7 Fine-tuning to follow instructions

This chapter covers

- The instruction fine-tuning process of LLMs
- Preparing a dataset for supervised instruction fine-tuning
- Organizing instruction data in training batches
- Loading a pretrained LLM and fine-tuning it to follow human instructions
- Extracting LLM-generated instruction responses for evaluation
- Evaluating an instruction-fine-tuned LLM

Previously, we implemented the LLM architecture, carried out pretraining, and imported pretrained weights from external sources into our model. Then, we focused on fine-tuning our LLM for a specific classification task: distinguishing between spam and non-spam text messages. Now we'll implement the process for fine-tuning an LLM to follow human instructions, as illustrated in figure 7.1. Instruction fine-tuning is one of the main techniques behind developing LLMs for chatbot applications, personal assistants, and other conversational tasks.

微调以遵循说明

本章涵盖

- LLMs的指令微调过程
- 准备用于监督指令微调的数据集
- 组织训练批次中的指令数据
- 加载预训练的LLM并将其微调以遵循人类指令
- 提取LLM生成的指令响应以进行评估
- 评估一个指令微调的LLM

之前，我们实现了LLM架构，进行了预训练，并将外部来源的预训练权重导入到我们的模型中。然后，我们专注于微调我们的LLM以完成特定的分类任务：区分垃圾邮件和非垃圾邮件短信。现在，我们将实现微调LLM以遵循人类指令的过程，如图7.1 所示。指令微调是开发LLMs用于聊天机器人应用、个人助理和其他对话任务背后的主要技术之一。

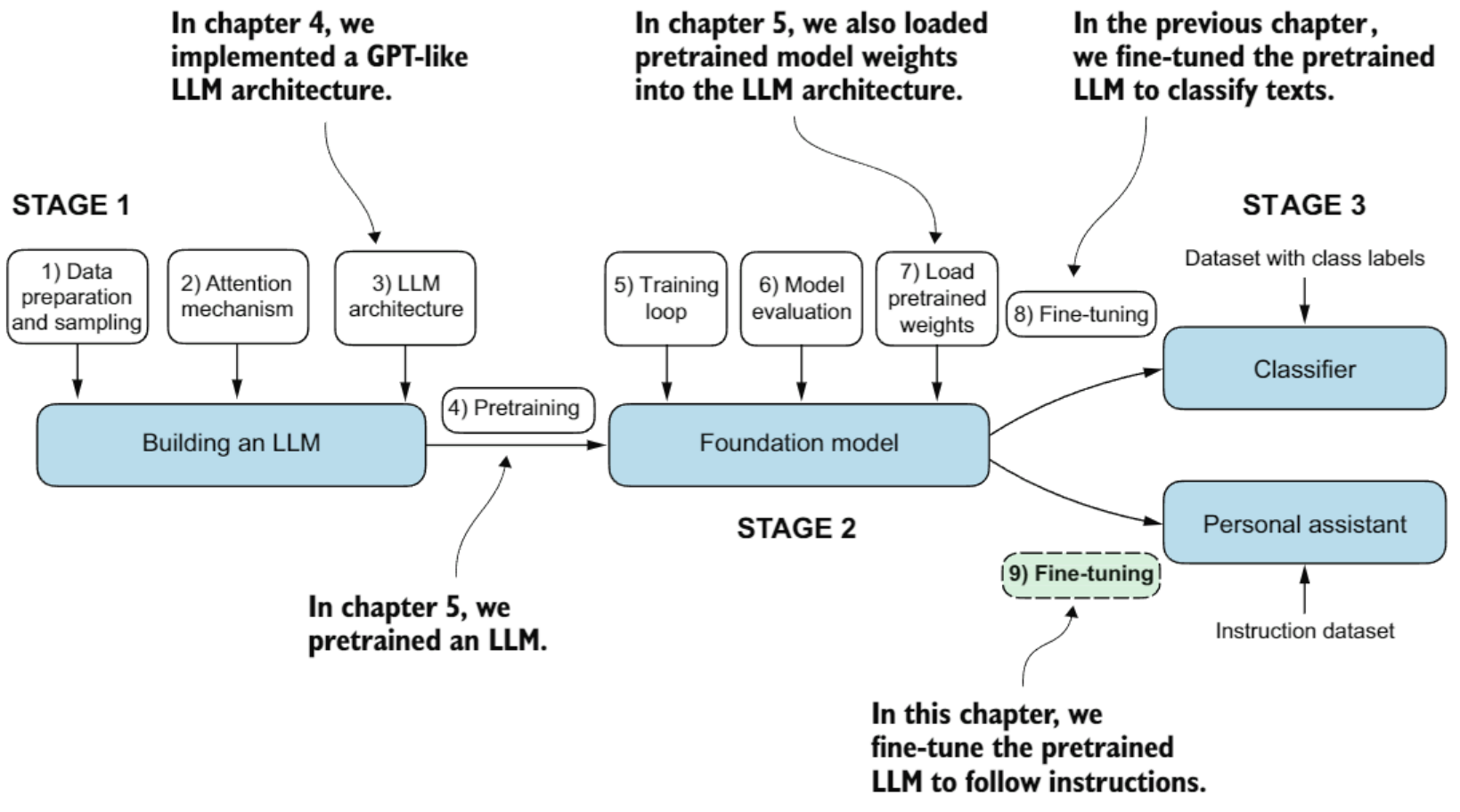

Figure 7.1 The three main stages of coding an LLM. This chapter focuses on step 9 of stage 3: fine-tuning a pretrained LLM to follow human instructions.

Figure 7.1 shows two main ways of fine-tuning an LLM: fine-tuning for classification (step 8) and fine-tuning an LLM to follow instructions (step 9). We implemented step 8 in chapter 6. Now we will fine-tune an LLM using an *instruction dataset*.

7.1 Introduction to instruction fine-tuning

We now know that pretraining an LLM involves a training procedure where it learns to generate one word at a time. The resulting pretrained LLM is capable of *text completion*, meaning it can finish sentences or write text paragraphs given a fragment as input. However, pretrained LLMs often struggle with specific instructions, such as "Fix the grammar in this text" or "Convert this text into passive voice." Later, we will examine a concrete example where we load the pretrained LLM as the basis for *instruction fine-tuning*, also known as *supervised instruction fine-tuning*.

Here, we focus on improving the LLM's ability to follow such instructions and generate a desired response, as illustrated in figure 7.2. Preparing the dataset is a key aspect of instruction fine-tuning. Then we'll complete all the steps in the three stages of the instruction fine-tuning process, beginning with the dataset preparation, as shown in figure 7.3.

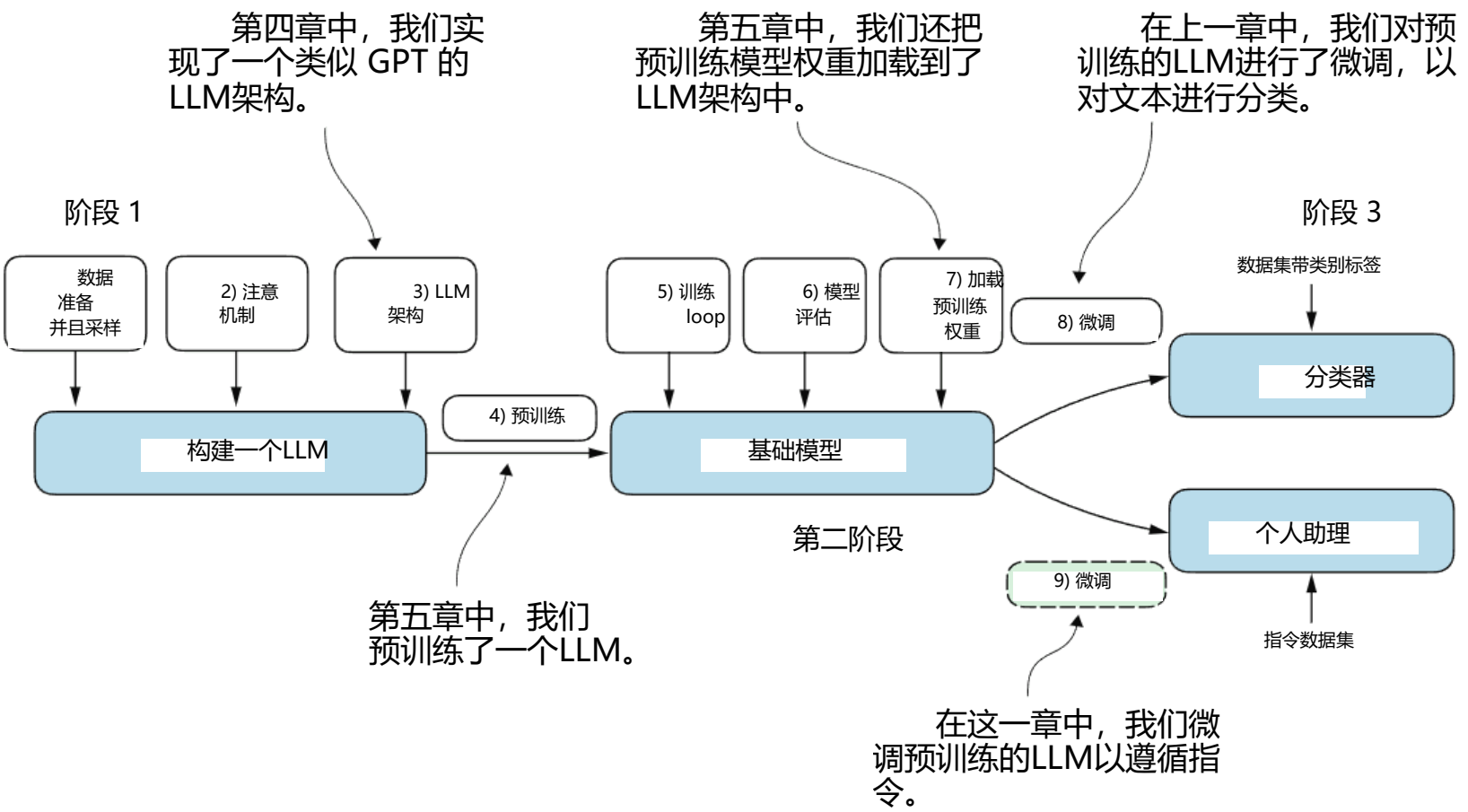

图 7.1 LLM编码的三个主要阶段。本章重点介绍第三阶段第 9 步：微调预训练的LLM以遵循人类指令。

图 7.1 展示了两种主要的微调方法：用于分类的微调（步骤 8）和用于遵循指令的微调LLM（步骤 9）。我们在第 6 章实现了步骤 8。现在我们将使用指令数据集对LLM进行微调。

7.1 指令微调简介

我们现在知道，预训练一个LLM涉及一个训练过程，在这个过程中它学会一次生成一个单词。结果预训练的LLM能够进行文本补全，这意味着它可以根据输入的片段完成句子或撰写文本段落。然而，预训练的LLMs往往难以处理特定的指令，例如“修正这段文本的语法”或“将这段文本改为被动语态”。稍后，我们将考察一个具体示例，其中我们将预训练的LLM作为指令微调的基础，也称为监督指令微调。

这里，我们专注于提高LLM遵循此类指令并生成所需响应的能力，如图 7.2 所示。准备数据集是指令微调的关键方面。然后我们将完成指令微调过程的三个阶段的所有步骤，从数据集准备开始，如图 7.3 所示。

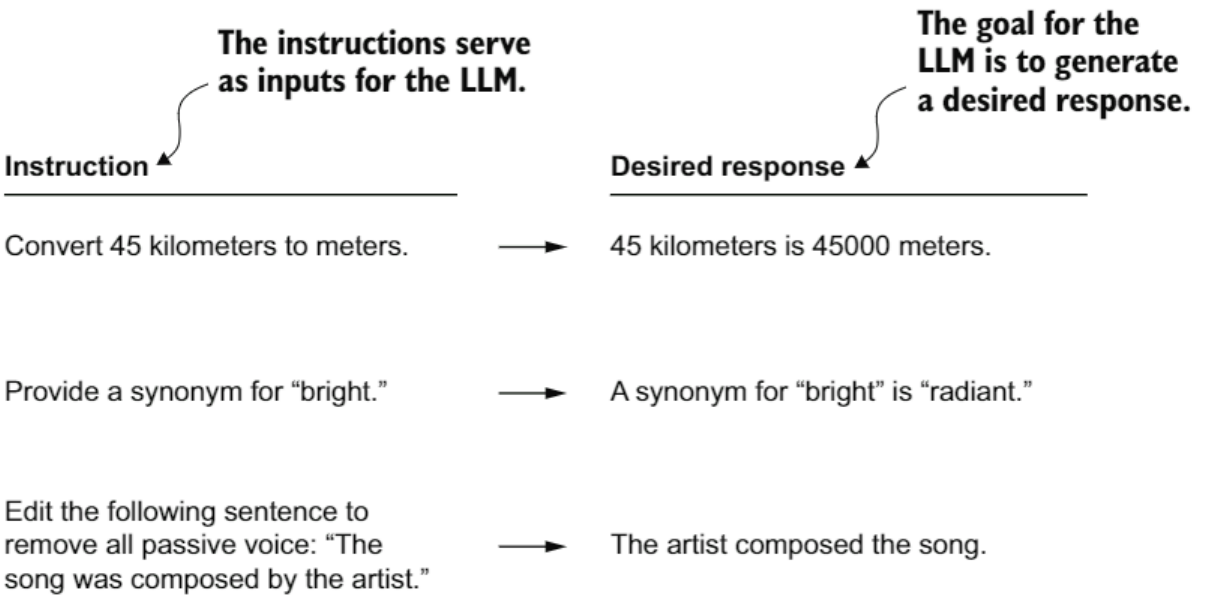

Figure 7.2 Examples of instructions that are processed by an LLM to generate desired responses

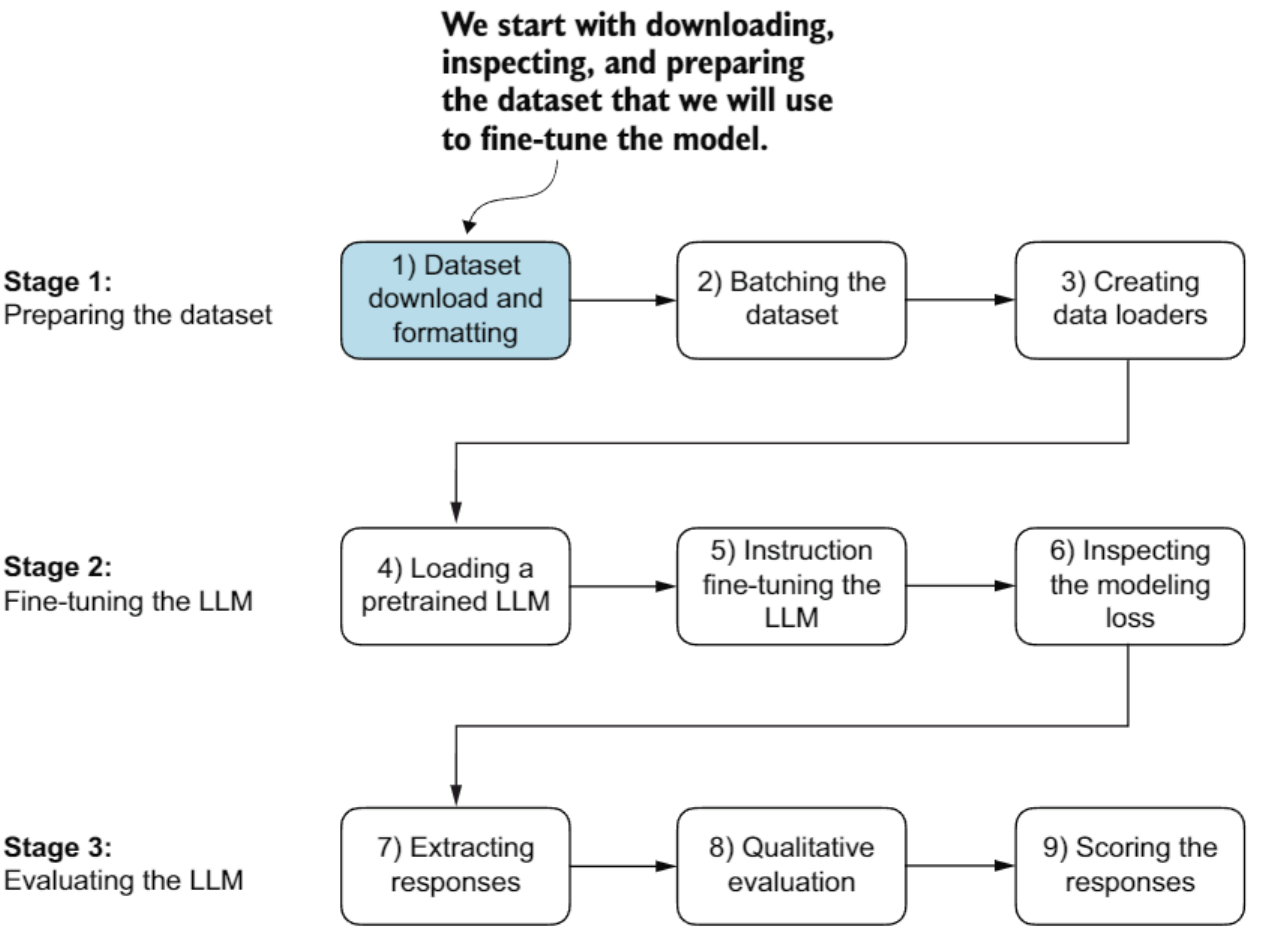

Figure 7.3 The three-stage process for instruction fine-tuning an LLM. Stage 1 involves dataset preparation, stage 2 focuses on model setup and fine-tuning, and stage 3 covers the evaluation of the model. We will begin with step 1 of stage 1: downloading and formatting the dataset.

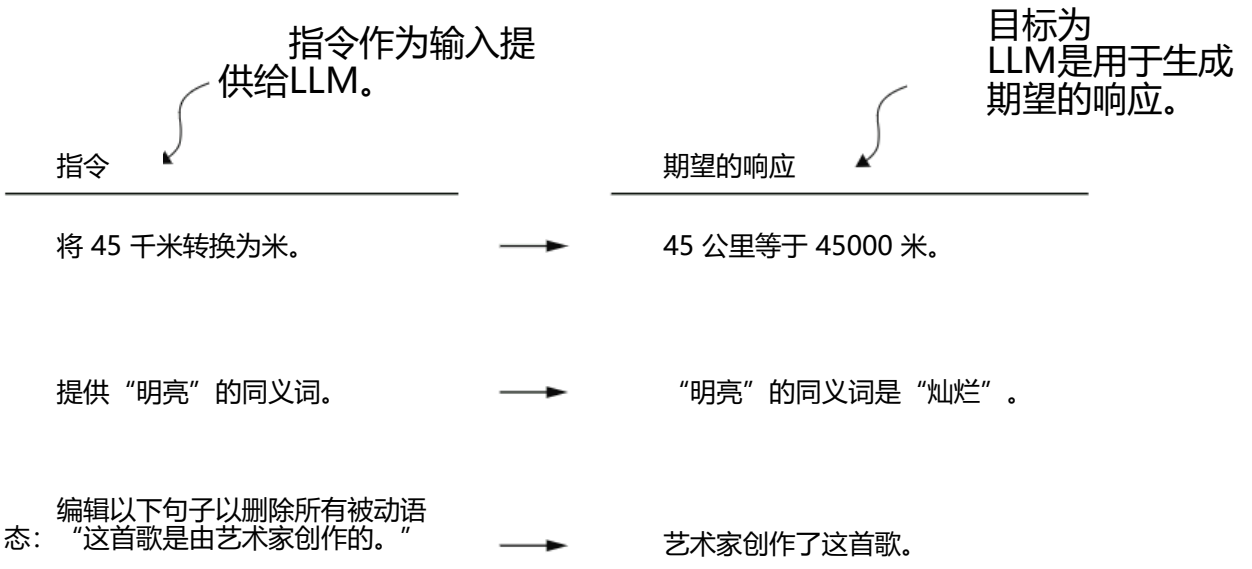

图 7.2 LLM处理以生成所需响应的指令示例

我们首先下载、检查和准备我们将用于微调模型的训练集。

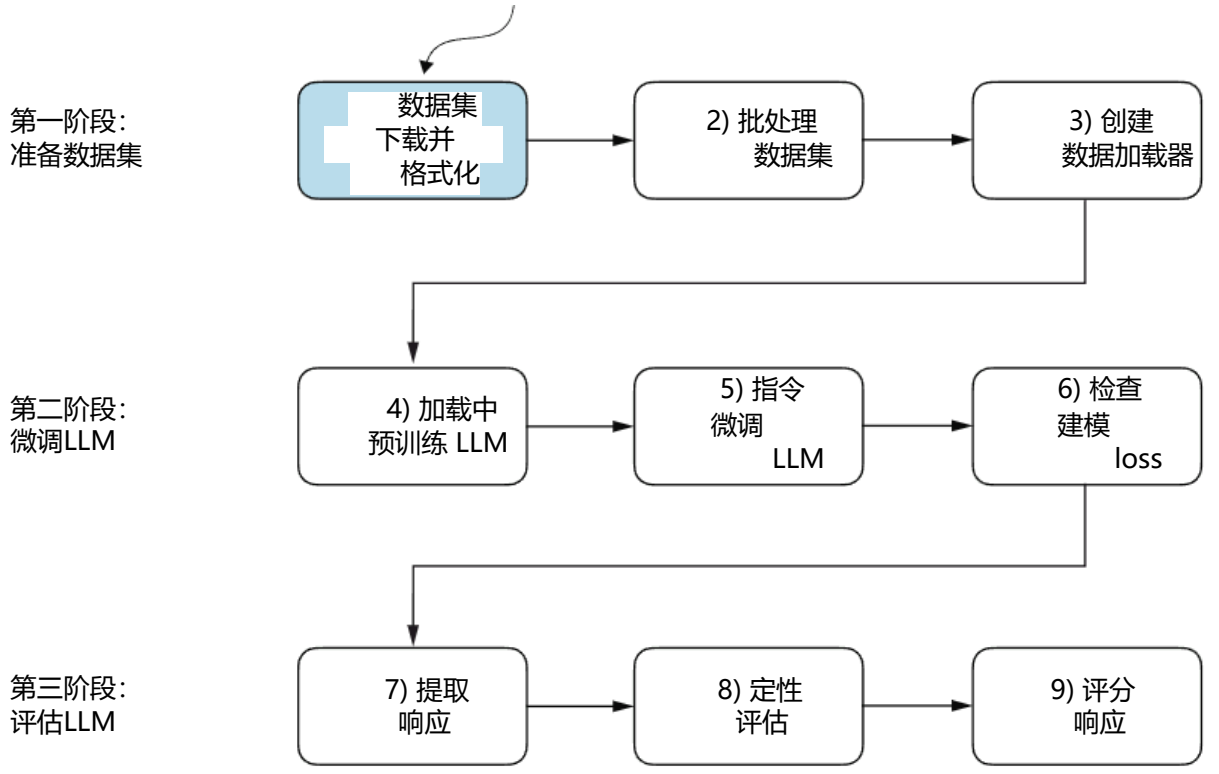

图 7.3 指令微调的三阶段过程 LLM。第一阶段包括数据集准备，第二阶段侧重于模型设置和微调，第三阶段涵盖模型的评估。我们将从第一阶段的第一步开始：下载和

格式化数据集。

7.2 Preparing a dataset for supervised instruction fine-tuning

Let's download and format the instruction dataset for instruction fine-tuning a pre-trained LLM. The dataset consists of 1,100 *instruction–response pairs* similar to those in figure 7.2. This dataset was created specifically for this book, but interested readers can find alternative, publicly available instruction datasets in appendix B.

