估迭代器

          打印训练准确率：{train_accuracy*100:.2f}% | 打印验证准确率：{val_accuracy*100:.2f}% train_accs.append(train_accuracy)
     val_accs.append(val_accuracy)

返回  训练损失  val_losses  训练准确率  val_accs  已看到的示例
```

评估模型函数与我们用于预训练的函数相同：

```
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 = 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
```

Next, we initialize the optimizer, set the number of training epochs, and initiate the training using the `train_classifier_simple` function. The training takes about 6 minutes on an M3 MacBook Air laptop computer and less than half a minute on a V100 or A100 GPU:

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5


train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50,
        eval_iter=5
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

The output we see during the training is as follows:

```
Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 5.65 minutes.
```

```
        训练损失 = calc_loss_loader(
        train_loader, model, device, num_batches=eval_iter) val_loss =
    calc_loss_loader()


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

接下来，我们初始化优化器，设置训练轮数，并使用 train_classifier_simple 函数开始训练。在 M3 MacBook Air 笔记本电脑上训练大约需要 6 分钟，而在 V100 或 A100 GPU 上不到半分钟：

```
导入 time

start_time = time.time() torch 手动设置种子(123) optimizer = torch.optim.AdamW(model 参
数(), lr=5e-5, weight_decay=0.1) num_epochs = 5



训练损失，验证损失，训练准确率，验证准确率，已见示例 = \
    训练简单分类器
    模型，训练加载器，验证加载器，优化器，设备，
num_epochs=num_epochs，评估频率=50，评估迭代=5
```

```
end_time = 时间.time() 执行时间（分钟）= (end_time -
start_time) / 60 打印(f"训练")
                                完成于 {执行时间（分钟）:.2f}                分钟。")
```

训练过程中我们看到的输出如下：

```
第 1 集（步骤 000000）：训练损失 2.153，验证损失 2.392 第 1 集
（步骤 000050）：训练损失 0.617，验证损失 0.637 第 1 集（步骤
000100）：训练损失 0.523，验证损失 0.557 训练准确率：70.00% | 验证准
确率：72.50% 第 2 集（步骤 000150）：训练损失 0.561，验证损失 0.489
第 2 集（步骤 000200）：训练损失 0.419，验证损失 0.397 第 2 集（步骤
000250）：训练损失 0.409，验证损失 0.353 训练准确率：82.50% | 验证准
确率：85.00% 第 3 集（步骤 000300）：训练损失 0.333，验证损失 0.320
第 3 集（步骤 000350）：训练损失 0.340，验证损失 0.306 训练准确率：
90.00% | 验证准确率：90.00% 第 4 集（步骤 000400）：训练损失 0.136，
验证损失 0.200 第 4 集（步骤 000450）：训练损失 0.153，验证损失
0.132 第 4 集（步骤 000500）：训练损失 0.222，验证损失 0.137 训练准
确率：100.00% | 验证准确率：97.50% 第 5 集（步骤 000550）：训练损失
0.207，验证损失 0.143 第 5 集（步骤 000600）：训练损失 0.083，验证损
失 0.074 训练准确率：100.00% | 验证准确率：97.50% 训练完成，耗时
5.65 分钟。
```

We then use Matplotlib to plot the loss function for the training and validation set.

Listing 6.11 Plotting the classification loss

```
import matplotlib.pyplot as plt

def plot_values(
        epochs_seen, examples_seen, train_values, val_values,
        label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    ax1.plot(epochs_seen, train_values, label=f"Training {label}")
    ax1.plot(
        epochs_seen, val_values, linestyle="-.",
        label=f"Validation {label}"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    ax2 = ax1.twiny()
    ax2.plot(examples_seen, train_values, alpha=0)
    ax2.set_xlabel("Examples seen")

    fig.tight_layout()
    plt.savefig(f"{label}-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)
```

Plots training and validation loss against epochs

Creates a second x-axis for examples seen

Invisible plot for aligning ticks

Adjusts layout to make room

Figure 6.16 plots the resulting loss curves.