The following code implements and executes a function to download this dataset, which is a relatively small file (only 204 KB) in JSON format. JSON, or JavaScript Object Notation, mirrors the structure of Python dictionaries, providing a simple structure for data interchange that is both human readable and machine friendly.

Listing 7.1 Downloading the dataset

```
import json
import os
import urllib

def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else:
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

file_path = "instruction-data.json"
url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
    "/main/ch07/01_main-chapter-code/instruction-data.json"
)


data = download_and_load_file(file_path, url)
print("Number of entries:", len(data))
```

Skips download if file was already downloaded

The output of executing the preceding code is

```
Number of entries: 1100
```

The `data` list that we loaded from the JSON file contains the 1,100 entries of the instruction dataset. Let's print one of the entries to see how each entry is structured:

```
print("Example entry:\n", data[50])
```

7.2 准备用于监督指令微调的数据集

让我们下载并格式化用于微调预训练模型LLM的指令数据集。该数据集包含 1,100 个类似于图 7.2 中的指令-响应对。该数据集专门为本书创建，但感兴趣的读者可以在附录 B 中找到其他可公开获取的指令数据集。

以下代码实现并执行了一个下载此数据集的函数，该数据集是一个相对较小的文件（仅 204 KB），格式为 JSON。JSON，或称 JavaScript 对象表示法，与 Python 字典的结构相似，提供了一种简单、易于人类阅读和机器处理的数据交换结构。

列表 7.1 下载数据集

```
导入 json
import os
导入     urllib

def 下载并加载文件（文件路径，网址）：
     如果 os.path.exists(file_path) 不存在
          使用 urllib.request.urlopen(url) 作为响应：
          text_data = response.read().decode("utf-8") 使用
      open(file_path, "w", encoding="utf-8") 作为文件
               file.write(text_data) # 写入文本数据到文件
          否则：
               with open(file_path, "r", 编码="utf-8") as file:
          file.read()text_data =
     with
     open(file_path
     "r") data = json.load(file)
     返回 as file:
     文件读取() 使用
     open(file_path,
    file_path 作为文件："instruction-data.json" url
= (
    "https://raw.githubusercontent.com/rasbt/LLMs-从零开始"
"/main/ch07/01_主要章节代码/指令数据.json" )


    data = 下载并加载文件(file_path, url) 打印("条目数
量：", len(data))
```

跳过下载文件已下载

执行上一段代码的输出是

```
数字     of  条目：      1100
```

我们从 JSON 文件中加载的数据列表包含了指令数据集的 1,100 个条目。让我们打印其中一个条目，以查看每个条目的结构：

```
打印("示例条目：\n",)          data[50]
```

The content of the example entry is

```
Example entry:
 {'instruction': 'Identify the correct spelling of the following word.',
  'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}
```

As we can see, the example entries are Python dictionary objects containing an `'instruction'`, `'input'`, and `'output'`. Let's take a look at another example:

```
print("Another example entry:\n", data[999])
```

Based on the contents of this entry, the `'input'` field may occasionally be empty:

```
Another example entry:
 {'instruction': "What is an antonym of 'complicated'?",
  'input': '',
  'output': "An antonym of 'complicated' is 'simple'."}
```

Instruction fine-tuning involves training a model on a dataset where the input-output pairs, like those we extracted from the JSON file, are explicitly provided. There are various methods to format these entries for LLMs. Figure 7.4 illustrates two different

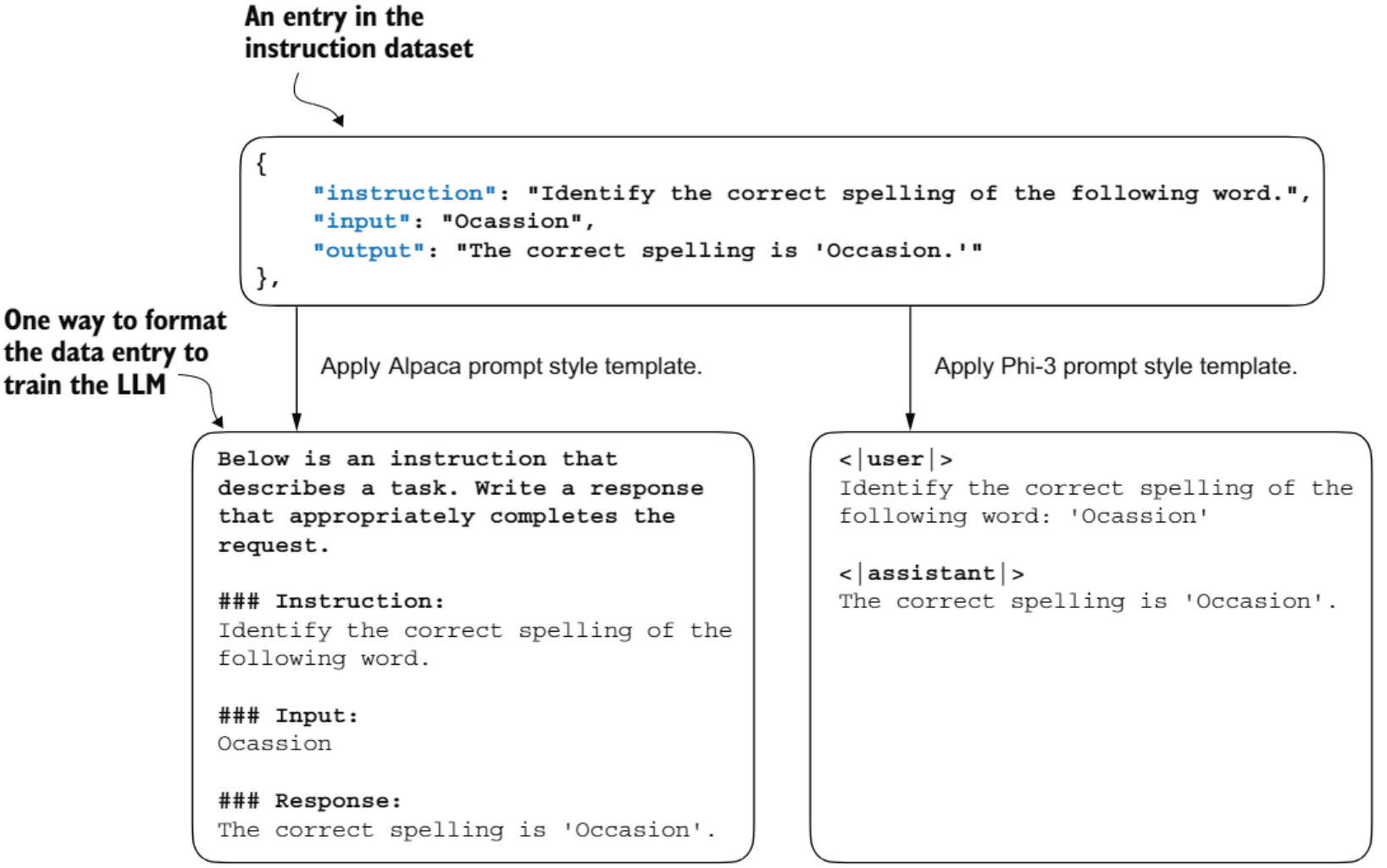

Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

示例条目内容的翻译为：示例条目内容的翻译

```
示例条目：{'指令'：'识别以下单词的正确拼写。'}

  输入：'inpu机会          'output'：″正确的拼写是'Occasion。'″
```

我们可以看到，示例条目是包含 'instruction'、'input' 和 'output' 的 Python 字典对象。让我们看看另一个示例：

```
打印("另一个示例")          entry:\n", 数据[999])
```

基于本条目内容，'input'字段有时可能为空：

```
另一个示例条目：{'instruction'：″‘complicated’的反义词是什
么？″}
  '输入'：''
  输出：″An″      'complicated'的反义词是          简单。″}
```

指令微调涉及在一个数据集上训练模型，其中输入-输出对，如我们从 JSON 文件中提取的，被明确提供。有各种方法来格式化这些条目为LLMs。图 7.4 说明了两种不同的

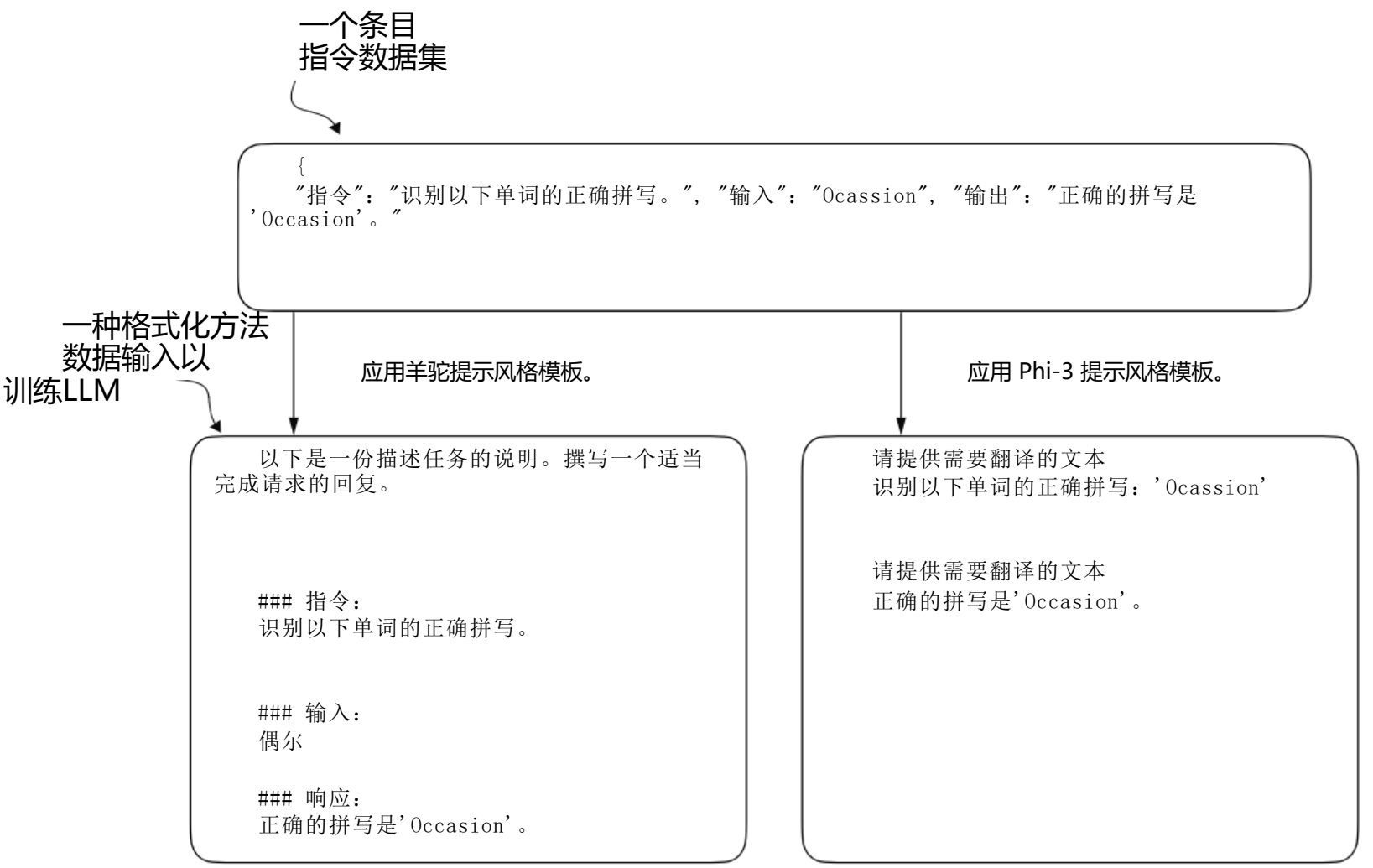

图 7.4 LLMs中指令微调的提示风格比较。Alpaca 风格（左侧）使用具有定义明确的指令、输入和响应部分的格式，而 Phi-3 风格（右侧）采用带有指定

example formats, often referred to as *prompt styles*, used in the training of notable LLMs such as Alpaca and Phi-3.

Alpaca was one of the early LLMs to publicly detail its instruction fine-tuning process. Phi-3, developed by Microsoft, is included to demonstrate the diversity in prompt styles. The rest of this chapter uses the Alpaca prompt style since it is one of the most popular ones, largely because it helped define the original approach to fine-tuning.

Exercise 7.1 Changing prompt styles

After fine-tuning the model with the Alpaca prompt style, try the Phi-3 prompt style shown in figure 7.4 and observe whether it affects the response quality of the model.

Let's define a `format_input` function that we can use to convert the entries in the `data` list into the Alpaca-style input format.

Listing 7.2 Implementing the prompt formatting function

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

This `format_input` function takes a dictionary `entry` as input and constructs a formatted string. Let's test it to dataset entry `data[50]`, which we looked at earlier:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

The formatted input looks like as follows:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Identify the correct spelling of the following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion.'
```

示例格式，通常称为提示风格，用于训练 Alpaca 和 Phi-3 等知名LLMs的培训。

羊驼是早期公开详细说明其指令微调过程的LLMs之一。微软开发的 Phi-3 被包括在内，以展示提示风格的多样性。本章的其余部分使用 Alpaca 提示风格，因为它是最受欢迎的之一，很大程度上是因为它帮助定义了原始的微调方法。

练习 7.1 更改提示样式

在用 Alpaca 提示风格微调模型后，尝试使用图 7.4 所示的 Phi-3 提示风格，并观察它是否影响模型响应质量。

让我们定义一个 format_input 函数，我们可以用它来转换条目 data 将列表转换为 Alpaca 风格的输入格式。

列表 7.2 实现提示格式化函数

```
def 格式化输入(entry):
    instruction_text = (
    以下是一个描述任务的说明。编写一个适当完成请求的回复。### 说明：
{entry['instruction']}
    输入文本翻译为简体中文为：（请提供需要翻译的文本内容）
    f"\n\n### 输入：\n{entry['input']}" if entry["input"] else "" ) 返回
instruction_text + input_text
```

这个 format_input 函数接收一个字典条目作为输入并构建一个格式化字符串。让我们测试它对数据集条目 data[50]进行格式化，这是我们之前看过的：

```
model_input = 格式化输入(data[50]) desired_response = f"\n\n### 响应：
\n{data[50]['output']}" 打印(model_input + desired_response)
```

格式化输入看起来如下所示：

```
以下是描述一项任务的说明。编写一个适当完成请求的回复。


### 指示：识别正确的
                        以下拼写                        单词。

### 输入：
偶尔

### 响应：
正确        拼写      is 场合。
```

Note that the `format_input` skips the optional `### Input:` section if the `'input'` field is empty, which we can test out by applying the `format_input` function to entry `data[999]` that we inspected earlier:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

The output shows that entries with an empty `'input'` field don't contain an `### Input:` section in the formatted input:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
What is an antonym of 'complicated'?

### Response:
An antonym of 'complicated' is 'simple'.
```

Before we move on to setting up the PyTorch data loaders in the next section, let's divide the dataset into training, validation, and test sets analogous to what we have done with the spam classification dataset in the previous chapter. The following listing shows how we calculate the portions.

Listing 7.3 Partitioning the dataset

```
train_portion = int(len(data) * 0.85)
test_portion = int(len(data) * 0.1)
val_portion = len(data) - train_portion - test_portion

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

Use 85% of the data for training

Use 10% for testing

Use remaining 5% for validation

This partitioning results in the following dataset sizes:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```

注意，如果 'input' 字段为空，format_input 格式将跳过可选的### 输入：部分，我们可以通过将 format_input 函数应用于条目来测试这一点

data[999] 我们之前检查过的：

```
model_input = 格式化输入(data[999]) desired_response = f"\n\n### 响应：
\n{data[999]['output']}" 打印(model_input + desired_response)
```

输出显示，空 '输入' 字段的条目不包含###

输入：输入部分在格式化输入中：

```
以下是一个描述任务的说明。编写一个适当完成请求的回复。


### 指令：
什么是 'complicated'的反义词？

### 响应：
反义词 of 复杂 是 '简单'。
```

在下一节中设置 PyTorch 数据加载器之前，让我们将数据集划分为训练集、验证集和测试集，类似于我们在上一章中对垃圾邮件分类数据集所做的那样。以下列表显示了我们是如何计算这些部分的。

列表 7.3 数据集划分

使用 85%的数据进行训练

使用 10% 测试

使用剩余 5%进行验证

```
训练部分 = int(len(data) * 0.85) 测试部分
= int(len(data) * 0.1) 验证部分 =
data 长度 - 训练部分 - 测试部分

train_data = data[:train_portion] test_data =
data[train_portion:test_portion + test_portion] val_data =
data[train_portion + test_portion:]


打印("训练集长度：", len(train_data)) 打印("验证集长
度：", len(val_data)) 打印("测试集长度：",
len(test_data))
```

这个分区结果导致以下数据集大小：