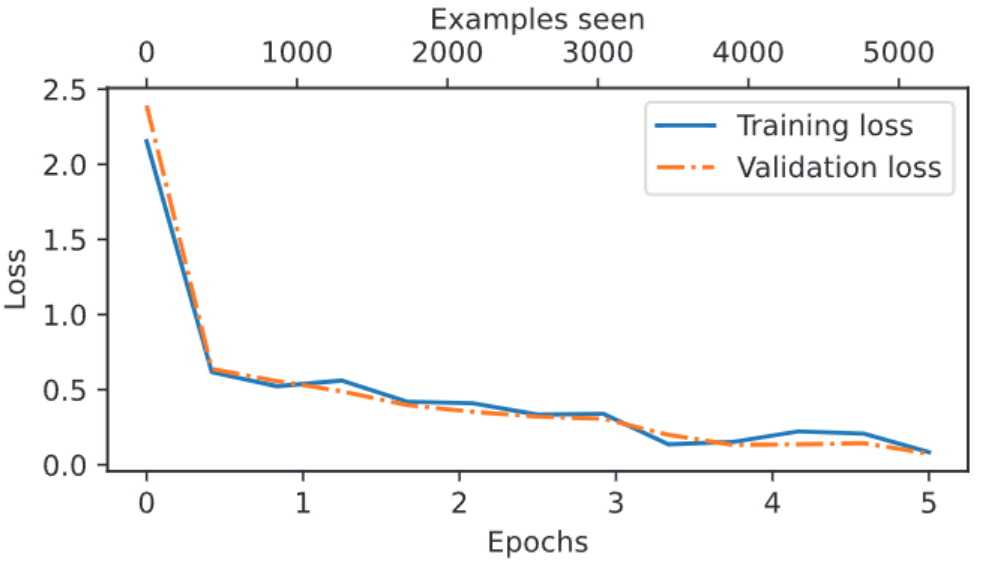


Figure 6.16 The model's training and validation loss over the five training epochs. Both the training loss, represented by the solid line, and the validation loss, represented by the dashed line, sharply decline in the first epoch and gradually stabilize toward the fifth epoch. This pattern indicates good learning progress and suggests that the model learned from the training data while generalizing well to the unseen validation data.

然后我们使用 Matplotlib 绘制训练集和验证集的损失函数图。

列表 6.11 绘制分类损失图

```
导入     matplotlib.pyplot     as  plt

def  plot_values()
    epochs_seen, examples_seen, train_values, val_values, 标签="损
失"): fig, ax1 = plt.subplots(figsize=(5, 3))
```

绘图训练
验证损失
对抗时代

```
    ax1.plot(epochs_seen, train_values, 标签=f"训练 {标签}") ax1.plot(

    epochs_seen, val_values, 样式="-.", 标签=f"验证
{label}" ) ax1.set_xlabel("时代")
ax1.set_ylabel(label.capitalize()) ax1.legend()
```

创建
第二 X 轴
示例所见

```
    ax2 = ax1.twiny() ax2.plot(已见示例, 训练值,
alpha=0) ax2.set_xlabel("已见示例")
```

隐形剧情
对齐刻度

```
    fig.tight_layout() plt.savefig(f"
{标签}-plot.pdf") plt.show()
```

调整布局
腾出空间

```
    epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen,                    train_losses 的长度

    plot_values(epochs_tensor, examples_seen_tensor,)          训练损失          val_losses)
```