```
训练集长度：935 验证集长
度：55 测试集长度：110
```

Having successfully downloaded and partitioned the dataset and gained a clear understanding of the dataset prompt formatting, we are now ready for the core implementation of the instruction fine-tuning process. Next, we focus on developing the method for constructing the training batches for fine-tuning the LLM.

7.3 Organizing data into training batches

As we progress into the implementation phase of our instruction fine-tuning process, the next step, illustrated in figure 7.5, focuses on constructing the training batches effectively. This involves defining a method that will ensure our model receives the formatted training data during the fine-tuning process.

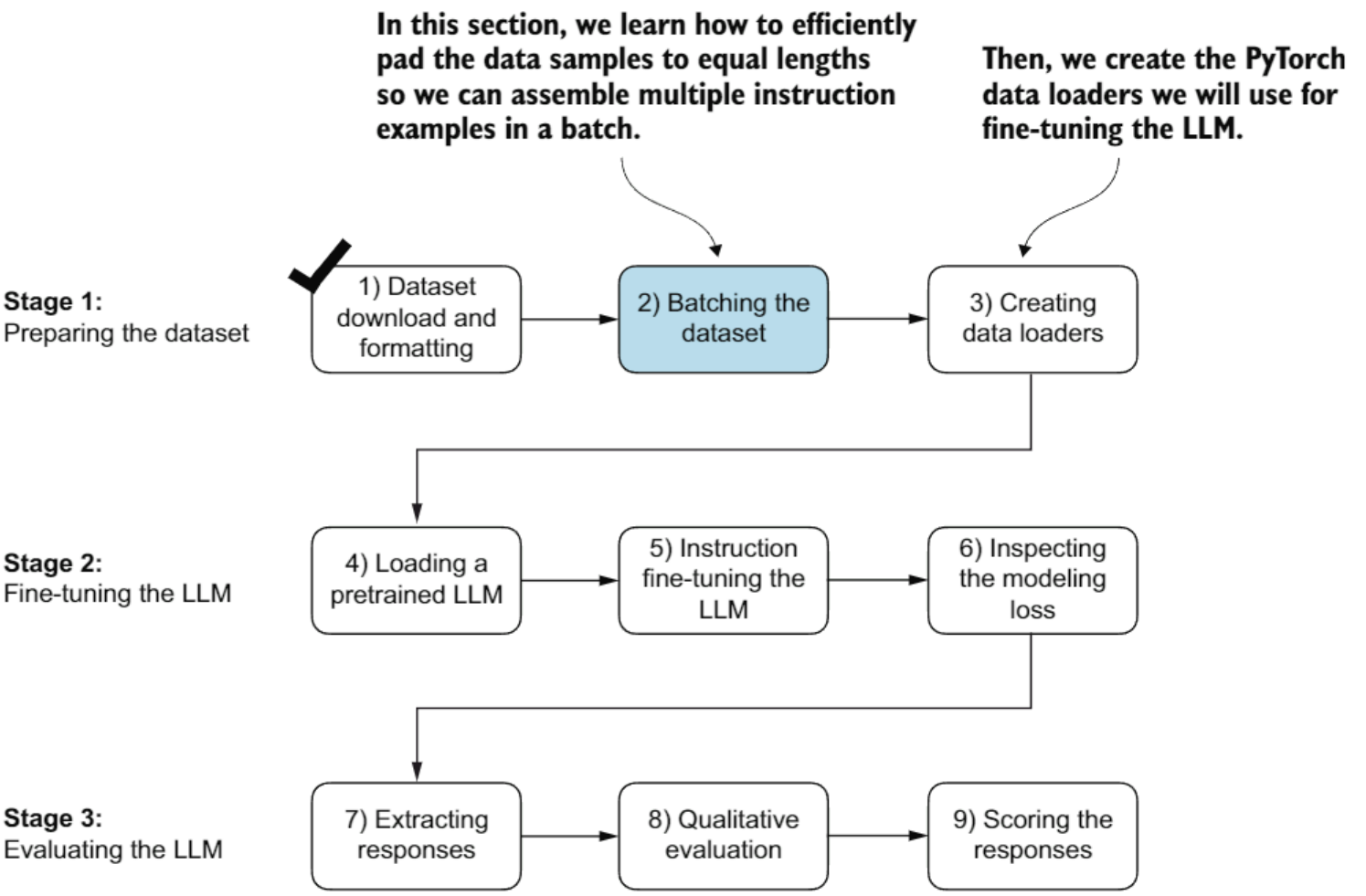

Figure 7.5 The three-stage process for instruction fine-tuning an LLM. Next, we look at step 2 of stage 1: assembling the training batches.

In the previous chapter, the training batches were created automatically by the PyTorch `DataLoader` class, which employs a default *collate* function to combine lists of samples into batches. A collate function is responsible for taking a list of individual data samples and merging them into a single batch that can be processed efficiently by the model during training.

However, the batching process for instruction fine-tuning is a bit more involved and requires us to create our own custom collate function that we will later plug into

成功下载并分区数据集，并对数据集提示格式有了清晰理解后，我们现在准备进行指令微调的核心实现。接下来，我们专注于开发用于微调的LLM的训练批次构建方法。

7.3 组织数据为训练批次

随着我们进入指令微调过程的实施阶段，下一步，如图 7.5 所示，重点在于有效地构建训练批次。这涉及到定义一种方法，确保我们的模型在微调过程中接收格式化的训练数据。

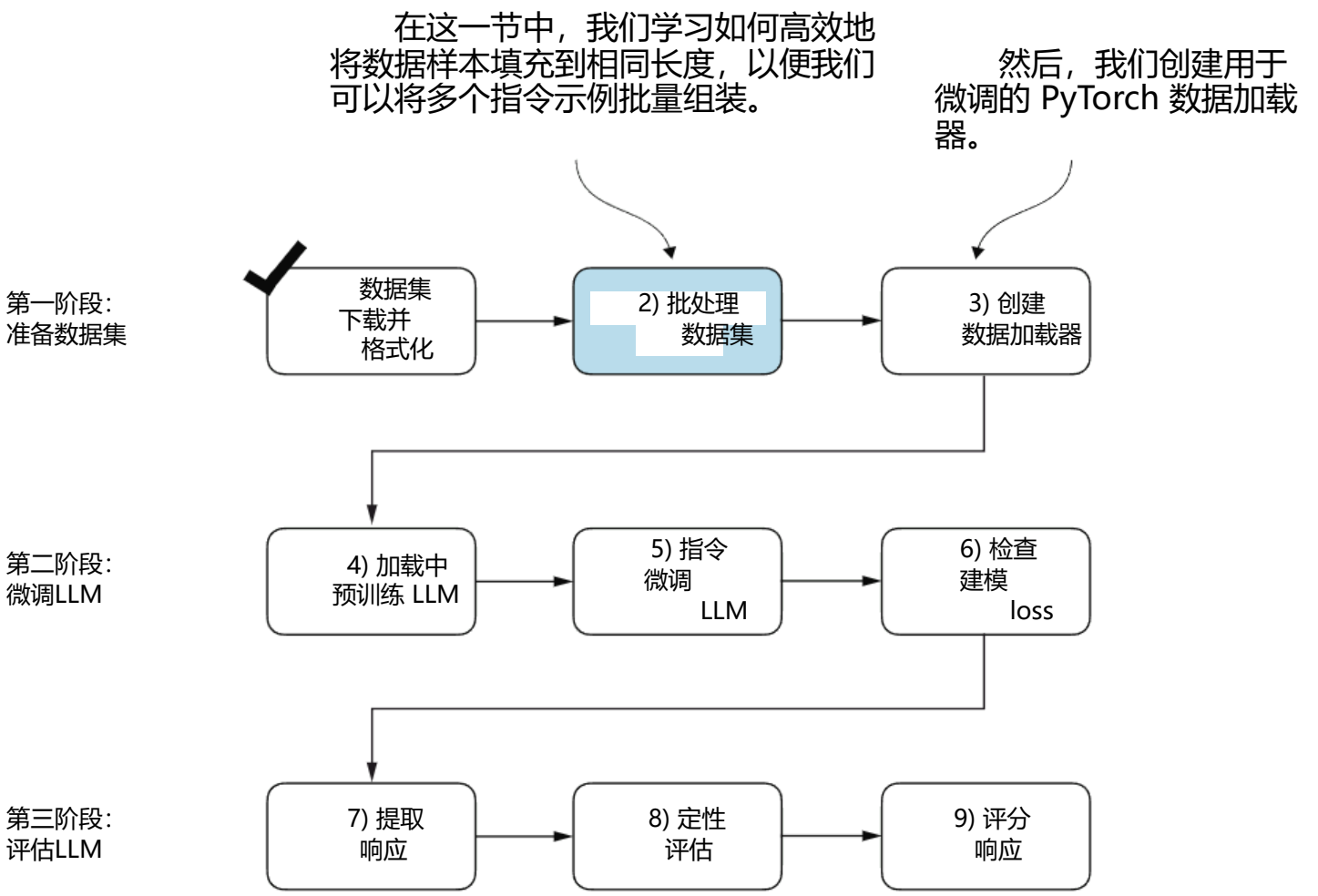

图 7.5 指令微调的三阶段过程 LLM。接下来，我们来看第一阶段第二步：组装训练批次。

在上一章中，训练批次是由 PyTorch DataLoader 类自动创建的，该类使用默认的 collate 函数将样本列表组合成批次。collate 函数负责将单个数据样本的列表合并成一个可以由模型在训练过程中高效处理的单个批次。

然而，指令微调的批处理过程稍微复杂一些，需要我们创建自己的自定义合并函数，稍后将其插入

the `DataLoader`. We implement this custom collate function to handle the specific requirements and formatting of our instruction fine-tuning dataset.

Let's tackle the *batching process* in several steps, including coding the custom collate function, as illustrated in figure 7.6. First, to implement steps 2.1 and 2.2, we code an `InstructionDataset` class that applies `format_input` and *pretokenizes* all inputs in the dataset, similar to the `SpamDataset` in chapter 6. This two-step process, detailed in figure 7.7, is implemented in the `__init__` constructor method of the `InstructionDataset`.

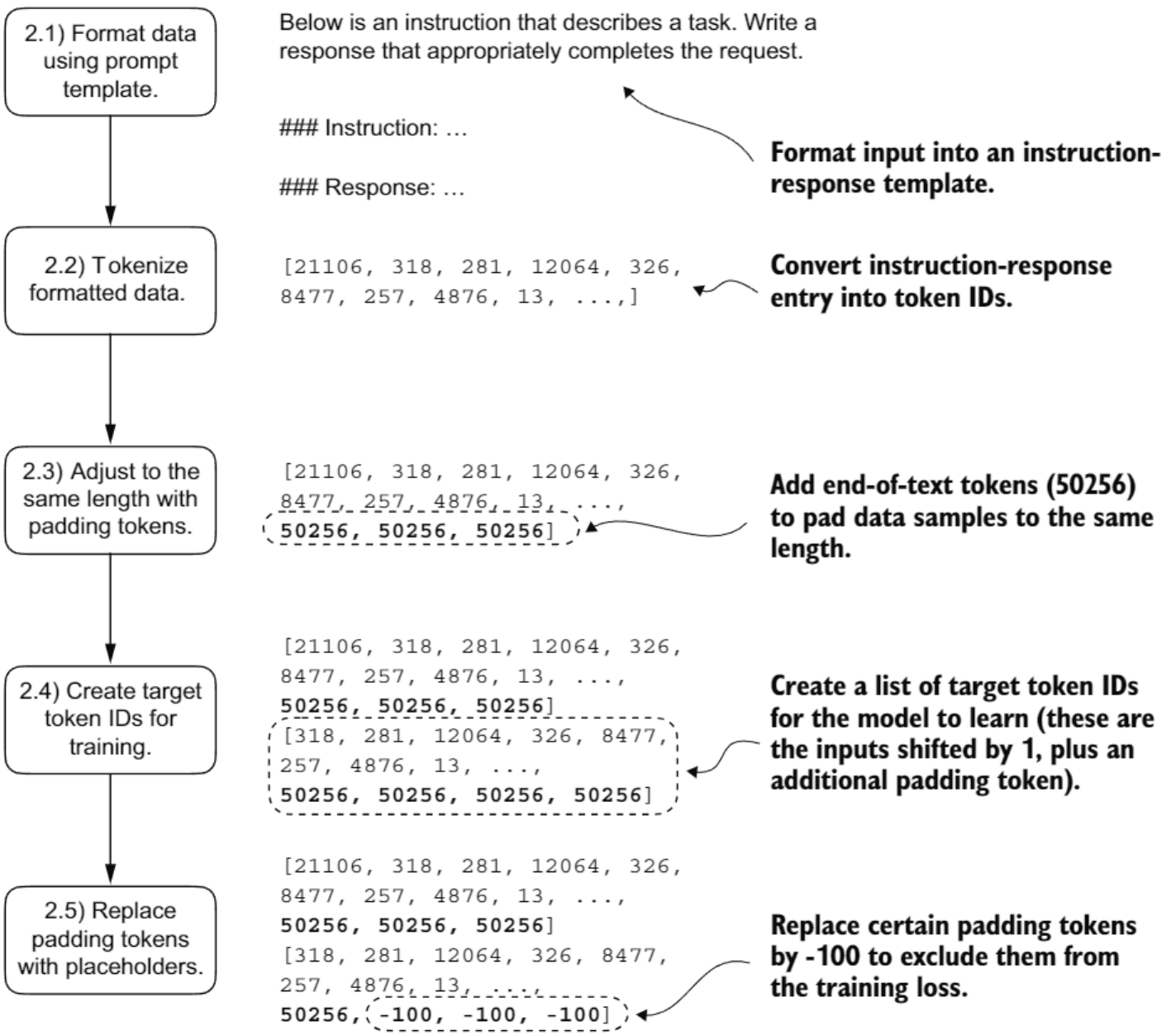

Figure 7.6 The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing `-100` placeholder tokens to mask padding tokens in the loss function.

指令微调的批处理过程稍微复杂一些，需要我们创建自己的自定义 collate 函数，稍后将其插入到 DataLoader 中。我们实现这个自定义 collate 函数来处理我们指令微调数据集的特定要求和格式。

让我们分几个步骤来处理批处理过程，包括编写自定义的 collate 函数，如图 7.6 所示。首先，为了实现步骤 2.1 和 2.2，我们编写了一个 InstructionDataset 类，该类应用 format_input 并预处理数据集中的所有输入，类似于第 6 章中的 SpamDataset。如图 7.7 所示，这个两步过程在__init__构造方法中实现。

指令数据集。

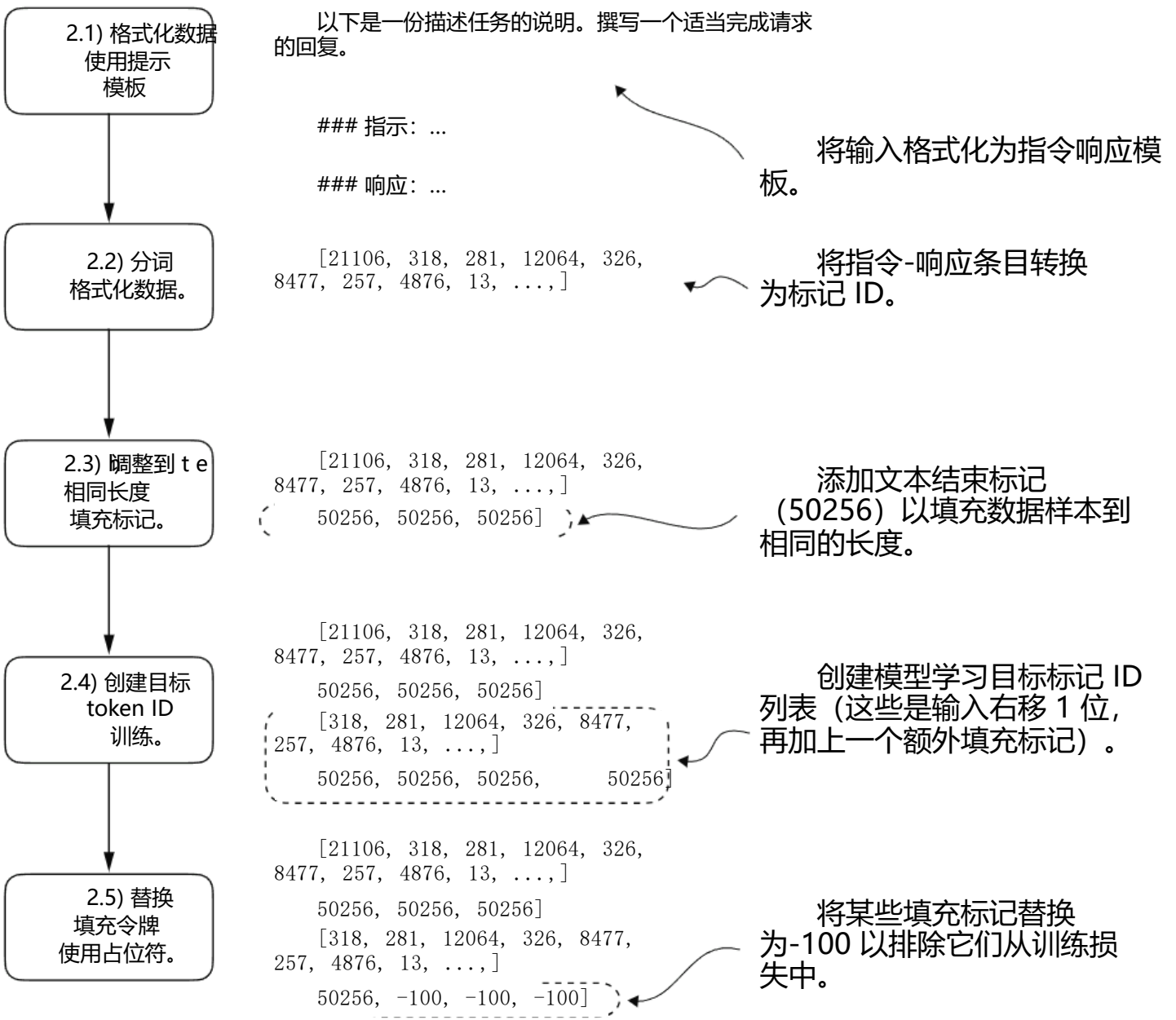

图 7.6 实施批处理过程的五个子步骤：（2.1）应用提示模板，（2.2）使用前几章中的分词，（2.3）添加填充标记，（2.4）创建目标标记 ID，（2.5）将-100 占位符标记替换为在损失函数中掩盖填充标记。

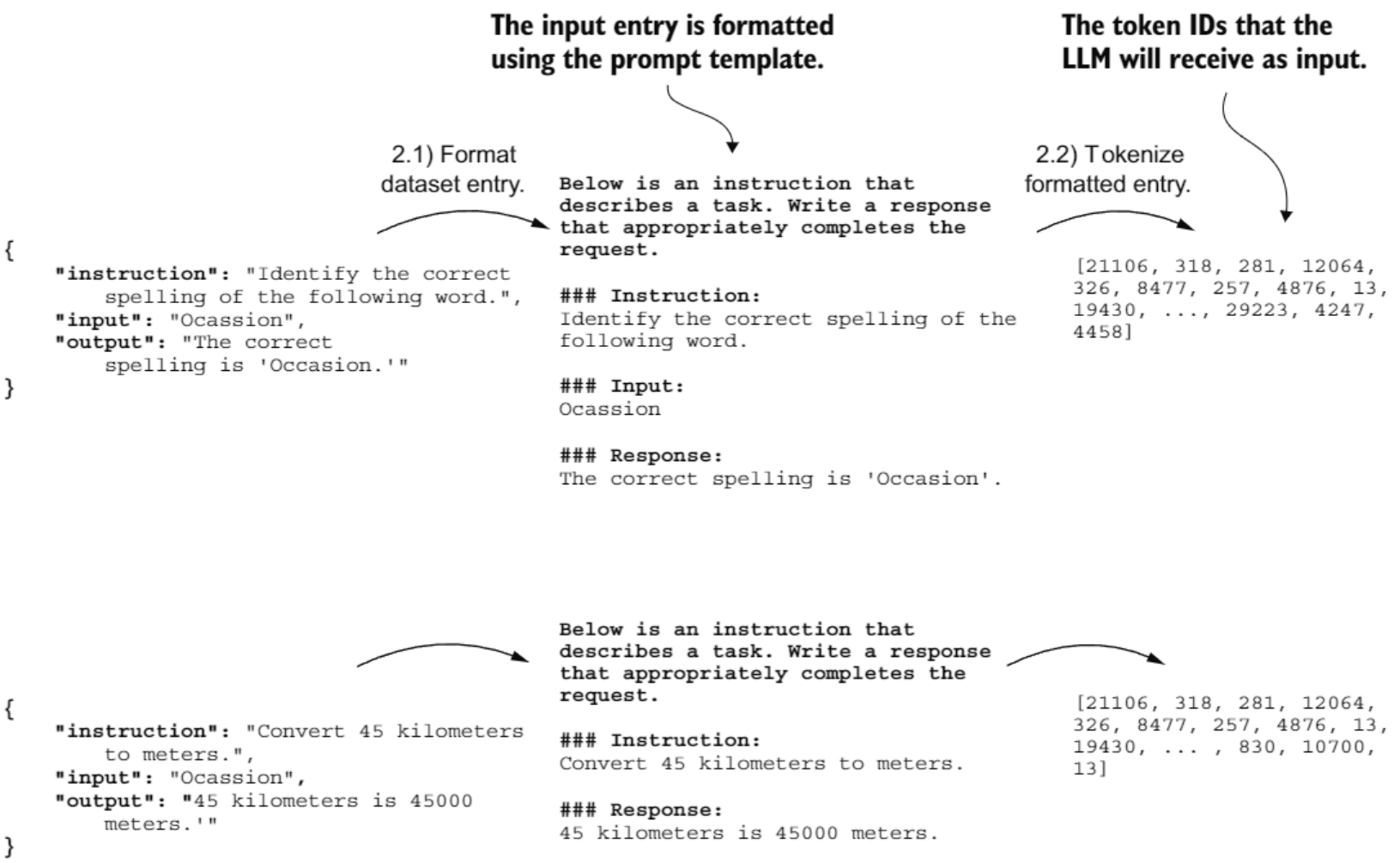

Figure 7.7 The first two steps involved in implementing the batching process. Entries are first formatted using a specific prompt template (2.1) and then tokenized (2.2), resulting in a sequence of token IDs that the model can process.

Listing 7.4 Implementing an instruction dataset class

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

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data:                                    # Pretokenizes texts
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )

    def __getitem__(self, index):
        return self.encoded_texts[index]

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

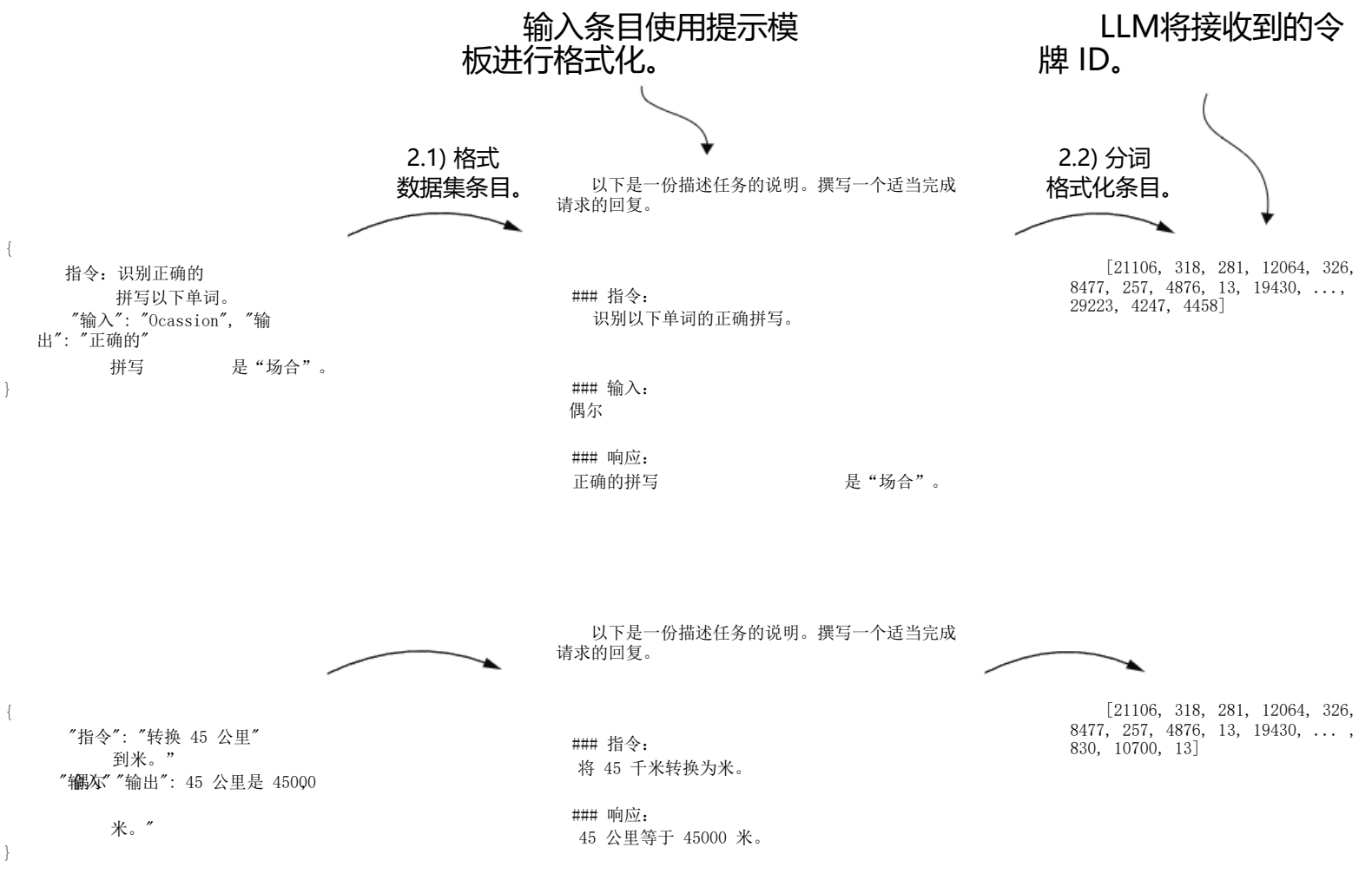

图 7.7 实施批处理过程的前两个步骤。首先使用特定的提示模板（2.1）对条目进行格式化，然后进行分词（2.2），从而得到模型可以处理的 token ID 序列。

列表 7.4 实现指令数据集类

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

class InstructionDataset(数据集):
     def __init__(self, 数据, 分词器):
          self.data = data
      self.encoded_texts = [] 对于
      entry in data:
                指令加输入 = 格式化输入条目 响应文本 = f"\n\n### 响应：
           \n{entry['输出']}" 全文 = 指令加输入 + 响应文本
           self.encoded_texts.append()


               tokenizer.encode(完整文本)


     def __getitem__(self, 索引):
          返回 self.encoded_texts[index]

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

预处理文本

Similar to the approach used for classification fine-tuning, we want to accelerate training by collecting multiple training examples in a batch, which necessitates padding all inputs to a similar length. As with classification fine-tuning, we use the `<|endoftext|>` token as a padding token.

Instead of appending the `<|endoftext|>` tokens to the text inputs, we can append the token ID corresponding to `<|endoftext|>` to the pretokenized inputs directly. We can use the tokenizer's `.encode` method on an `<|endoftext|>` token to remind us which token ID we should use:

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

The resulting token ID is `50256`.

Moving on to step 2.3 of the process (see figure 7.6), we adopt a more sophisticated approach by developing a custom collate function that we can pass to the data loader. This custom collate function pads the training examples in each batch to the same length while allowing different batches to have different lengths, as demonstrated in figure 7.8. This approach minimizes unnecessary padding by only extending sequences to match the longest one in each batch, not the whole dataset.

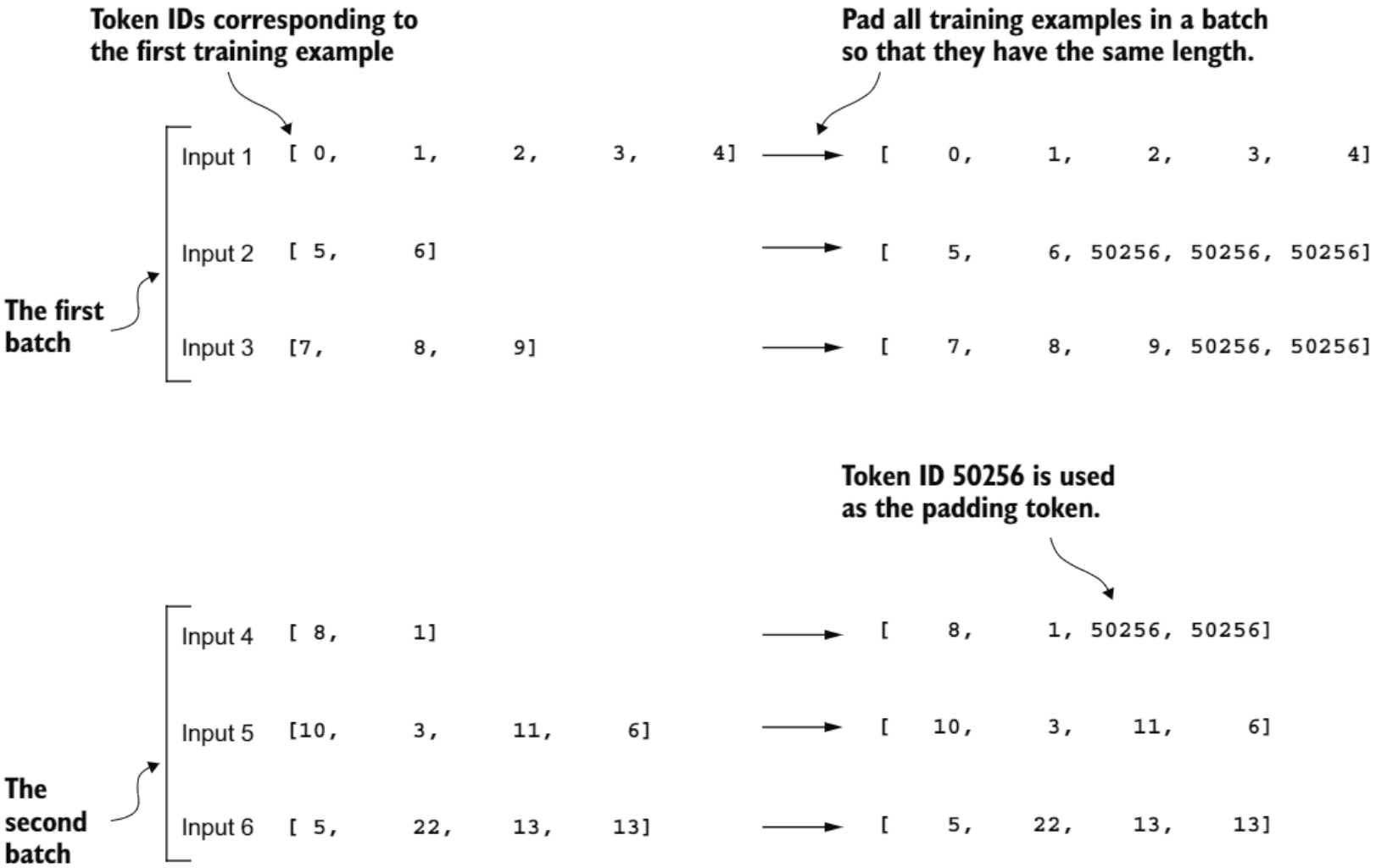

Figure 7.8 The padding of training examples in batches using token ID `50256` to ensure uniform length within each batch. Each batch may have different lengths, as shown by the first and second.

与用于分类微调的方法类似，我们希望通过收集多个训练示例到一个批次中来加速训练，这需要将所有输入填充到相似长度。与分类微调一样，我们使用`token`作为填充标记。

将 token 对应的 ID 直接附加到预分词输入中，而不是将 token 附加到文本输入中。我们可以使用 tokenizer 的.encode 方法对一个 token 进行编码，以提醒我们应使用哪个 token ID：

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

结果令牌 ID 为 50256。

继续到过程的第 2.3 步（见图 7.6），我们通过开发一个自定义的 collate 函数来采用更复杂的方法，可以将该函数传递给数据加载器。这个自定义的 collate 函数将每个批次的训练示例填充到相同的长度，同时允许不同的批次有不同的长度，如图 7.8 所示。这种方法通过仅将序列扩展到匹配每个批次中最长的序列，而不是整个数据集，从而最小化了不必要的填充。

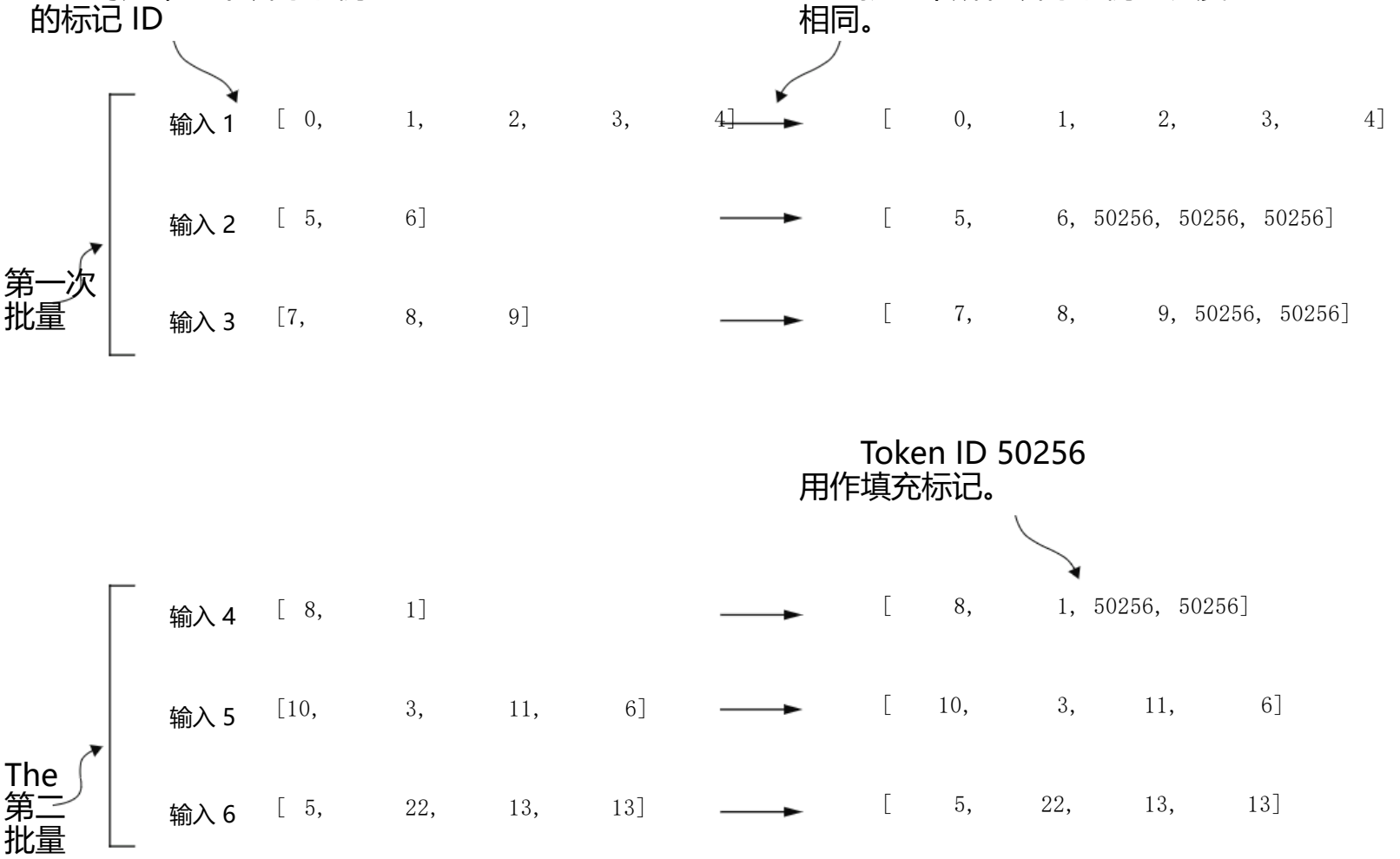

图 7.8 使用 token ID 50256 对训练样本进行批处理填充，以确保每个批次内的长度一致。每个批次可能具有不同的长度，如第一和第二个所示。

We can implement the padding process with a custom collate function:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst = []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]


        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    return inputs_tensor
```

Finds the longest sequence in the batch

Pads and prepares inputs

Removes extra padded token added earlier

Converts the list of inputs to a tensor and transfers it to the target device

The `custom_collate_draft_1` we implemented is designed to be integrated into a PyTorch `DataLoader`, but it can also function as a standalone tool. Here, we use it independently to test and verify that it operates as intended. Let's try it on three different inputs that we want to assemble into a batch, where each example gets padded to the same length:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
    inputs_3
)
print(custom_collate_draft_1(batch))
```

The resulting batch looks like the following:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
```

This output shows all inputs have been padded to the length of the longest input list, `inputs_1`, containing five token IDs.

我们可以使用自定义的 collate 函数来实现填充过程：

```
def custom_collate_draft_1(
    批量
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
inputs_lst = []


    for item in 批次:
        new_item = item.copy()
     new_item       += [填充令牌 ID]

     新项 = 项目.copy() 新项

         填充 = (
         new_item + [填充令牌 ID] *
     (batch_max_length - len(new_item)) ) inputs =
     torch.tensor(padded[:-1])
     inputs_lst.append(inputs)



    inputs_tensor = torch.stack(inputs_lst).to(设备) return
inputs_tensor
```

找到最长的
序列在
批量

垫子和
准备输入

移除额外
填充标记
添加了较早

将输入列表转
换为张量并将其传
输到目标设备

我们实现的 custom_collate_draft_1 是为了集成到 PyTorch DataLoader 中，但它也可以作为一个独立的工具使用。在这里，我们独立使用它来测试和验证它是否按预期运行。让我们尝试将其应用于三个不同的输入，这些输入是我们想要组装成批次的，其中每个示例都被填充到相同的长度：

```
    inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6] inputs_3 = [7,
8, 9] batch = (


        inputs_1,
        inputs_2
        输入_3
    ) 打印(custom_collate_draft_1(batch))
```

结果批次看起来如下：

```
    张量([[         0,       1,       2,       3,       4],
             [ 5,  6, 50256, 50256, 50256], [ 7,  8,]
                                    9, 50256, 50256]]]
```

此输出显示所有输入都填充到了最长输入列表 inputs_1 的长度，包含五个标记 ID。

We have just implemented our first custom collate function to create batches from lists of inputs. However, as we previously learned, we also need to create batches with the target token IDs corresponding to the batch of input IDs. These target IDs, as shown in figure 7.9, are crucial because they represent what we want the model to generate and what we need during training to calculate the loss for the weight updates. That is, we modify our custom collate function to return the target token IDs in addition to the input token IDs.

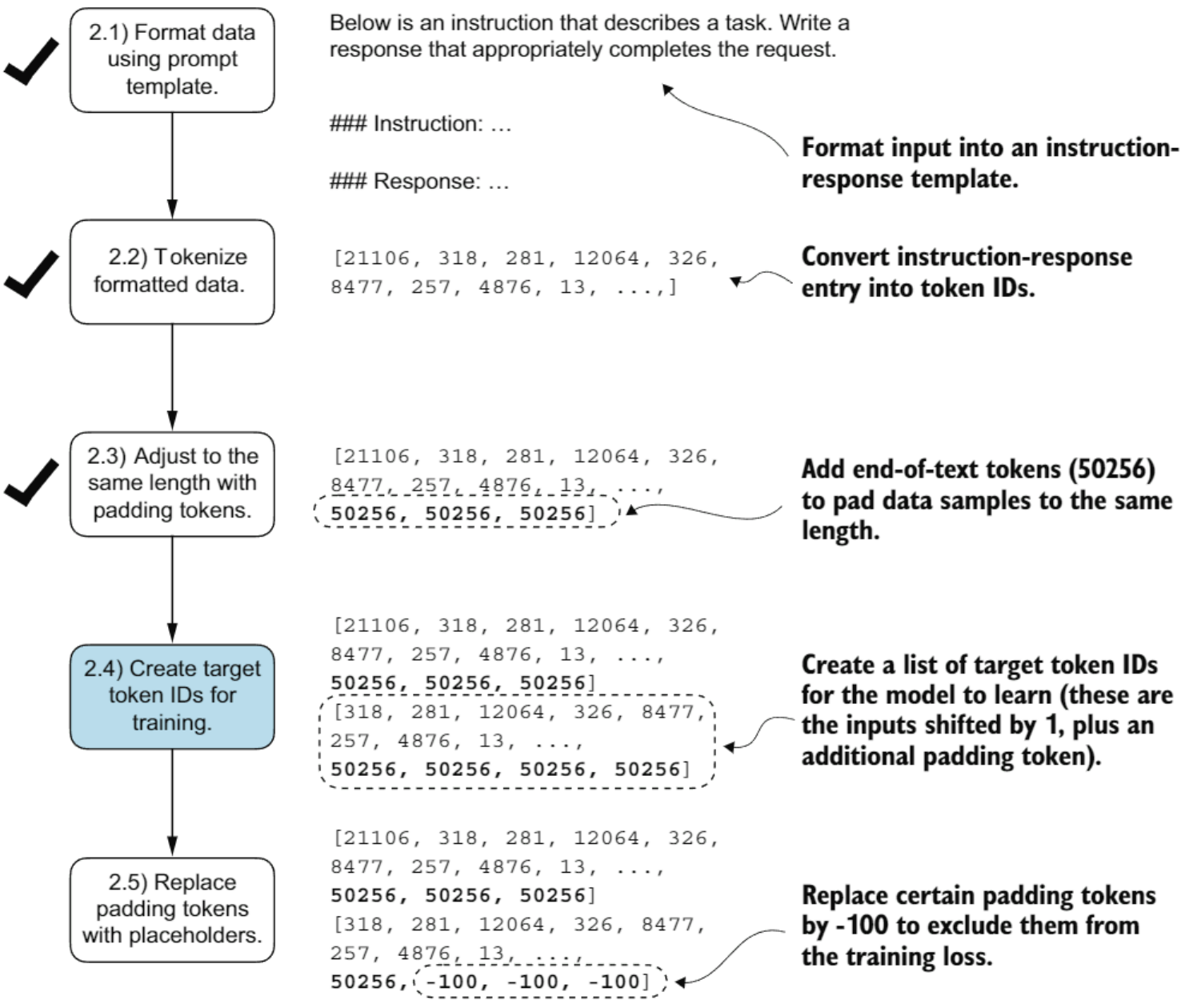

Figure 7.9 The five substeps involved in implementing the batching process. We are now focusing on step 2.4, the creation of target token IDs. This step is essential as it enables the model to learn and predict the tokens it needs to generate.

Similar to the process we used to pretrain an LLM, the target token IDs match the input token IDs but are shifted one position to the right. This setup, as shown in figure 7.10, allows the LLM to learn how to predict the next token in a sequence.

我们刚刚实现了我们的第一个自定义排序函数，用于从输入列表中创建批次。然而，正如我们之前所学的，我们还需要创建与输入 ID 批次相对应的目标标记 ID 的批次。这些目标 ID，如图 7.9 所示，至关重要，因为它们代表了我们希望模型生成的内容，以及在训练期间计算损失以更新权重所需的内容。也就是说，我们修改了我们的自定义排序函数，使其除了返回输入标记 ID 之外，还返回目标标记 ID。

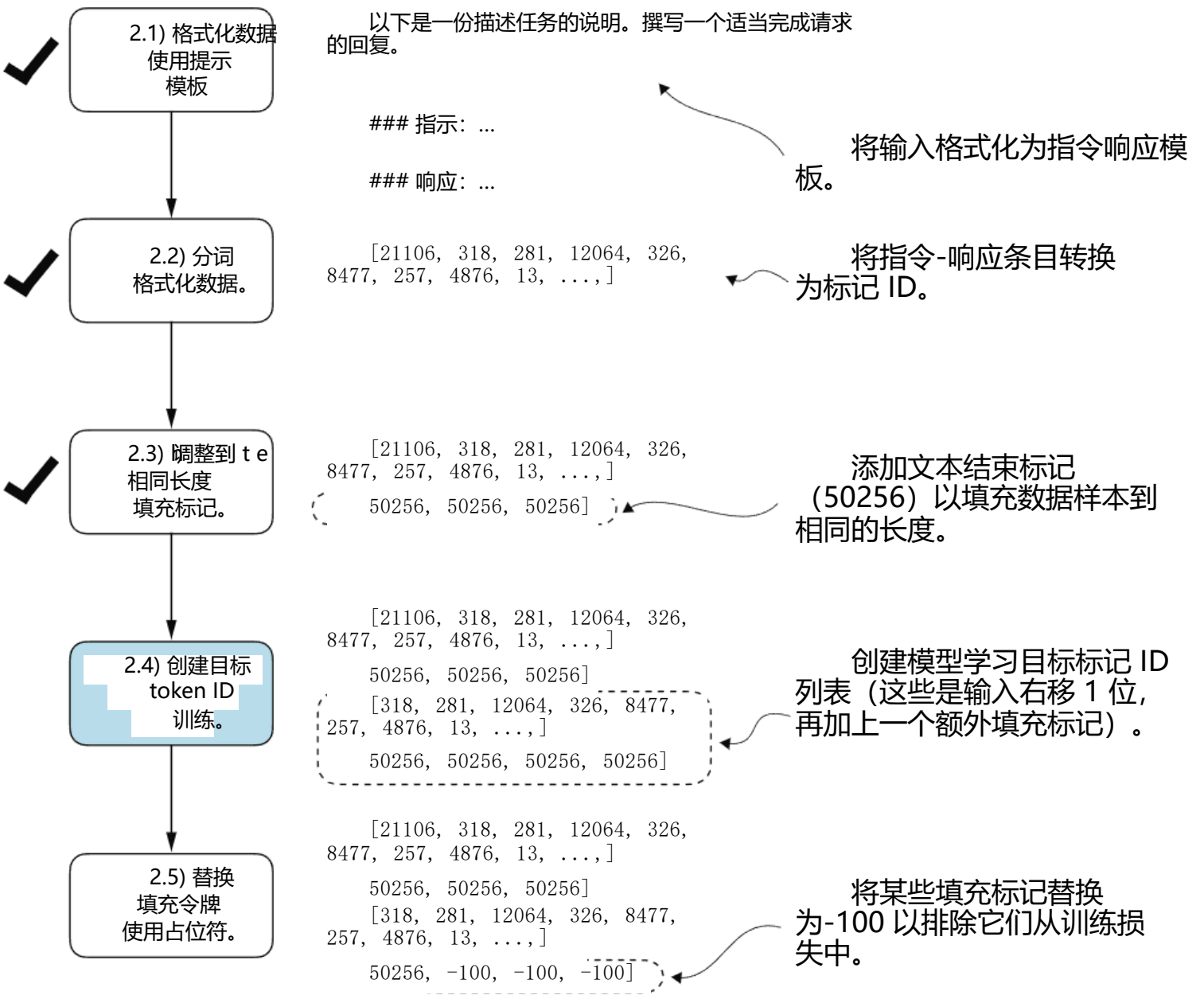

图 7.9 实施批处理过程的五个子步骤。我们现在专注于第 2.4 步，即创建目标标记 ID。这一步至关重要，因为它使模型能够学习和预测它需要生成的标记。

与我们在预训练中使用的过程类似，目标标记 ID 与输入标记 ID 匹配，但向右移动一个位置。如图 7.10 所示，这种设置允许LLM学习如何预测序列中的下一个标记。