图 6.16 绘制了结果损失曲线。

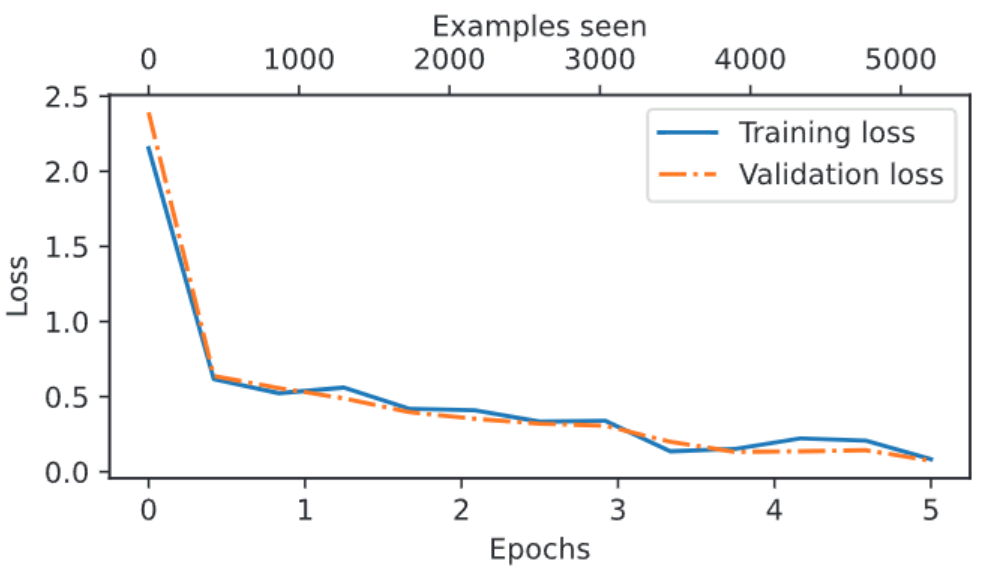


图 6.16 模型在五个训练周期中的训练和验证损失。训练损失（实线表示）和验证损失（虚线表示）在第一个周期内急剧下降，并逐渐稳定到第五个周期。这种模式表明良好的学习进展，并表明模型从训练数据中学习，同时很好地泛化到未见过的验证数据。

As we can see based on the sharp downward slope in figure 6.16, the model is learning well from the training data, and there is little to no indication of overfitting; that is, there is no noticeable gap between the training and validation set losses.

Choosing the number of epochs

Earlier, when we initiated the training, we set the number of epochs to five. The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation, although an epoch number of five is usually a good starting point. If the model overfits after the first few epochs as a loss plot (see figure 6.16), you may need to reduce the number of epochs. Conversely, if the trendline suggests that the validation loss could improve with further training, you should increase the number of epochs. In this concrete case, five epochs is a reasonable number as there are no signs of early overfitting, and the validation loss is close to 0.

Using the same `plot_values` function, let's now plot the classification accuracies:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="accuracy"
)
```

Figure 6.17 graphs the resulting accuracy. The model achieves a relatively high training and validation accuracy after epochs 4 and 5. Importantly, we previously set `eval_iter=5`

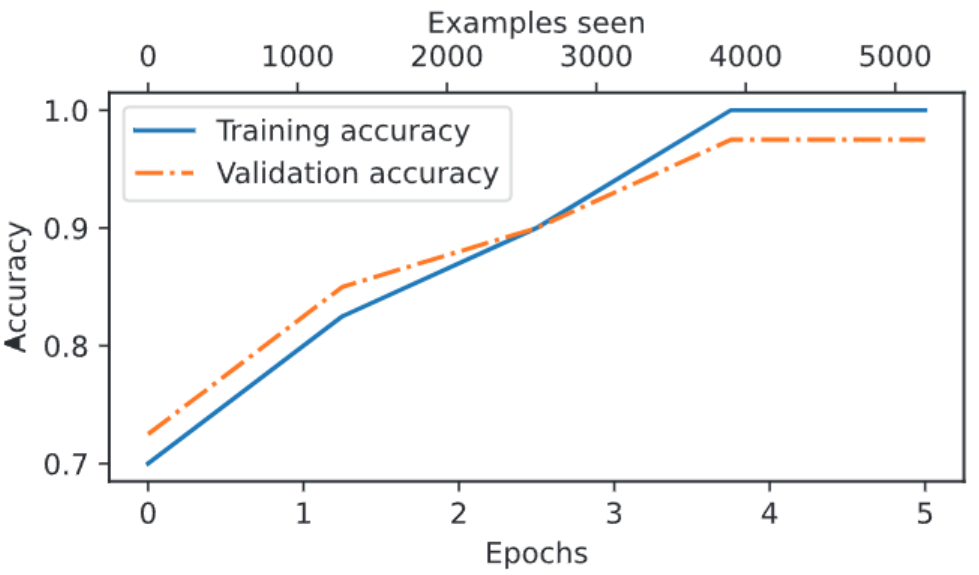


Figure 6.17 Both the training accuracy (solid line) and the validation accuracy (dashed line) increase substantially in the early epochs and then plateau, achieving almost perfect accuracy scores of 1.0. The close proximity of the two lines throughout the epochs suggests that the model does not overfit the training data very much.

如图 6.16 所示，我们可以看到模型从训练数据中学习得很好，几乎没有过拟合的迹象；也就是说，训练集和验证集损失之间没有明显的差距。

选择训练轮数

之前，当我们开始训练时，我们将 epoch 数设置为五个。epoch 数取决于数据集和任务的难度，没有通用的解决方案或建议，尽管五个 epoch 通常是一个好的起点。如果模型在最初的几个 epoch 后过拟合（如损失图所示，见图 6.16），你可能需要减少 epoch 数。相反，如果趋势线表明验证损失可以通过进一步训练而改善，你应该增加 epoch 数。在这个具体案例中，五个 epoch 是一个合理的数字，因为没有出现早期过拟合的迹象，验证损失接近 0。

使用相同的 plot_values 函数，现在让我们绘制分类准确率：