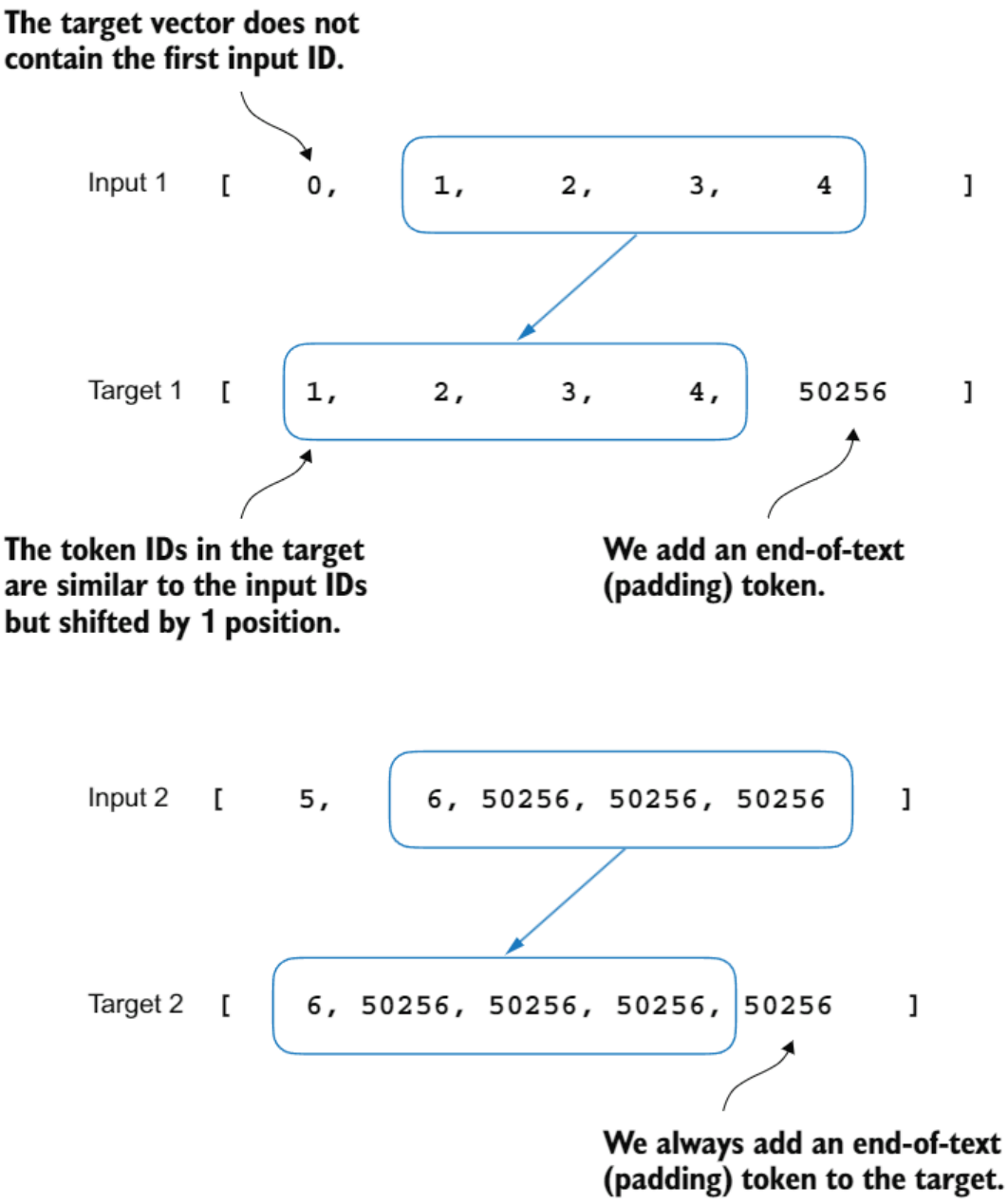

Figure 7.10 The input and target token alignment used in the instruction fine-tuning process of an LLM. For each input sequence, the corresponding target sequence is created by shifting the token IDs one position to the right, omitting the first token of the input, and appending an end-of-text token.

The following updated collate function generates the target token IDs from the input token IDs:

```
def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
```

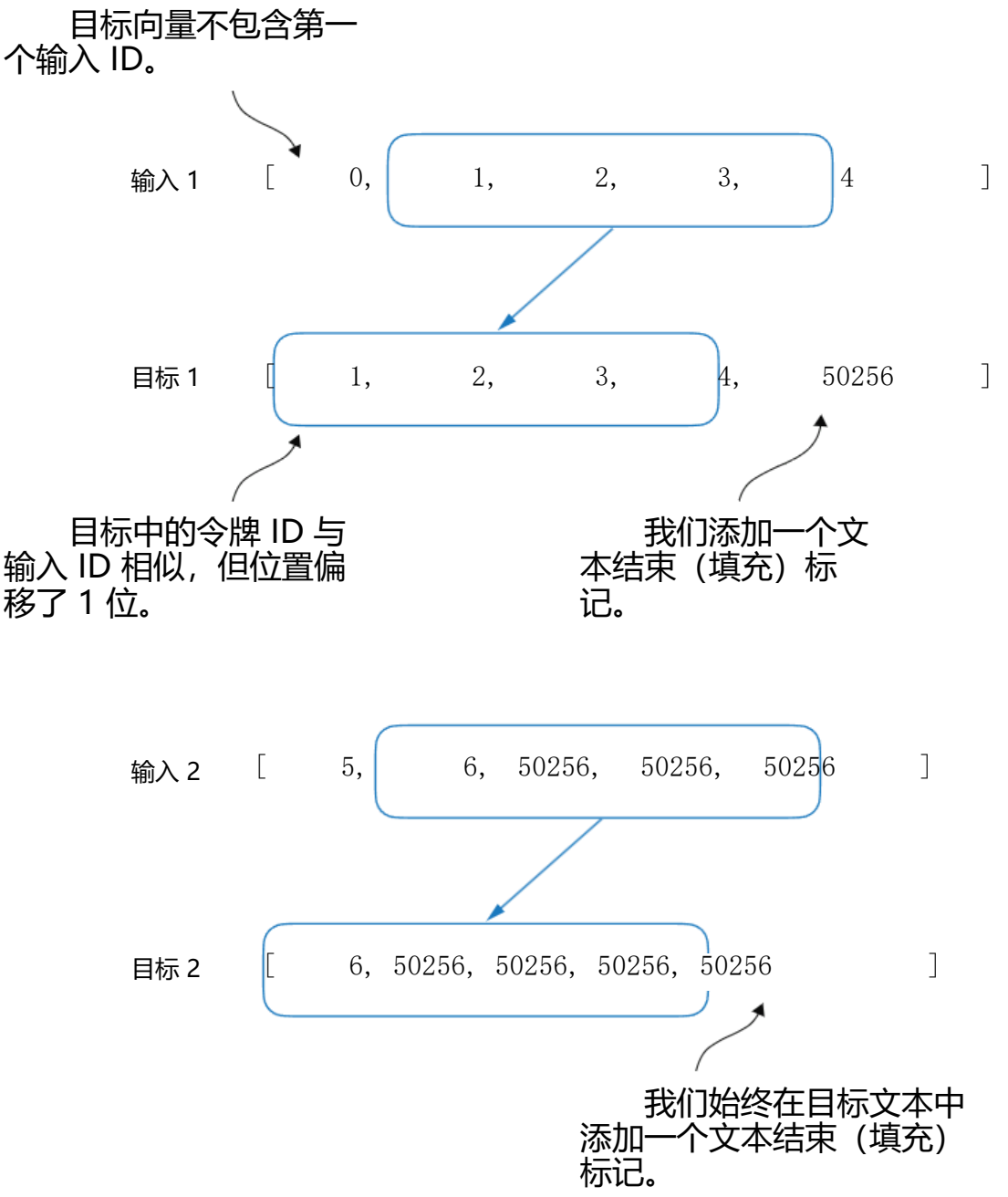

图 7.10 在LLM的指令微调过程中使用的输入和目标标记对齐。对于每个输入序列，通过将标记 ID 向右移动一个位置，省略输入的第一个标记，并添加一个文本结束标记来创建相应的目标序列。

以下更新的 collate 函数从输入 token IDs 生成目标 token IDs：

```
def custom_collate_draft_2(
    批量
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
 inputs_lst, targets_lst = [], []

    for item in 批次:
        new_item = item.copy()
     new_item       += [填充令牌 ID]

     新项 = 项目.copy() 新项
```

```
            padded = (
                new_item + [pad_token_id] *
                (batch_max_length - len(new_item))
            )
            inputs = torch.tensor(padded[:-1])
            targets = torch.tensor(padded[1:])
            inputs_lst.append(inputs)
            targets_lst.append(targets)

        inputs_tensor = torch.stack(inputs_lst).to(device)
        targets_tensor = torch.stack(targets_lst).to(device)
        return inputs_tensor, targets_tensor

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

Truncates the last token for inputs

Shifts +1 to the right for targets

Applied to the example `batch` consisting of three input lists we defined earlier, the new `custom_collate_draft_2` function now returns the input and the target batch:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
tensor([[    1,     2,     3,     4, 50256],
        [    6, 50256, 50256, 50256, 50256],
        [    8,     9, 50256, 50256, 50256]])
```

The first tensor represents inputs.

The second tensor represents the targets.

In the next step, we assign a `-100` placeholder value to all padding tokens, as highlighted in figure 7.11. This special value allows us to exclude these padding tokens from contributing to the training loss calculation, ensuring that only meaningful data influences model learning. We will discuss this process in more detail after we implement this modification. (When fine-tuning for classification, we did not have to worry about this since we only trained the model based on the last output token.)

However, note that we retain one end-of-text token, ID `50256`, in the target list, as depicted in figure 7.12. Retaining it allows the LLM to learn when to generate an end-of-text token in response to instructions, which we use as an indicator that the generated response is complete.

In the following listing, we modify our custom collate function to replace tokens with ID `50256` with `-100` in the target lists. Additionally, we introduce an `allowed_max_length` parameter to optionally limit the length of the samples. This adjustment will be useful if you plan to work with your own datasets that exceed the 1,024-token context size supported by the GPT-2 model.

```
        填充 = (
        new_item + [填充令牌 ID] *
    (batch_max_length - len(new_item)) ) inputs =
    torch.tensor(padded[:-1]) targets =
    torch.tensor(padded[1:])
    inputs_lst.append(inputs)
    targets_lst.append(targets)
```

截断
最后一个标记
输入

Shifts +1 到
适合目标

```
        inputs_tensor = torch.stack(inputs_lst).to(设备)
    targets_tensor = torch.stack(targets_lst).to(设备) 返回
    inputs_tensor, targets_tensor


inputs, targets = custom_collate_draft_2(批次) 打印
(inputs) 打印(targets)
```

应用于我们之前定义的包含三个输入列表的示例批次，新的 custom_collate_draft_2 函数现在返回输入和目标批次：

```
张量([[    0,    1,    2,    3,    4],
       [ 5,  6, ]    50256, 50256, 50256]
       [    7,    8,    9, 50256, 50256]])
张量([[    1,    2,    3,    4, 50256]
       [ 6,  ]  50256, 50256, 50256, 50256]
       [    8,    9, 50256, 50256, 50256]])
```

第一个张量
表示输入。

第二个张量表示
目标。

在下一步中，我们将所有填充标记分配一个-100 占位符值，如图 7.11 所示。这个特殊值允许我们排除这些填充标记对训练损失计算的贡献，确保只有有意义的数据影响模型学习。在实施此修改后，我们将更详细地讨论此过程。（当进行分类微调时，我们不必担心这一点，因为我们只基于最后一个输出标记训练模型。）

然而，请注意，我们在目标列表中保留了最后一个文本结束标记，ID 为 50256，如图 7.12 所示。保留它允许LLM学习何时根据指令生成文本结束标记，我们将其用作生成响应完整的指示器。

在以下列表中，我们修改了自定义的排序函数，将目标列表中 ID 为 50256 的标记替换为-100。此外，我们引入了一个可选的 allowed_max_length 参数，以限制样本的长度。如果您计划使用超出 GPT-2 模型支持的 1,024 个标记上下文大小的自定义数据集，这项调整将非常有用。

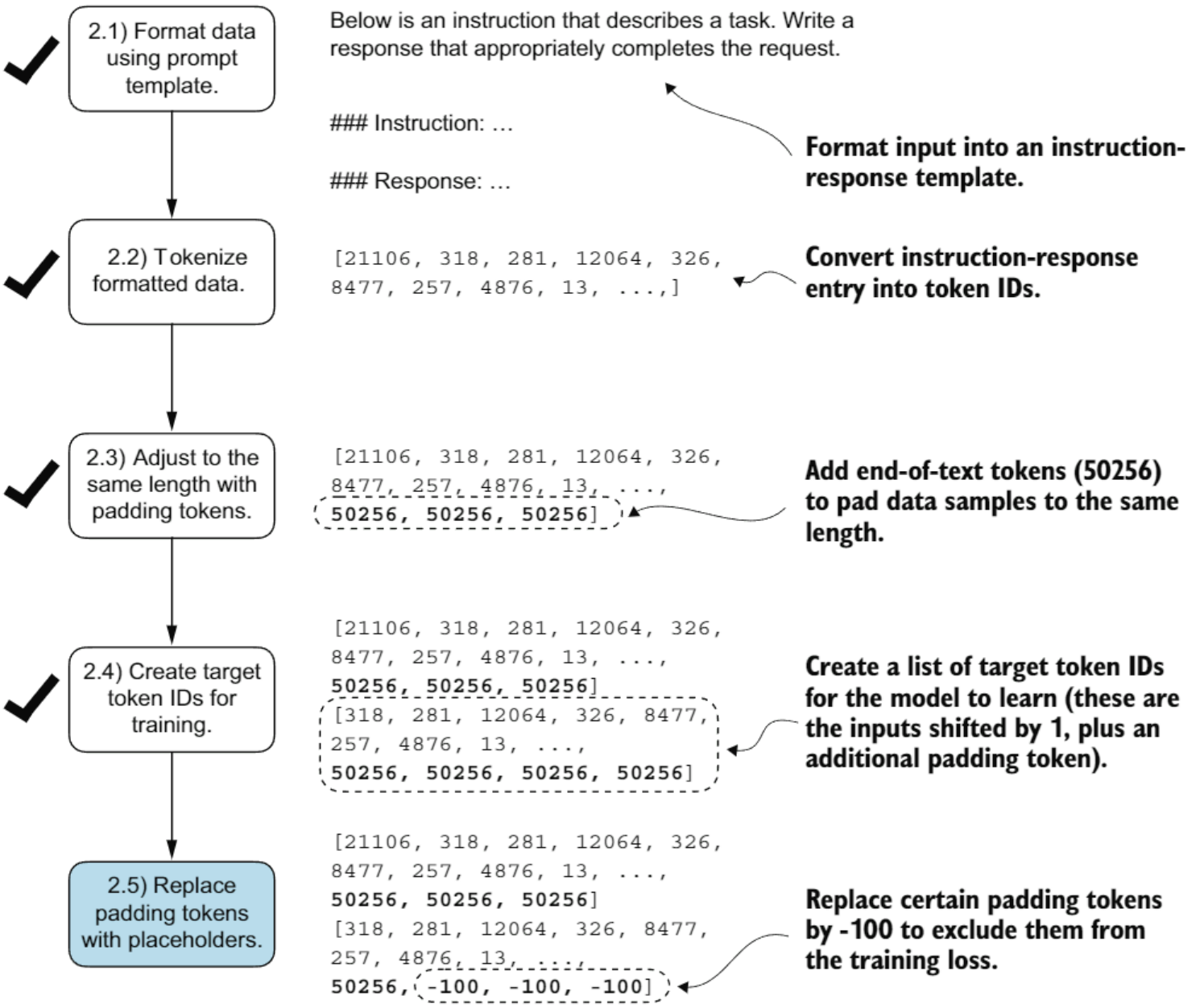

Figure 7.11 The five substeps involved in implementing the batching process. After creating the target sequence by shifting token IDs one position to the right and appending an end-of-text token, in step 2.5, we replace the end-of-text padding tokens with a placeholder value (`-100`).

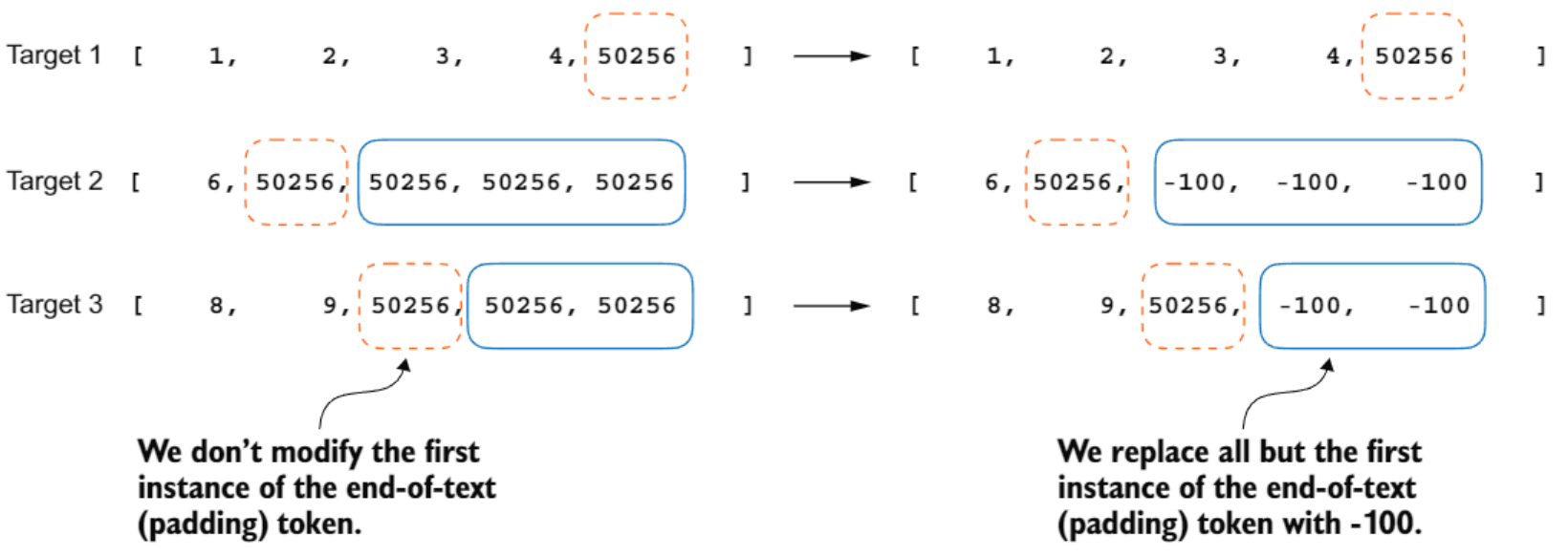

Figure 7.12 Step 2.4 in the token replacement process in the target batch for the training data preparation. We replace all but the first instance of the end-of-text token, which we use as padding, with the placeholder value `-100`, while keeping the initial end-of-text token in each target sequence.