```
    epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor        torch.linspace(0, examples_seen, len(train_accs))

    plot_values()
    epochs_tensor, examples_seen_tensor, train_accs, val_accs, 标签="准
确率" )
```

图 6.17 显示了结果的准确率。模型在 4 和 5 个 epoch 后达到了相对较高的训练和验证准确率。重要的是，我们之前设置了 eval_iter=5

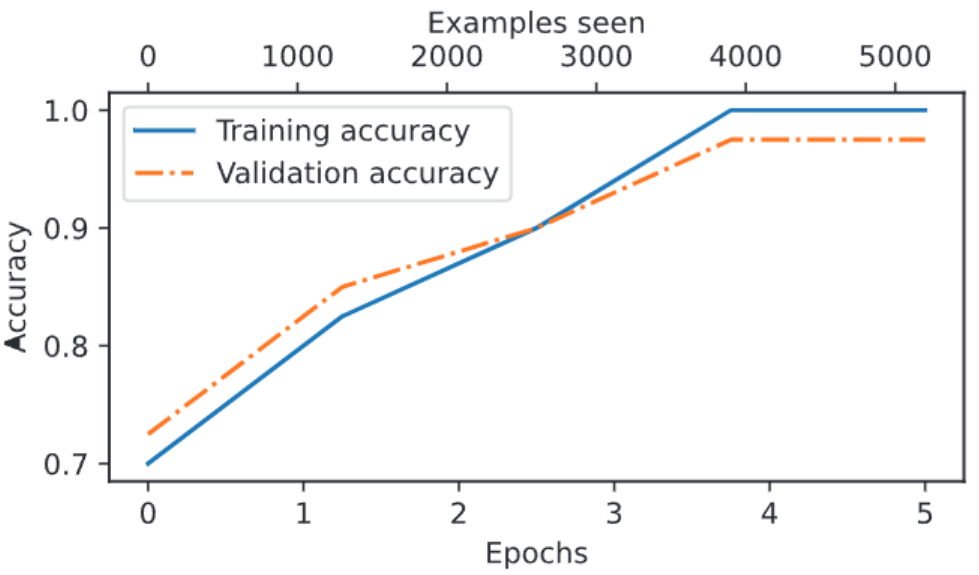


图 6.17 训练准确率（实线）和验证准确率（虚线）在早期 epoch 显著增加，然后趋于平稳，几乎达到完美的准确率 1.0。两条线在整个 epoch 中的紧密接近表明模型对训练数据过度拟合的程度不大。

when using the `train_classifier_simple` function, which means our estimations of training and validation performance are based on only five batches for efficiency during training.

Now we must calculate the performance metrics for the training, validation, and test sets across the entire dataset by running the following code, this time without defining the `eval_iter` value:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

The training and test set performances are almost identical. The slight discrepancy between the training and test set accuracies suggests minimal overfitting of the training data. Typically, the validation set accuracy is somewhat higher than the test set accuracy because the model development often involves tuning hyperparameters to perform well on the validation set, which might not generalize as effectively to the test set. This situation is common, but the gap could potentially be minimized by adjusting the model's settings, such as increasing the dropout rate (`drop_rate`) or the `weight_decay` parameter in the optimizer configuration.

6.8 Using the LLM as a spam classifier

Having fine-tuned and evaluated the model, we are now ready to classify spam messages (see figure 6.18). Let's use our fine-tuned GPT-based spam classification model. The following `classify_review` function follows data preprocessing steps similar to those we used in the `SpamDataset` implemented earlier. Then, after processing text into token IDs, the function uses the model to predict an integer class label, similar to what we implemented in section 6.6, and then returns the corresponding class name.

使用 train_classifier_simple 函数时，这意味着我们的训练和验证性能估计仅基于五个批次以提高训练效率。

现在我们必须通过运行以下代码，在整个数据集上计算训练、验证和测试集的性能指标，这次不定义 eval_iter 值：

```
    训练准确率 = calc_accuracy_loader(train_loader, model, device) 验证准确率 =
calc_accuracy_loader(val_loader, model, device) 测试准确率 =
calc_accuracy_loader(test_loader, model, device)


    打印训练准确率：{train_accuracy*100:.2f}% 打印验证准确率：
{val_accuracy*100:.2f}% 打印测试准确率：{test_accuracy*100:.2f}%
```

结果准确度值是

```
    训练准确率：97.21% 验证准确
率：97.32% 测试准确率：95.67%
```

训练集和测试集的性能几乎相同。训练集和测试集准确率之间的微小差异表明训练数据过拟合最小。通常，验证集准确率略高于测试集准确率，因为模型开发通常涉及调整超参数以在验证集上表现良好，这可能不会像对测试集那样有效地推广。这种情况很常见，但通过调整模型的设置，如增加 dropout 率（drop_rate）或优化器配置中的 weight_decay 参数，可以潜在地最小化这种差距。

6.8 使用LLM作为垃圾邮件分类器

对模型进行微调和评估后，我们现在可以分类垃圾邮件（见图 6.18）。让我们使用我们的基于 GPT 的微调垃圾邮件分类模型。下面的 classify_review 函数遵循与我们在之前实现的 SpamDataset 中使用的数据预处理步骤类似。然后，在将文本处理成标记 ID 后，该函数使用模型预测一个整数类别标签，类似于我们在第 6.6 节中实现的方法，然后返回相应的类别名称。

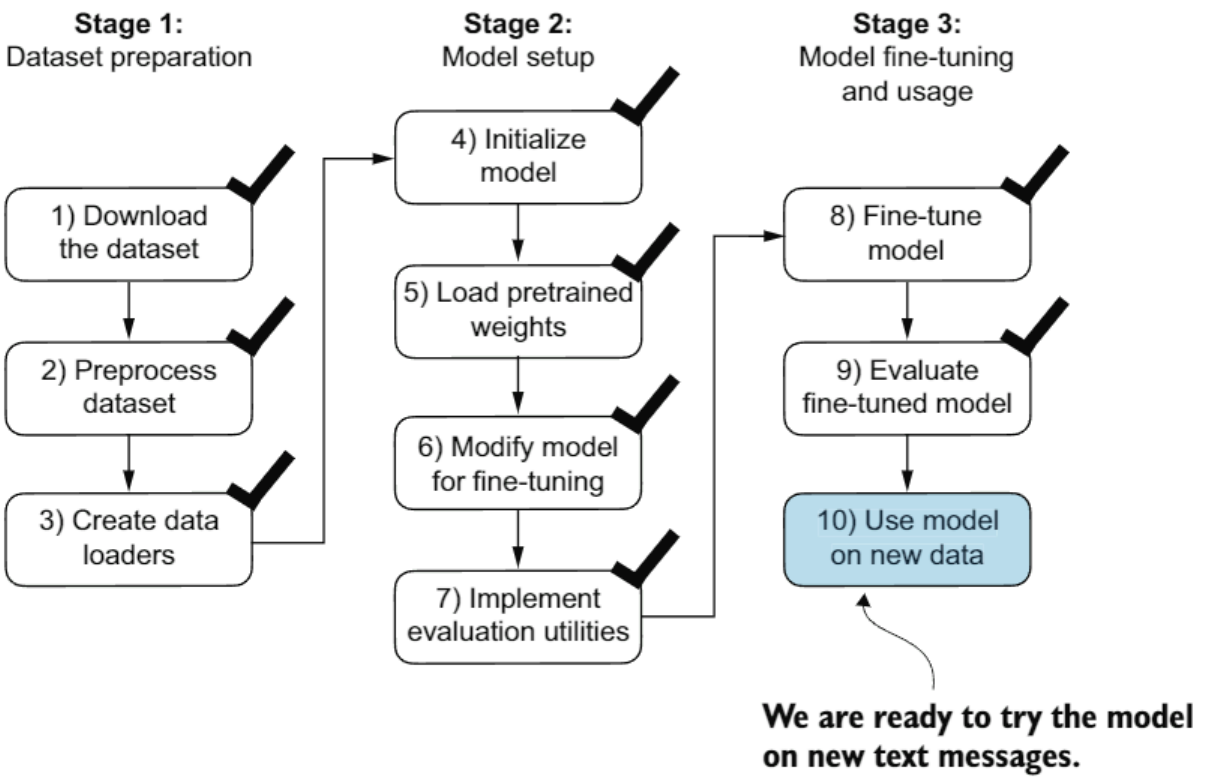


Figure 6.18 The three-stage process for classification fine-tuning our LLM. Step 10 is the final step of stage 3—using the fine-tuned model to classify new spam messages.

Listing 6.12 Using the model to classify new texts