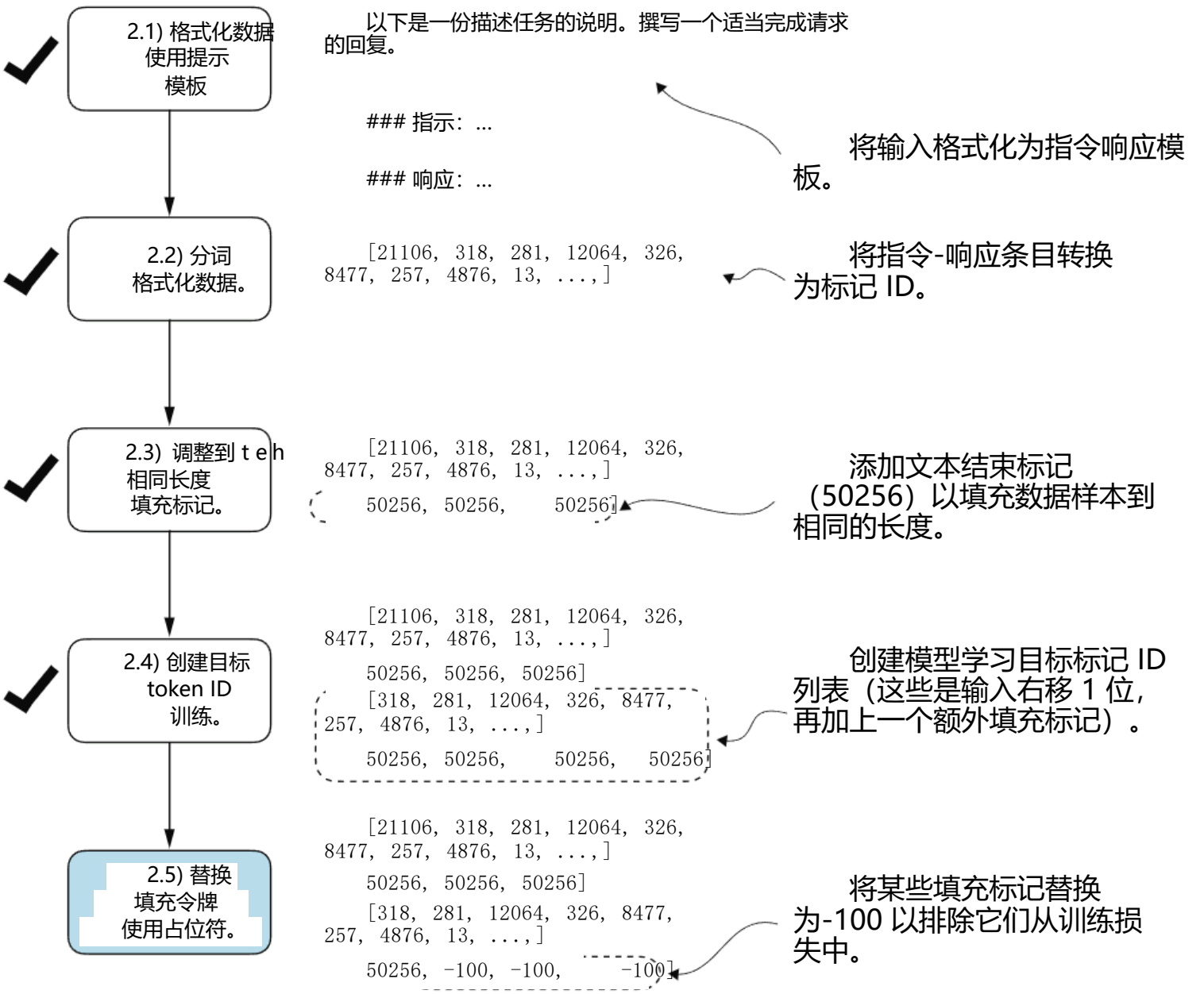

图 7.11 实施批处理过程涉及的五个子步骤。在通过将标记 ID 向右移动一个位置并附加一个文本结束标记创建目标序列后，在第 2.5 步中，我们将文本结束填充标记替换为占位符值（-100）。

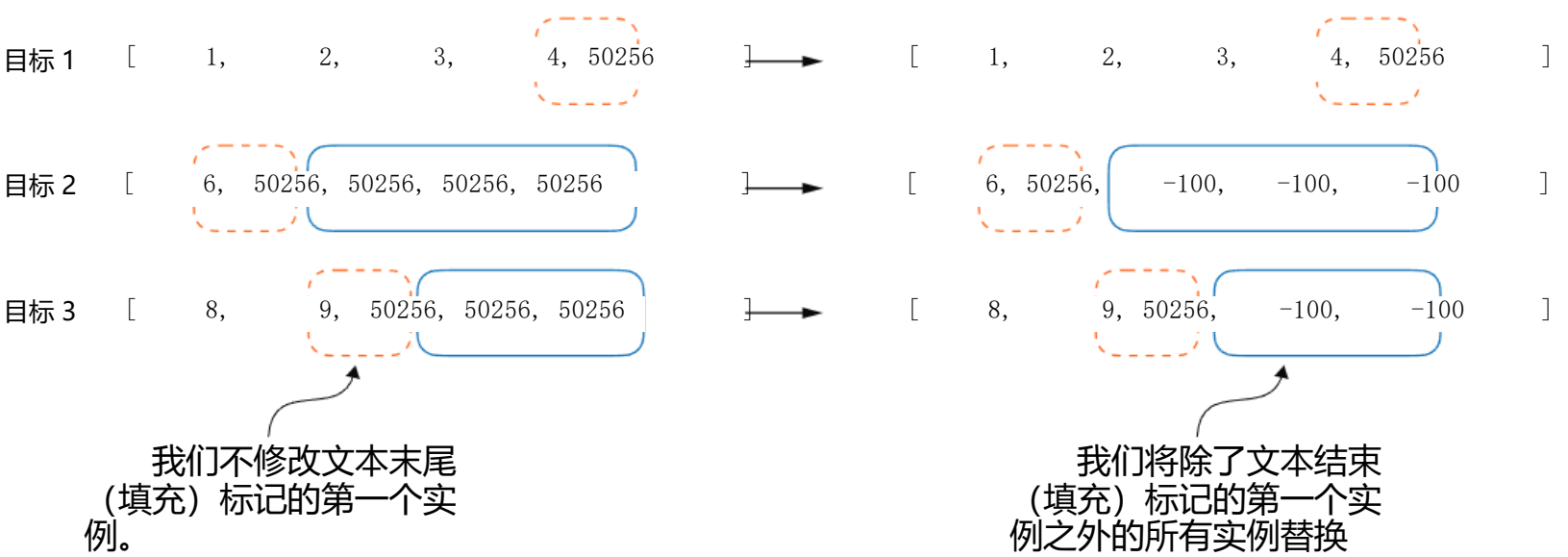

图 7.12 在训练数据准备的目标批次中，标记替换过程的第 2.4 步。我们将除了每个目标序列中第一个文本结束标记之外的所有文本结束标记替换为占位符值-100，同时保留每个目标序列中的初始文本结束标记。

Listing 7.5 Implementing a custom batch collate function

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]


        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor
```

Pads sequences to max_length

Truncates the last token for inputs

Shifts +1 to the right for targets

Replaces all but the first padding tokens in targets by ignore_index

Optionally truncates to the maximum sequence length

Again, let's try the collate function on the sample batch that we created earlier to check that it works as intended:

```
inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)
```

The results are as follows, where the first tensor represents the inputs and the second tensor represents the targets:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
```

列表 7.5 实现自定义批量合并函数

```
    def custom_collate_fn(
        批量
        pad_token_id=50256,
    ignore_index=-100,
allowed_max_length=None,
device="cpu"）：


        batch_max_length = max(len(item)+1 for item in batch)
     inputs_lst, targets_lst = [], []


        for item in 批次:
            new_item = item.copy()
         new_item += [填充令牌 ID]



            填充 = (
            new_item + [填充令牌 ID] *
        (batch_max_length - len(new_item)) ) inputs =
        torch.tensor(padded[:-1]) targets =
        torch.tensor(padded[1:])



            mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
                targets[indices[1:]] =      忽略索引
            如果 allowed_max_length 不是 None:
                inputs = 输入[:allowed_max_length] targets
             = 目标[:allowed_max_length]


            inputs_lst.append(inputs)
         targets_lst.append(targets)


        inputs_tensor = torch.stack(inputs_lst).to(设备)
     targets_tensor = torch.stack(targets_lst).to(设备) 返回
     inputs_tensor, targets_tensor
```

垫序列至最大长度

截断输入的最后标记

Shifts +1 向右移动以针对目标

替换目标中的除第一个填充标记之外的所有填充标记为

可选地截断到最大序列长度

再次，让我们尝试使用我们之前创建的样本批次来测试 collate 函数是否按预期工作：

```
   inputs, targets = custom_collate_fn(批次) 打印
(inputs) 打印(targets)
```

结果如下，其中第一个张量表示输入，第二个张量表示目标：

```
张量([[         0,       1,       2,       3,       4],
          [ 5, 6, ]          50256,   50256,   50256]
          [     7,       8,       9,   50256,   50256]])
```

```
tensor([[    1,     2,     3,     4, 50256],
        [    6, 50256,  -100,  -100,  -100],
        [    8,     9, 50256,  -100,  -100]])
```

The modified collate function works as expected, altering the target list by inserting the token ID `-100`. What is the logic behind this adjustment? Let's explore the underlying purpose of this modification.

For demonstration purposes, consider the following simple and self-contained example where each output logit corresponds to a potential token from the model's vocabulary. Here's how we might calculate the cross entropy loss (introduced in chapter 5) during training when the model predicts a sequence of tokens, which is similar to what we did when we pretrained the model and fine-tuned it for classification:

```
logits_1 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5]]
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

predictions for 1st token

predictions for 2nd token

The loss value calculated by the previous code is `1.1269`:

```
tensor(1.1269)
```

As we would expect, adding an additional token ID affects the loss calculation:

```
logits_2 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5],
     [-0.5, 1.5]]
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

New third token ID prediction

After adding the third token, the loss value is `0.7936`.

So far, we have carried out some more or less obvious example calculations using the cross entropy loss function in PyTorch, the same loss function we used in the training functions for pretraining and fine-tuning for classification. Now let's get to the interesting part and see what happens if we replace the third target token ID with `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

```
张量([[         1,         2,         3,         4, 50256]
          [      6, 50256,      -100,      -100,      -100]
          [      8,         9, 50256,      -100,      -100]])
```

修改后的 collate 函数按预期工作，通过插入令牌 ID -100 来更改目标列表。这种调整背后的逻辑是什么？让我们探索这次修改的潜在目的。

为了演示目的，考虑以下简单且自包含的示例，其中每个输出对数似然对应于模型词汇表中的一个潜在标记。以下是我们在训练期间计算交叉熵损失（在第 5 章中介绍）的方法，当模型预测一系列标记时，这与我们在预训练模型和微调其进行分类时所做的类似：

```
logits_1 = torch.tensor(
    [[-1.0, 1.0]]
     [-0.5,] 1.5]]
)
targets_1 = torch.tensor([0, 1]) # 正确的 token 索引以生成 loss_1 =
torch.nn.functional.cross_entropy(logits_1, targets_1) 打印(loss_1)
```

预测第 1 个标记

预测第 2 个标记

前一行代码计算出的损失值为 1.1269：

```
张量(1.1269)
```

正如我们所预期的那样，添加一个额外的令牌 ID 会影响损失计算：

```
logits_2 = torch.tensor(
    [[-1.0, 1.0]]
     [-0.5, 1.5]
     [-0.5,] 1.5]]
)
targets_2 = torch.tensor([0, 1, 1]) loss_2 =
torch.nn.functional.cross_entropy(logits_2, targets_2) 打印(loss_2)
```

新第三代令牌 ID 预测

在添加第三个标记后，损失值为 0.7936。

到目前为止，我们已经使用 PyTorch 中的交叉熵损失函数进行了一些或多或少明显的示例计算，这个损失函数是我们用于预训练和微调分类训练函数的相同损失函数。现在让我们进入有趣的部分，看看如果我们用-100 替换第三个目标标记 ID 会发生什么：

```
targets_3 = torch.tensor([0, 1, -100]) loss_3 =
torch.nn.functional.cross_entropy(logits_2, targets_3) 打印(loss_3) 打印
("loss_1 == loss_3:", loss_1 == loss_3)
```

The resulting output is

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

The resulting loss on these three training examples is identical to the loss we calculated from the two training examples earlier. In other words, the cross entropy loss function ignored the third entry in the `targets_3` vector, the token ID corresponding to `-100`. (Interested readers can try to replace the `-100` value with another token ID that is not `0` or `1`; it will result in an error.)

So what's so special about `-100` that it's ignored by the cross entropy loss? The default setting of the cross entropy function in PyTorch is `cross_entropy(..., ignore_index=-100)`. This means that it ignores targets labeled with `-100`. We take advantage of this `ignore_index` to ignore the additional end-of-text (padding) tokens that we used to pad the training examples to have the same length in each batch. However, we want to keep one `50256` (end-of-text) token ID in the targets because it helps the LLM to learn to generate end-of-text tokens, which we can use as an indicator that a response is complete.

In addition to masking out padding tokens, it is also common to mask out the target token IDs that correspond to the instruction, as illustrated in figure 7.13. By masking out the LLM's target token IDs corresponding to the instruction, the cross entropy loss is only computed for the generated response target IDs. Thus, the model is trained to focus on generating accurate responses rather than memorizing instructions, which can help reduce overfitting.

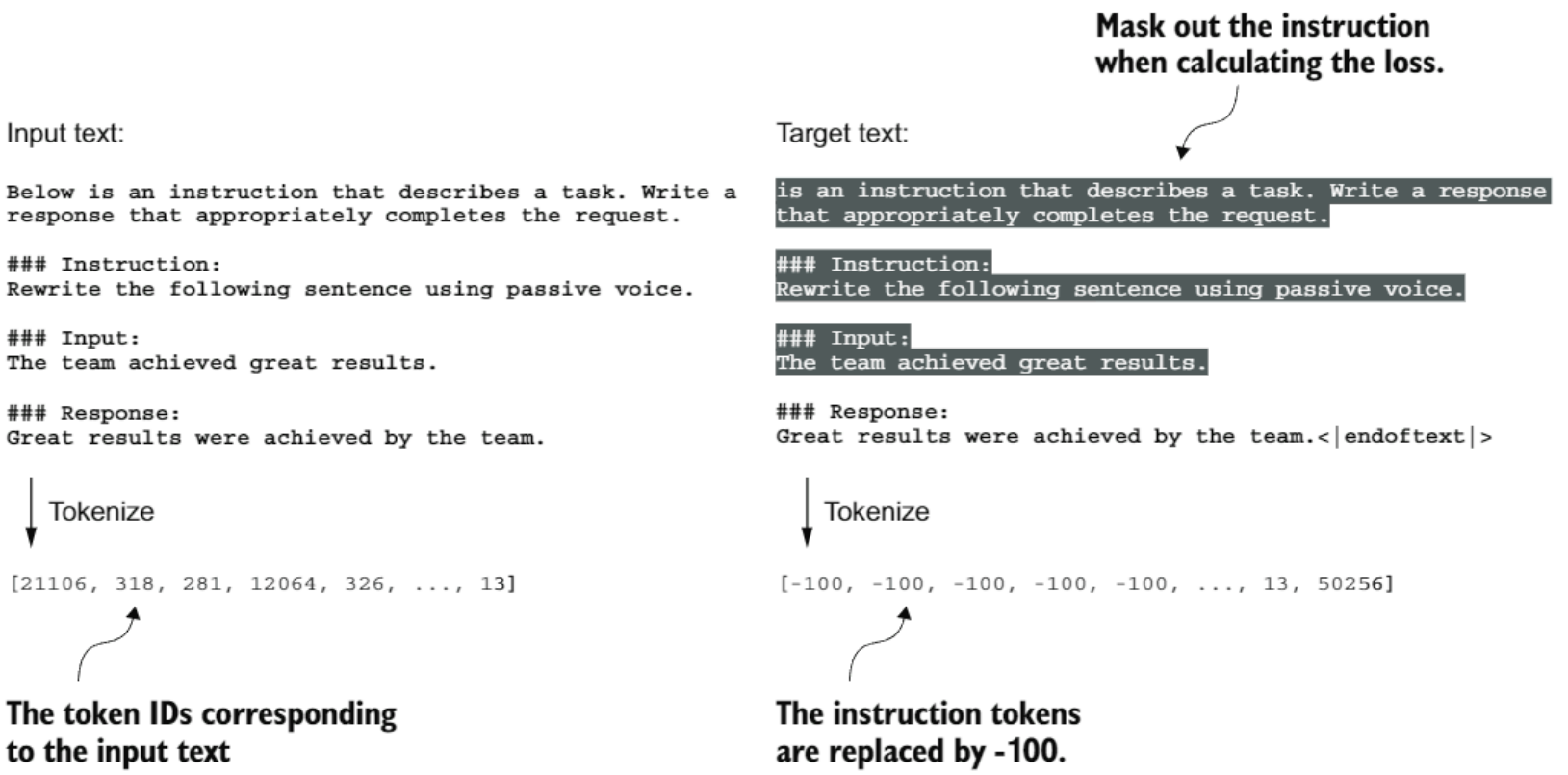

Figure 7.13 Left: The formatted input text we tokenize and then feed to the LLM during training. Right: The target text we prepare for the LLM where we can optionally mask out the instruction section, which means replacing the corresponding token IDs with the `-100` `ignore_index` value.

结果输出

```
张量(1.1269)
损失_1    == loss_3:    张量(真)
```

这三个训练示例的结果损失与之前从两个训练示例中计算出的损失相同。换句话说，交叉熵损失函数忽略了 targets_3 向量中的第三个条目，即对应于-100 的标记 ID。（感兴趣的读者可以尝试用另一个非 0 或 1 的标记 ID 替换-100 值；这将导致错误。）那么-100 有什么特别之处，以至于交叉熵损失会忽略它？PyTorch 中交叉熵函数的默认设置是 cross_entropy(...,
忽略索引=-100)。这意味着它忽略了标记为-100 的目标。我们利用这个 ignore_index 来忽略我们用来填充训练示例以使每个批次长度相同的额外文本结束（填充）标记。然而，我们想在目标中保留一个 50256（文本结束）标记 ID，因为这有助于LLM学习生成文本结束标记，我们可以将其用作指示响应完整的标志。

除了屏蔽填充标记之外，通常还会屏蔽与指令对应的目标标记 ID，如图 7.13 所示。通过屏蔽与指令对应的LLM的目标标记 ID，仅计算生成的响应目标 ID 的交叉熵损失。因此，模型被训练以专注于生成准确的响应，而不是记忆指令，这有助于减少过拟合。

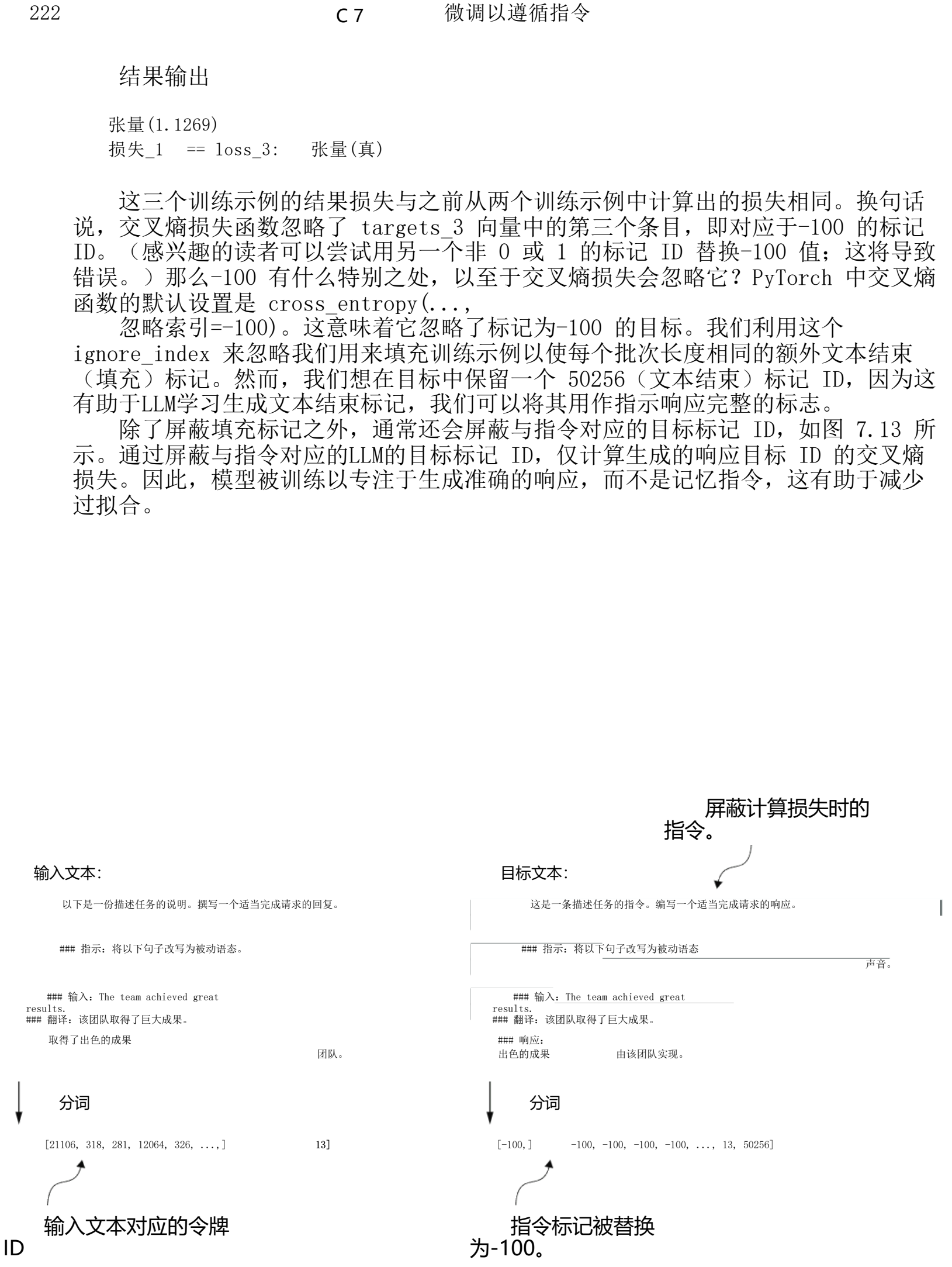

图 7.13 左：我们在训练过程中分词并输入到LLM的格式化输入文本。右：我们为LLM准备的目标文本，我们可以选择性地屏蔽指令部分，即用-100 ignore_index 值替换相应的标记 ID。

As of this writing, researchers are divided on whether masking the instructions is universally beneficial during instruction fine-tuning. For instance, the 2024 paper by Shi et al., "Instruction Tuning With Loss Over Instructions" (https://arxiv.org/abs/2405.14394), demonstrated that not masking the instructions benefits the LLM performance (see appendix B for more details). Here, we will not apply masking and leave it as an optional exercise for interested readers.

Exercise 7.2 Instruction and input masking

After completing the chapter and fine-tuning the model with `InstructionDataset`, replace the instruction and input tokens with the `-100` mask to use the instruction masking method illustrated in figure 7.13. Then evaluate whether this has a positive effect on model performance.