```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]

    input_ids = input_ids[:min(
        max_length, supported_context_length
    )]

    input_ids += [pad_token_id] * (max_length - len(input_ids))

    input_tensor = torch.tensor(
        input_ids, device=device
    ).unsqueeze(0)

    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam"
```

Prepares inputs to the model

Truncates sequences if they are too long

Pads sequences to the longest sequence

Adds batch dimension

Models inference without gradient tracking

Logits of the last output token

Returns the classified result

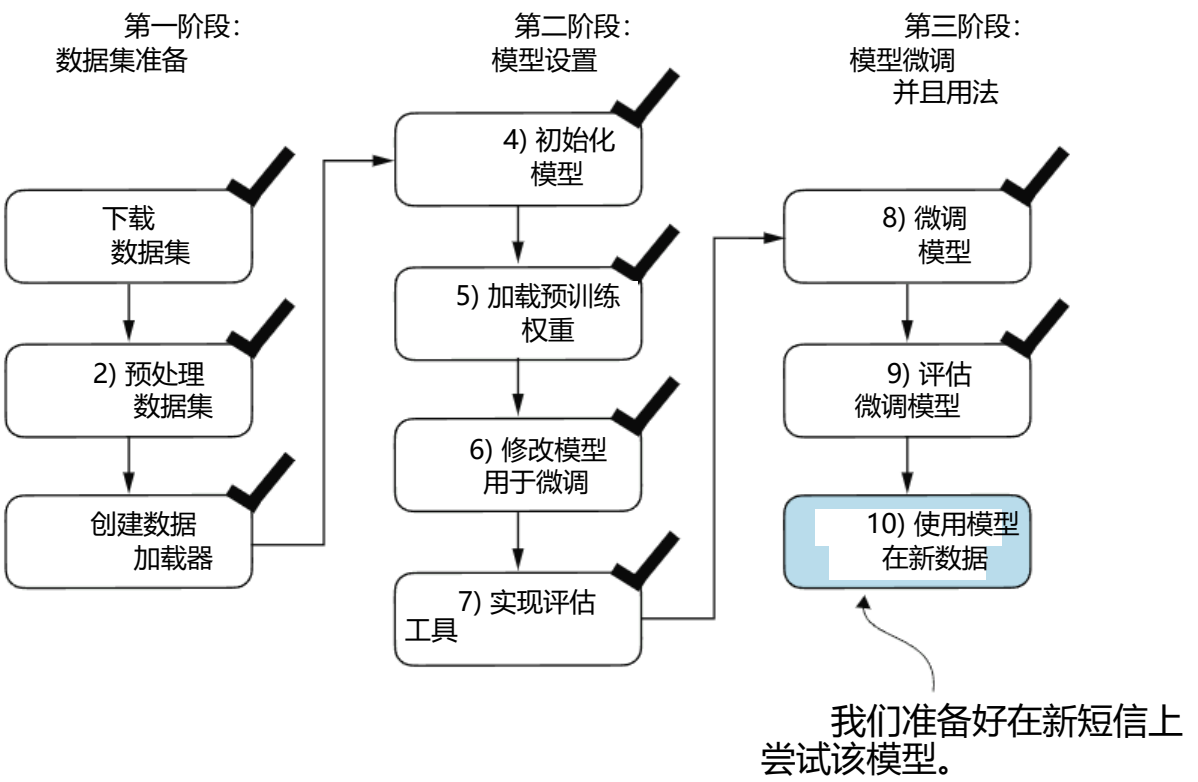


图 6.18 对分类微调的三个阶段过程。第 10 步是第三阶段的最后一步——使用微调后的模型来分类新的垃圾邮件。

列表 6.12 使用模型对新的文本进行分类

```
def classify_review(
    text, 模型, 分词器, 设备, max_length=None,
pad_token_id=50256): 模型评估()


    input_ids = 分词器.encode(text) 支持的
上下文长度 =                          model.pos_emb.weight.shape[1]


    input_ids = input_ids[:min(
    [max_length, supported_context_length ]


    输入 ID     += [pad_token_id] * (最大长度 - 输入 ID 长度)


    input_tensor = torch.tensor(
    input_ids, 设备
=device).unsqueeze(0)


    使用 torch.no_grad():
    logits = 模型(input_tensor)[:, -1, :] 预测
标签 = torch.argmax(logits,)                    dim=-1).item()


    返回 "垃圾邮件" if  predicted_label ==     1 else  "not  垃圾邮件"
                                          否则
```

准备输入至于模型

截断过长序列

垫序列对最长的序列

添加批次维度

模型推理没有梯度跟踪

最后一输出标记的对数

返回分类结果

Let's try this `classify_review` function on an example text:

```
text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The resulting model correctly predicts `"spam"`. Let's try another example:

```
text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(
    text_2, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The model again makes a correct prediction and returns a "not spam" label.

Finally, let's save the model in case we want to reuse the model later without having to train it again. We can use the `torch.save` method:

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

Once saved, the model can be loaded:

```
model_state_dict = torch.load("review_classifier.pth, map_location=device")
model.load_state_dict(model_state_dict)
```

Summary

- There are different strategies for fine-tuning LLMs, including classification fine-tuning and instruction fine-tuning.
- Classification fine-tuning involves replacing the output layer of an LLM via a small classification layer.
- In the case of classifying text messages as "spam" or "not spam," the new classification layer consists of only two output nodes. Previously, we used the number of output nodes equal to the number of unique tokens in the vocabulary (i.e., 50,256).
- Instead of predicting the next token in the text as in pretraining, classification fine-tuning trains the model to output a correct class label—for example, "spam" or "not spam."
- The model input for fine-tuning is text converted into token IDs, similar to pretraining.

让我们尝试将这个 classify_review 函数应用于一个示例文本：

```
text_1 = (
    您是一位赢家，您已被特别选中，可领取 1000 美元现金或 2000
美元奖金。


打印(classify_review(
text_1, 模型, 分词器, 设备, 最大长度=train_dataset.max_length)))
```

结果模型正确预测为“垃圾邮件”。让我们再试一个例子：

```
text_2 = (
嗨，只是想确认我们今晚是否还在“晚餐”上？告诉我！



打印(classify_review(
text_2, 模型, 分词器, 设备, 最大长度=train_dataset.max_length)))
```

该模型再次做出正确预测并返回“非垃圾邮件”标签。

最后，让我们保存模型，以防我们以后想要重用模型而无需再次训练。我们可以使用 torch.save 方法：

```
torch.save(model.state_dict(), "review_classifier.pth") 保存模型的状态字典到 "review_classifier.pth" 文件
中
```

一旦保存，模型即可加载：

```
    model_state_dict = torch.load("review_classifier.pth", map_location=device)
model.load_state_dict(model_state_dict)
```

摘要

- 存在不同的策略用于微调LLMs，包括分类微调和指令微调。
- 分类微调涉及通过一个小型分类层替换 LLM 的输出层。
- 在将文本消息分类为“垃圾邮件”或“非垃圾邮件”的情况下，新的分类层仅包含两个输出节点。之前，我们使用的输出节点数量等于词汇表中的唯一标记数量（即 50,256）。
- 与预训练时预测文本中的下一个标记不同，分类微调训练模型输出正确的类别标签——例如，“垃圾邮件”或“非垃圾邮件”。
- 模型微调的输入是将文本转换为标记 ID，类似于预训练。

- Before fine-tuning an LLM, we load the pretrained model as a base model.
- Evaluating a classification model involves calculating the classification accuracy (the fraction or percentage of correct predictions).
- Fine-tuning a classification model uses the same cross entropy loss function as when pretraining the LLM.

- 在微调LLM之前，我们加载预训练模型作为基础模型。
- 评估分类模型涉及计算分类准确率（正确预测的比例或百分比）。

- 微调分类模型使用与预训练时相同的交叉熵损失函数。