7.4 Creating data loaders for an instruction dataset

We have completed several stages to implement an `InstructionDataset` class and a `custom_collate_fn` function for the instruction dataset. As shown in figure 7.14, we are ready to reap the fruits of our labor by simply plugging both `InstructionDataset` objects and the `custom_collate_fn` function into PyTorch data loaders. These loaders

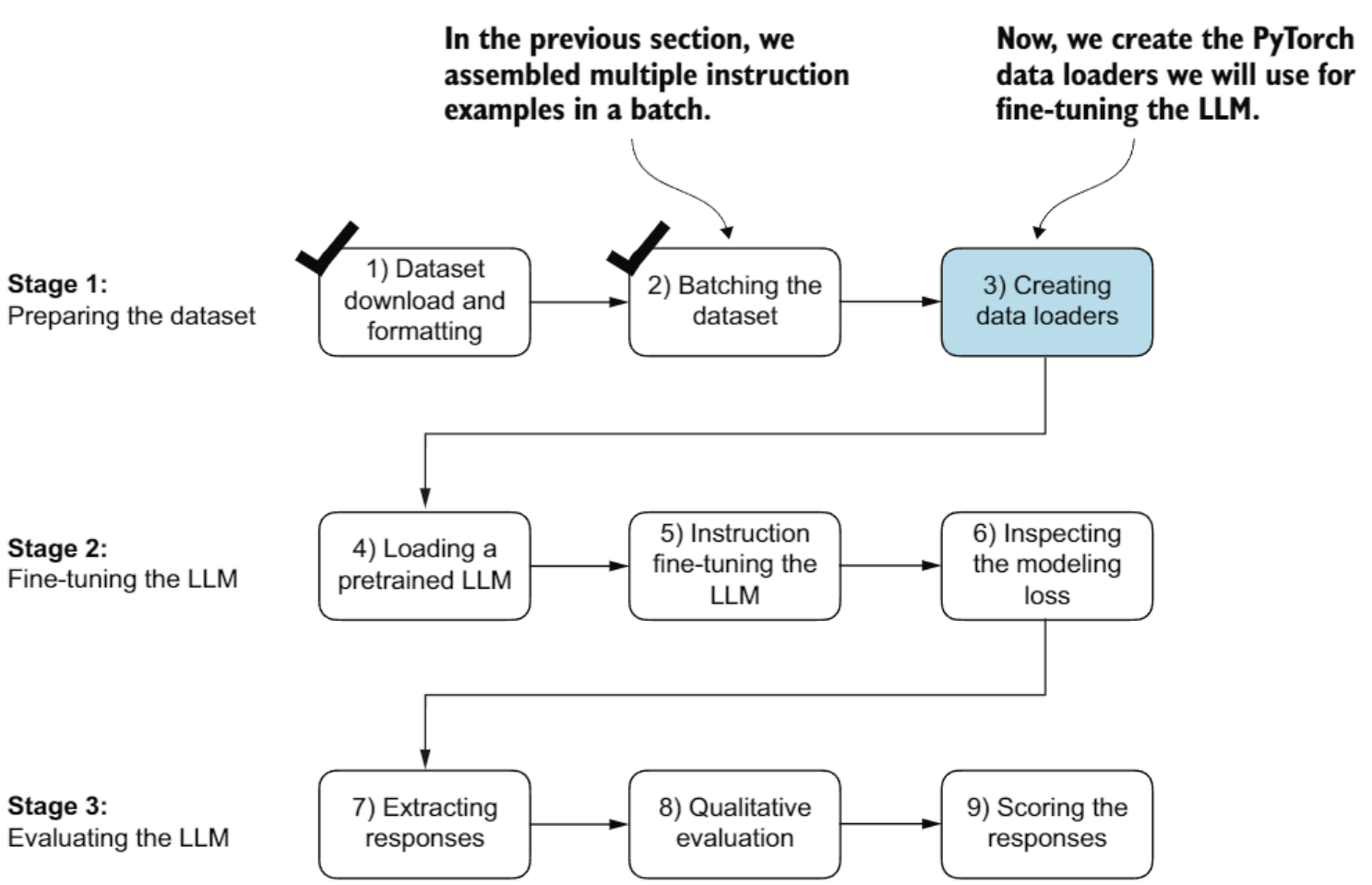

Figure 7.14 The three-stage process for instruction fine-tuning an LLM. Thus far, we have prepared the dataset and implemented a custom collate function to batch the instruction dataset. Now, we can create and apply the data loaders to the training, validation, and test sets needed for the LLM instruction fine-tuning and evaluation.

关于本写作，研究人员在是否在指令微调期间遮蔽指令普遍有益的问题上存在分歧。例如，Shi 等人 2024 年的论文《带有指令损失的指令微调》（https://arxiv.org/abs/2405.14394）表明，不遮蔽指令有利于LLM性能（更多细节请见附录 B）。在此，我们将不应用遮蔽，将其作为对感兴趣的读者的一项可选练习。

练习 7.2 指令和输入掩码

完成章节并使用 InstructionDataset 微调模型后，将指令和输入标记替换为-100 掩码，以使用图 7.13 中所示的指令掩码方法。然后评估这是否对模型性能有积极影响。

7.4 创建指令数据集的数据加载器

我们已经完成了几个阶段，以实现一个 InstructionDataset 类和一个 custom_collate_fn 函数，用于指令数据集。如图 7.14 所示，我们只需将这两个 InstructionDataset 对象和 custom_collate_fn 函数插入 PyTorch 数据加载器中，就可以收获我们劳动的果实。这些加载器

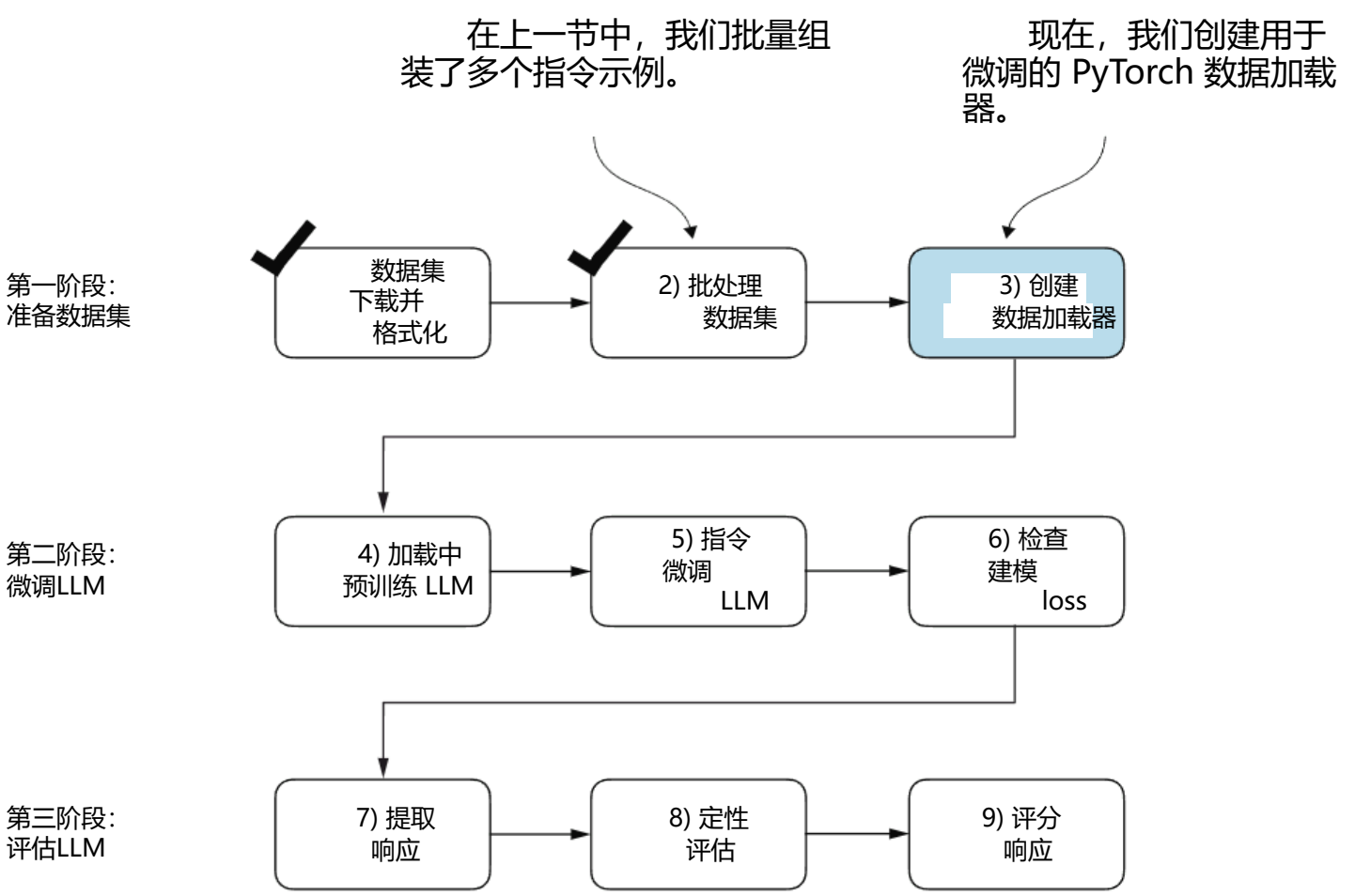

图 7.14 指令微调的三个阶段过程 LLM。迄今为止，我们已经准备好了数据集并实现了一个自定义的 collate 函数来批量处理指令数据集。现在，我们可以创建并应用数据加载器到所需的训练、验证和测试集，用于LLM指令微调和评估。

will automatically shuffle and organize the batches for the LLM instruction fine-tuning process.

Before we implement the data loader creation step, we have to briefly talk about the `device` setting of the `custom_collate_fn`. The `custom_collate_fn` includes code to move the input and target tensors (for example, `torch.stack(inputs_lst).to(device)`) to a specified device, which can be either `"cpu"` or `"cuda"` (for NVIDIA GPUs) or, optionally, `"mps"` for Macs with Apple Silicon chips.

> **NOTE** Using an `"mps"` device may result in numerical differences compared to the contents of this chapter, as Apple Silicon support in PyTorch is still experimental.

Previously, we moved the data onto the target device (for example, the GPU memory when `device="cuda"`) in the main training loop. Having this as part of the collate function offers the advantage of performing this device transfer process as a background process outside the training loop, preventing it from blocking the GPU during model training.

The following code initializes the `device` variable:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available():
#     device = torch.device("mps")"
print("Device:", device)
```

Uncomments these two lines to use the GPU on an Apple Silicon chip

This will either print `"Device: cpu"` or `"Device: cuda"`, depending on your machine.

Next, to reuse the chosen device setting in `custom_collate_fn` when we plug it into the PyTorch `DataLoader` class, we use the `partial` function from Python's `functools` standard library to create a new version of the function with the device argument prefilled. Additionally, we set the `allowed_max_length` to `1024`, which truncates the data to the maximum context length supported by the GPT-2 model, which we will fine-tune later:

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
    allowed_max_length=1024
)
```

Next, we can set up the data loaders as we did previously, but this time, we will use our custom collate function for the batching process.

将自动打乱并组织批次以供LLM指令微调过程。

在我们实现数据加载器创建步骤之前，我们不得不简要谈谈
设备设置中的 custom_collate_fn。custom_collate_fn 包含代码
将输入和目标张量（例如，torch.stack(inputs_lst).to(device)）移动到指定的设备，可以是“cpu”或“cuda”（用于 NVIDIA GPU）或可选的“mps”（用于搭载 Apple Silicon 芯片的 Mac）。

注意：使用“mps”设备可能会导致与本章内容存在数值差异，因为 PyTorch 对 Apple Silicon 的支持仍然是实验性的。

之前，我们在主训练循环中将数据移动到目标设备上（例如，当 device="cuda"时为 GPU 内存）。将此作为 collate 函数的一部分，可以提供将设备传输过程作为后台进程执行的优势，从而防止它在模型训练期间阻塞 GPU。

以下代码初始化设备变量：

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # if
torch.backends.mps.is_available(): # device = torch.device("mps")" 打印("设备：",
device)
```

取消注释这两行以在苹果硅芯片上使用 GPU

这将根据您的机器打印 "设备：cpu" 或 "设备：cuda"。

接下来，当我们将选定的设备设置在 custom_collate_fn 中重新使用时，将其插入使用 Python 的 functools 标准库中的 partial 函数创建一个新的版本，该版本预先填充了设备参数。此外，我们将允许的最大长度设置为 1024，这将截断数据到 GPU-2 模型支持的最大上下文长度，我们将在稍后进行微调：

PyTorch 数据加载器，在其中我们创建一个新的

```
    from functools 模块导入 partial

    自定义合并函数 = 部分函数
        自定义合并函数
    设备=设备，允许最大长度
=1024）
```

接下来，我们可以像之前一样设置数据加载器，但这次我们将使用我们自定义的 collate 函数进行批处理过程。

Listing 7.6 Initializing the data loaders

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers
)

val_dataset = InstructionDataset(val_data, tokenizer)
val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers
)

test_dataset = InstructionDataset(test_data, tokenizer)
test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers
)
```

You can try to increase this number if parallel Python processes are supported by your operating system.

Let's examine the dimensions of the input and target batches generated by the training loader:

```
print("Train loader:")
for inputs, targets in train_loader:
    print(inputs.shape, targets.shape)
```

The output is as follows (truncated to conserve space):

```
Train loader:
torch.Size([8, 61]) torch.Size([8, 61])
torch.Size([8, 76]) torch.Size([8, 76])
torch.Size([8, 73]) torch.Size([8, 73])
...
```

列表 7.6 初始化数据加载器

```
from  torch.utils.data 导入 DataLoader

num_workers = 0
batch_size = 8

torch 手动设置随机种子(123)

训练数据集 = InstructionDataset(train_data, tokenizer) 训练加载器
= DataLoader()
    train_dataset, 批大小=batch_size, 合
并函数=customized_collate_fn, 打乱=True,
丢弃最后=True, 工作进程数=num_workers )



    val_dataset = 指令数据集(val_data, tokenizer) val_loader = 数
据加载器(
    val_dataset, batch_size=batch_size,
collate_fn=customized_collate_fn,
shuffle=False, drop_last=False,
num_workers=num_workers)
```

您可以尝试增加这个数字，如果您的操作系统支持并行 Python 进程。

```
    test_dataset = 指令数据集(test_data, tokenizer) test_loader =
数据加载器(
    测试数据集，batch_size=批量大小，
collate_fn=自定义合并函数，
shuffle=False, drop_last=False,
num_workers=工作进程数）
```

让我们检查由训练加载器生成的输入和目标批次的维度：

```
    打印("训练加载器：") for 输入, 目标 in
训练加载器：
        打印(inputs 的形状)     targets.shape) # 目标形状
```

以下为输出结果（已截断以节省空间）：

```
    训练加载器：torch.Size([8, 61])
torch.Size([8, 61]) torch.Size([8, 76])
torch.Size([8, 76]) torch.Size([8, 73])
torch.Size([8, 73]) ...
```

```
torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])
```

This output shows that the first input and target batch have dimensions 8×61, where 8 represents the batch size and 61 is the number of tokens in each training example in this batch. The second input and target batch have a different number of tokens—for instance, 76. Thanks to our custom collate function, the data loader is able to create batches of different lengths. In the next section, we load a pretrained LLM that we can then fine-tune with this data loader.

7.5 Loading a pretrained LLM

We have spent a lot of time preparing the dataset for instruction fine-tuning, which is a key aspect of the supervised fine-tuning process. Many other aspects are the same as in pretraining, allowing us to reuse much of the code from earlier chapters.

Before beginning instruction fine-tuning, we must first load a pretrained GPT model that we want to fine-tune (see figure 7.15), a process we have undertaken previously. However, instead of using the smallest 124-million-parameter model as before, we load the medium-sized model with 355 million parameters. The reason for this choice is that the 124-million-parameter model is too limited in capacity to achieve

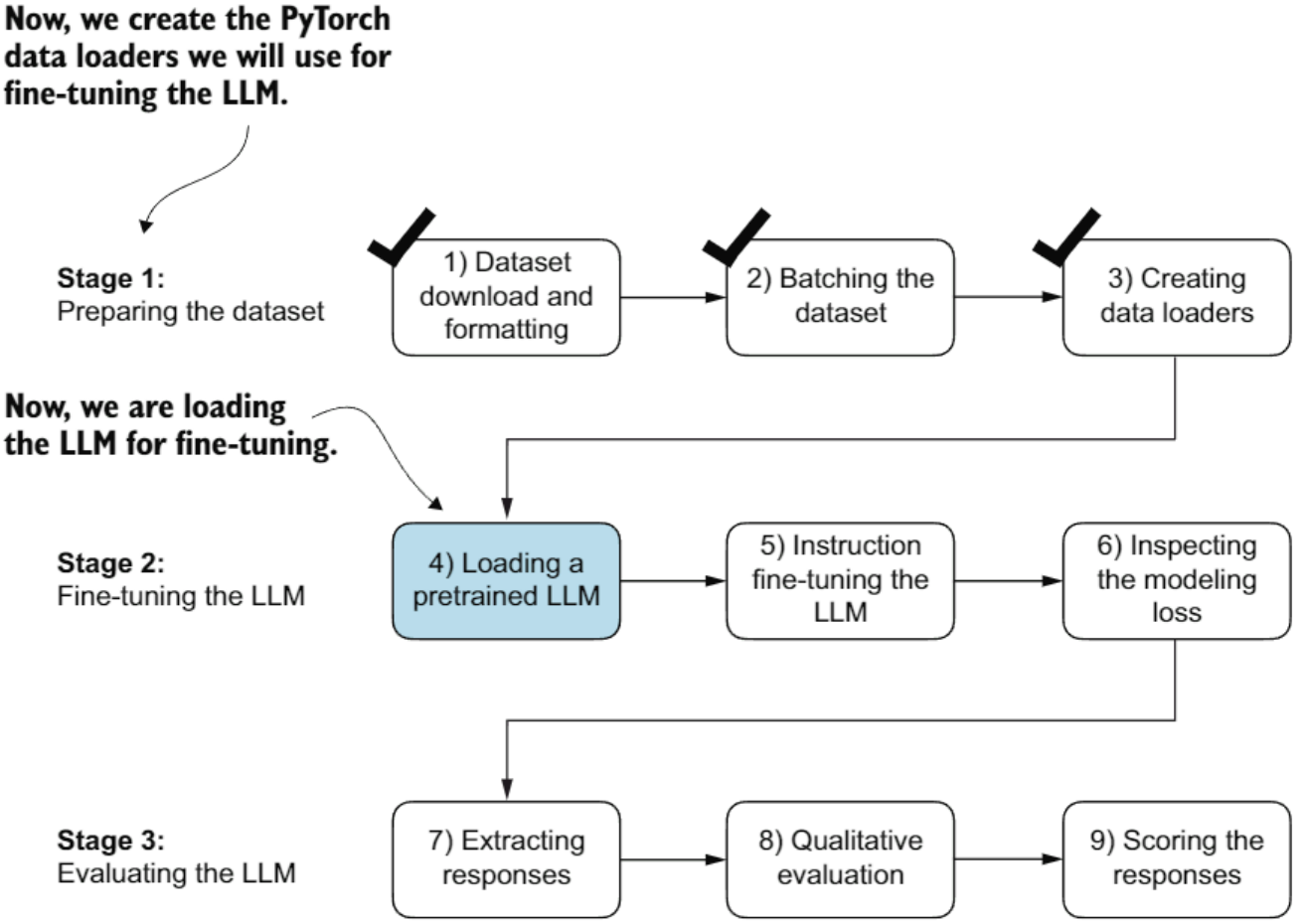

Figure 7.15 The three-stage process for instruction fine-tuning an LLM. After the dataset preparation, the process of fine-tuning an LLM for instruction-following begins with loading a pretrained LLM, which serves as the foundation for subsequent training.

```
    torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])
```

此输出显示，第一个输入和目标批次维度为 8 × 61，其中 8 表示批次大小，61 是每个训练示例中此批次的标记数。第二个输入和目标批次标记数不同——例如，76。多亏了我们的自定义合并函数，数据加载器能够创建不同长度的批次。在下一节中，我们加载一个预训练的LLM，然后我们可以使用此数据加载器对其进行微调。

7.5 加载预训练的 LLM

我们花费了大量时间准备用于指令微调的数据集，这是监督微调过程中的关键方面。许多其他方面与预训练相同，使我们能够重用早期章节中的大量代码。

在开始指令微调之前，我们首先需要加载一个预训练的 GPT 模型，这是我们想要微调的模型（见图 7.15），这是我们之前已经进行过的过程。然而，与之前使用最小的 124 百万参数模型不同，我们加载了一个中等大小的 355 百万参数模型。选择这个模型的原因是 124 百万参数模型在容量上过于有限，无法实现

现在，我们创建用于微调的 PyTorch 数据加载器。

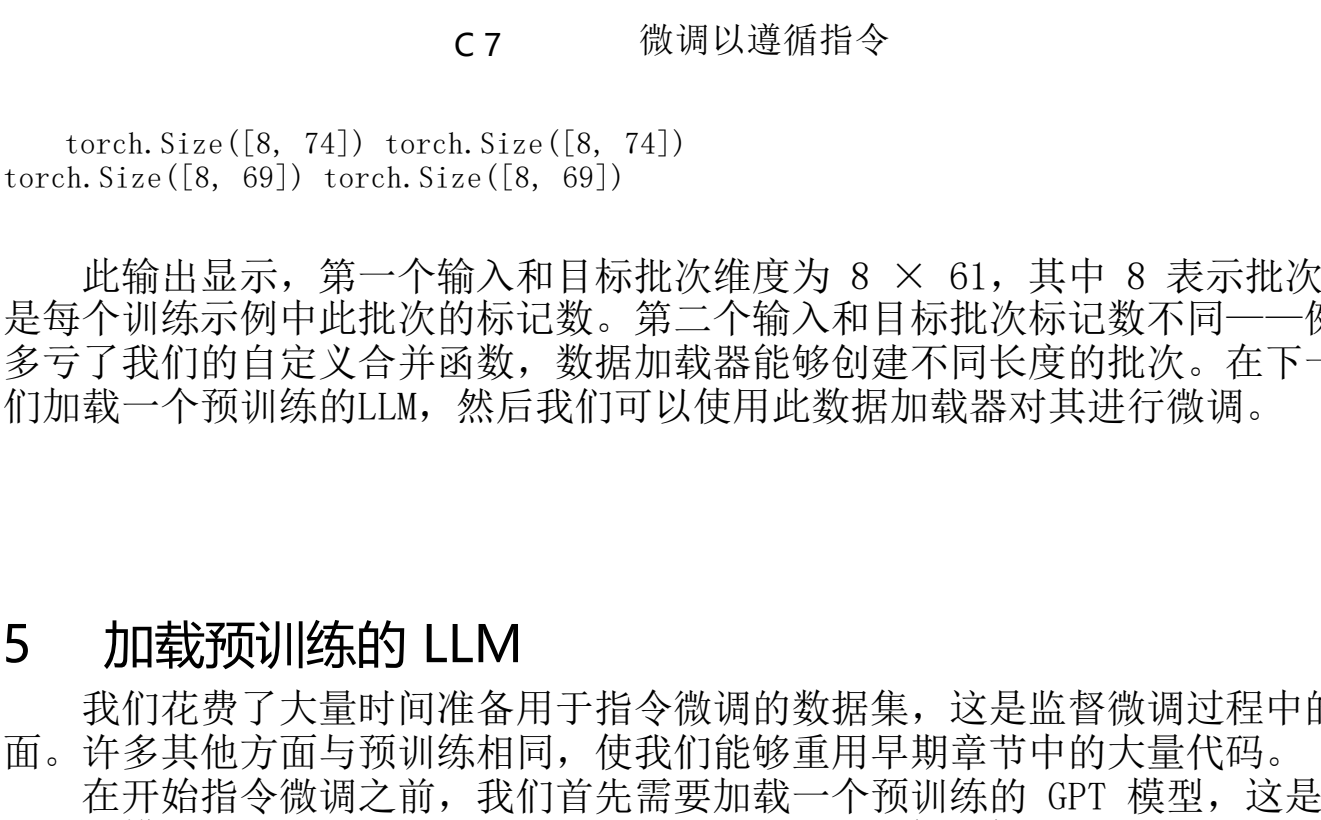

图 7.15 指令微调的三个阶段过程 LLM。在数据集准备之后，指令遵循的微调过程 LLM 以加载预训练的 LLM 为基础，作为后续训练的基础。

satisfactory results via instruction fine-tuning. Specifically, smaller models lack the necessary capacity to learn and retain the intricate patterns and nuanced behaviors required for high-quality instruction-following tasks.

Loading our pretrained models requires the same code as when we pretrained the data (section 5.5) and fine-tuned it for classification (section 6.4), except that we now specify `"gpt2-medium (355M)"` instead of `"gpt2-small (124M)"`.

> **NOTE** Executing this code will initiate the download of the medium-sized GPT model, which has a storage requirement of approximately 1.42 gigabytes. This is roughly three times larger than the storage space needed for the small model.

Listing 7.7 Loading the pretrained model

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

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},
}

CHOOSE_MODEL = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

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 executing the code, several files will be downloaded:

```
checkpoint: 100%|███████████| 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json: 100%|███████████| 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json: 100%|███████████| 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001: 100%|███████████| 1.42G/1.42G
```

通过指令微调获得满意的结果。特别是，较小的模型缺乏学习和保留高质量指令遵循任务所需的复杂模式和细微行为所必需的容量。

加载我们的预训练模型需要与我们在（第 5.5 节）预训练数据时相同的代码，并且为分类进行了微调（第 6.4 节），除了我们现在

指定 "gpt2-medium (355M)" 而不是 "gpt2-small (124M)"。

注意：执行此代码将启动中等规模 GPT 模型的下载，该模型存储需求约为 1.42GB。这大约是小模型所需存储空间的 3 倍。

列表 7.7 加载预训练模型

```
从 gpt_download 模块导入 download_and_load_gpt2 函
数，从 chapter04 模块导入 GPTModel 类，从 chapter05 模块
导入 load_weights_into_gpt 函数


BASE_CONFIG = {
"vocab_size": 50257, # 词汇量大小 "context_length":
1024, # 上下文长度 "drop_rate": 0.0, # 抖动率 "qkv_bias":
True # Query-key-value 偏置 }



模型配置 = {
"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}, }



选择模型 = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])


model_size = 选择模型.split(" ")[-1].lstrip("(").rstrip(")")

设置，参数 = 下载并加载 gpt2()
model_size=model_size,
模型目录="gpt2" )



model = GPT 模型(BASE_CONFIG) 将权重加载
到 gpt(model, params) model.eval();
```

执行代码后，将下载几个文件：

```
检查点：100%|██████████| 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json：100%|██████████| 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json：100%|██████████| 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001：100%|██████████| 1.42G/1.42G
```

```
[05:50<00:00, 4.05MiB/s]
model.ckpt.index: 100%|██████████| 10.4k/10.4k [00:00<00:00, 18.1MiB/s]
model.ckpt.meta: 100%|██████████| 927k/927k [00:02<00:00, 454kiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:01<00:00, 283kiB/s]
```

Now, let's take a moment to assess the pretrained LLM's performance on one of the validation tasks by comparing its output to the expected response. This will give us a baseline understanding of how well the model performs on an instruction-following task right out of the box, prior to fine-tuning, and will help us appreciate the effect of fine-tuning later on. We will use the first example from the validation set for this assessment:

```
torch.manual_seed(123)
input_text = format_input(val_data[0])
print(input_text)
```

The content of the instruction is as follows:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal every day.'
```

Next we generate the model's response using the same `generate` function we used to pretrain the model in chapter 5:

```
from chapter05 import generate, text_to_token_ids, token_ids_to_text

token_ids = generate(
    model=model,
    idx=text_to_token_ids(input_text, tokenizer),
    max_new_tokens=35,
    context_size=BASE_CONFIG["context_length"],
    eos_id=50256,
)
generated_text = token_ids_to_text(token_ids, tokenizer)
```

The `generate` function returns the combined input and output text. This behavior was previously convenient since pretrained LLMs are primarily designed as text-completion models, where the input and output are concatenated to create coherent and legible text. However, when evaluating the model's performance on a specific task, we often want to focus solely on the model's generated response.

To isolate the model's response text, we need to subtract the length of the input instruction from the start of the `generated_text`:

```
response_text = generated_text[len(input_text):].strip()
print(response_text)
```

```
    42G [05:50<00:00, 4.05MiB/s] model.ckpt.index: 100%|████████████| 
10.4k/10.4k [00:00<00:00, 18.1MiB/s] model.ckpt.meta: 100%|████████████| 
927k/927k [00:02<00:00, 454kiB/s] vocab.bpe: 100%|████████████| 456k/456k 
[00:01<00:00, 283kiB/s]
```

现在，让我们花一点时间评估预训练的LLM在验证任务上的性能，通过将其输出与预期响应进行比较。这将为我们提供一个基准，了解模型在未经微调的情况下直接执行指令任务的表现如何，并有助于我们后来欣赏微调的效果。我们将使用验证集的第一个示例进行此评估：

```
    torch 手动设置随机种子(123) input_text = 格
式化输入(val_data[0]) 打印(input_text)
```

指令内容如下：

```
以下是一个描述任务的说明。编写一个适当完成请求的回复。


### 指令：转换活动句子
                                        转换为被动语态:厨师烹饪                每天的一餐。
```

接下来，我们使用与第 5 章中预训练模型相同的生成函数来生成模型的响应：

```
    从第五章导入 generate,                text_to_token_ids,     token_ids 转换为文本

    token_ids = generate()
    model=model, idx=将输入文本转换为 token_id(input_text, 
tokenizer), 最大新 token 数=35, 上下文大小
=BASE_CONFIG["context_length"], eos_id=50256, ) 生成的文本=将
token_id 转换为文本(token_ids, tokenizer)
```

生成函数返回输入和输出的组合文本。这种行为之前很方便，因为预训练的LLMs主要设计为文本补全模型，输入和输出连接起来可以创建连贯易读的文本。然而，当评估模型在特定任务上的性能时，我们通常只想关注模型的生成响应。

将模型响应文本隔离，我们需要从生成的文本起始处减去输入指令的长度：

```
    response_text = 生成文本[len(input_text):].strip() 打印
(response_text)
```

This code removes the input text from the beginning of the `generated_text`, leaving us with only the model's generated response. The `strip()` function is then applied to remove any leading or trailing whitespace characters. The output is

```
### Response:

The chef cooks the meal every day.

### Instruction:

Convert the active sentence to passive: 'The chef cooks the
```

This output shows that the pretrained model is not yet capable of correctly following the given instruction. While it does create a Response section, it simply repeats the original input sentence and part of the instruction, failing to convert the active sentence to passive voice as requested. So, let's now implement the fine-tuning process to improve the model's ability to comprehend and appropriately respond to such requests.

7.6 Fine-tuning the LLM on instruction data

It's time to fine-tune the LLM for instructions (figure 7.16). We will take the loaded pretrained model in the previous section and further train it using the previously prepared instruction dataset prepared earlier in this chapter. We already did all the hard work when we implemented the instruction dataset processing at the beginning of

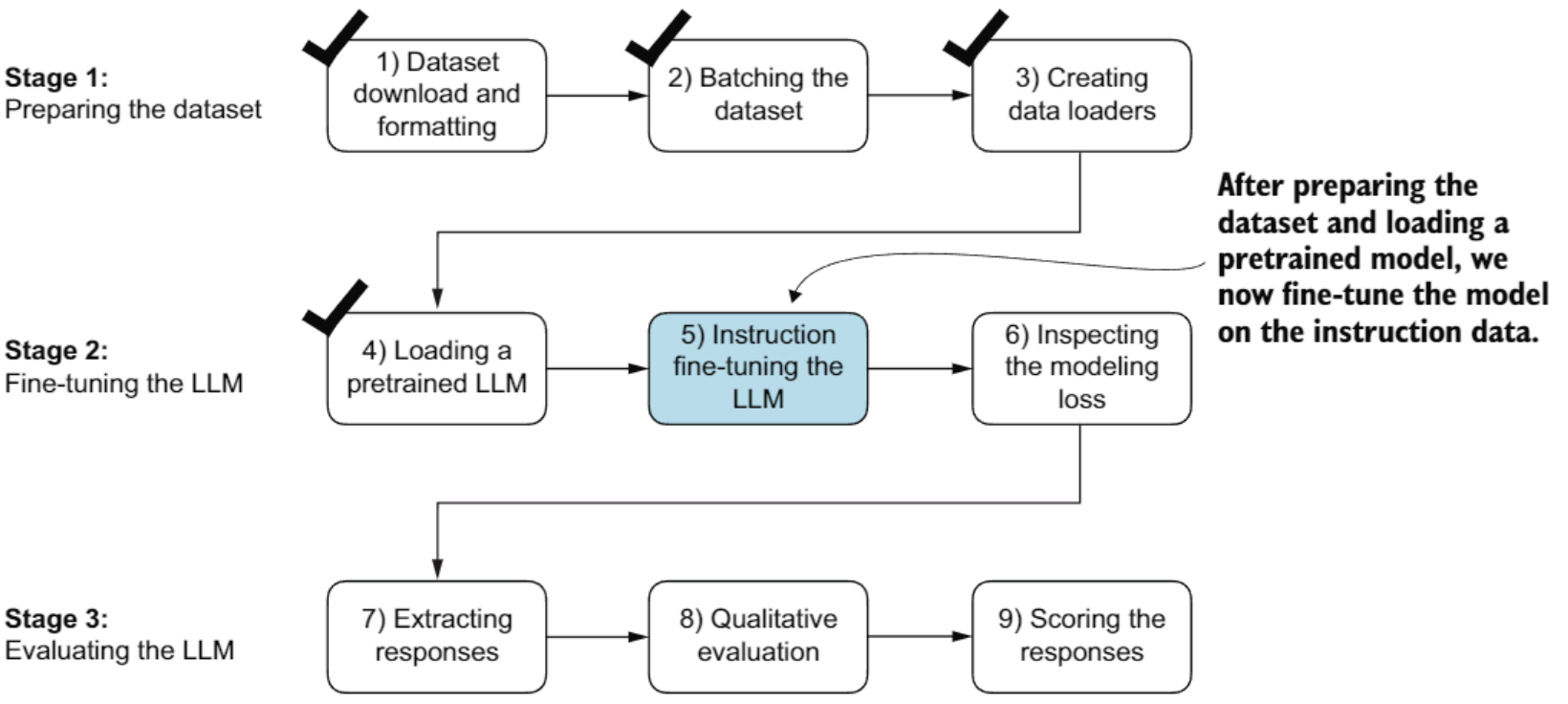

Figure 7.16 The three-stage process for instruction fine-tuning an LLM. In step 5, we train the pretrained model we previously loaded on the instruction dataset we prepared earlier.

此代码从生成的_text 开头删除输入文本，只留下模型的生成响应。然后应用 strip()函数以删除任何前导或尾随空白字符。输出为

```
### 响应：

厨师烹饪                    meal  每一天。

### 指令：

转换                活跃      输入文本的简体中被动翻译如下：The sentence 烹饪
```

该输出表明预训练模型尚不能正确遵循给定指令。虽然它确实创建了一个响应部分，但它只是重复了原始输入句子和部分指令，未能将主动句转换为请求的被动语态。因此，现在让我们实施微调过程，以提高模型理解和适当响应此类请求的能力。

7.6 微调指令数据中的LLM

现在是时候微调LLM以用于指令（图 7.16）。我们将使用上一节中加载的预训练模型，并使用本章之前准备好的指令数据集进一步训练它。当我们一开始实现指令数据集处理时，我们已经完成了所有艰苦的工作。

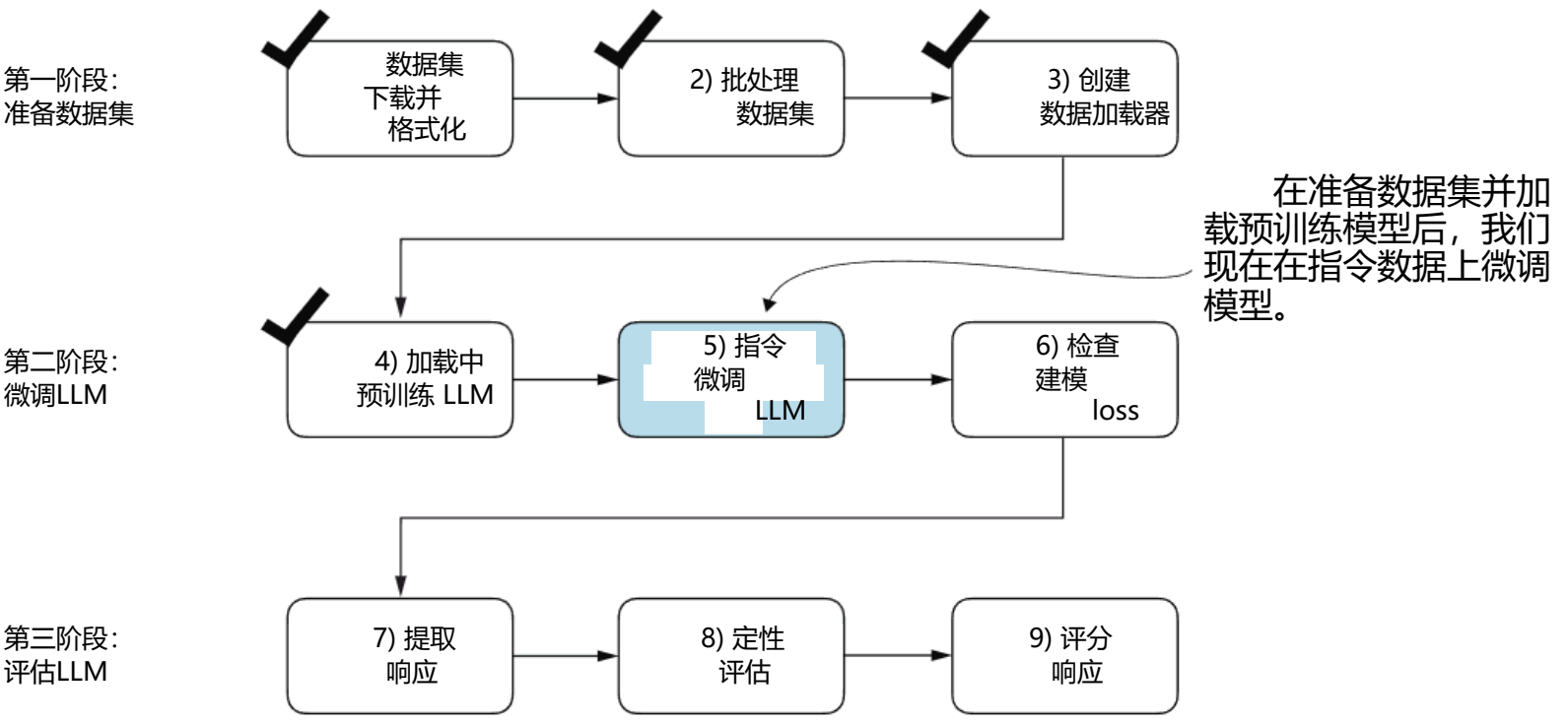

图 7.16 指令微调的三阶段过程 LLM。在第 5 步，我们在之前准备好的指令数据集上训练我们之前加载的预训练模型。

this chapter. For the fine-tuning process itself, we can reuse the loss calculation and training functions implemented in chapter 5:

```
from chapter05 import (
    calc_loss_loader,
    train_model_simple
)
```

Before we begin training, let's calculate the initial loss for the training and validation sets:

```
model.to(device)
torch.manual_seed(123)

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
)

print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

The initial loss values are as follows; as previously, our goal is to minimize the loss:

```
Training loss: 3.825908660888672
Validation loss: 3.7619335651397705
```

Dealing with hardware limitations

Using and training a larger model like GPT-2 medium (355 million parameters) is more computationally intensive than the smaller GPT-2 model (124 million parameters). If you encounter problems due to hardware limitations, you can switch to the smaller model by changing `CHOOSE_MODEL = "gpt2-medium (355M)"` to `CHOOSE_MODEL = "gpt2-small (124M)"` (see section 7.5). Alternatively, to speed up the model training, consider using a GPU. The following supplementary section in this book's code repository lists several options for using cloud GPUs: https://mng.bz/EOEq.

The following table provides reference run times for training each model on various devices, including CPUs and GPUs, for GPT-2. Running this code on a compatible GPU requires no code changes and can significantly speed up training. For the results shown in this chapter, I used the GPT-2 medium model and trained it on an A100 GPU.

本章。对于微调过程本身，我们可以重用第 5 章中实现的损失计算和训练函数：

```
从第五章导入（
    calc_loss_loader
    训练模型简单
）
```

在开始训练之前，让我们计算训练集和验证集的初始损失：

```
    model.to(设备)
torch.manual_seed(123)

    使用 torch.no_grad():
        train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5) val_loss
 = calc_loss_loader()

            val_loader, 模型, 设备,          num_batches=5
    )

    打印("训练损失:", train_loss) 打印("验
证损失:", val_loss)
```

初始损失值如下；与之前一样，我们的目标是最小化损失：

```
训练损失：3.825908660888672 验证
            损失：3.7619335651397705
```

处理硬件限制

使用和训练更大的模型如 GPT-2 中型（3.55 亿参数）比使用较小的 GPT-2 模型（1.24 亿参数）计算量更大。如果您因硬件限制遇到问题，可以切换到较小的

将模型通过将 CHOOSE_MODEL = "gpt2-medium (355M)"更改为 CHOOSE_MODEL = "gpt2-small (124M)"进行更改（见第 7.5 节）。或者，为了加快模型训练速度，考虑使用 GPU。本书代码库中的以下补充部分列出了使用云 GPU 的几种选项：https://mng.bz/EOEq。

以下表格提供了在包括 CPU 和 GPU 在内的各种设备上训练每个模型的参考运行时间，针对 GPT-2。在兼容的 GPU 上运行此代码无需修改代码，并且可以显著加快训练速度。对于本章中显示的结果，我使用了 GPT-2 中型模型，并在 A100 GPU 上进行了训练。

Model name	Device	Run time for two epochs
gpt2-medium (355M)	CPU (M3 MacBook Air)	15.78 minutes
gpt2-medium (355M)	GPU (NVIDIA L4)	1.83 minutes
gpt2-medium (355M)	GPU (NVIDIA A100)	0.86 minutes
gpt2-small (124M)	CPU (M3 MacBook Air)	5.74 minutes
gpt2-small (124M)	GPU (NVIDIA L4)	0.69 minutes
gpt2-small (124M)	GPU (NVIDIA A100)	0.39 minutes

With the model and data loaders prepared, we can now proceed to train the model. The code in listing 7.8 sets up the training process, including initializing the optimizer, setting the number of epochs, and defining the evaluation frequency and starting context to evaluate generated LLM responses during training based on the first validation set instruction (`val_data[0]`) we looked at in section 7.5.

Listing 7.8 Instruction fine-tuning the pretrained LLM

```
import time
start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1
)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

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

The following output displays the training progress over two epochs, where a steady decrease in losses indicates improving ability to follow instructions and generate appropriate responses:

```
Ep 1 (Step 000000): Train loss 2.637, Val loss 2.626
Ep 1 (Step 000005): Train loss 1.174, Val loss 1.103
Ep 1 (Step 000010): Train loss 0.872, Val loss 0.944
Ep 1 (Step 000015): Train loss 0.857, Val loss 0.906
...
```

型号名称	设备	运行两个 epoch 的时间
gpt2-medium（355M）	CPU（M3 MacBook Air）	15.78 分钟
gpt2-medium（355M）	GPU（NVIDIA L4）	1.83 分钟
gpt2-medium（355M）	GPU（NVIDIA A100）	0.86 分钟
gpt2 小型（124M）	CPU（M3 MacBook Air）	5.74 分钟
gpt2 小型（124M）	GPU（NVIDIA L4）	0.69 分钟
gpt2 小型（124M）	GPU（NVIDIA A100）	0.39 分钟

有了模型和数据加载器准备就绪，我们现在可以开始训练模型。列表 7.8 中的代码设置了训练过程，包括初始化优化器、设置训练轮数以及定义评估频率和起始上下文，以便在训练过程中根据我们在 7.5 节中查看的第一个验证集指令（val_data[0]）评估生成的LLM响应。

列表 7.8 指令微调预训练的LLM

```
导入 time
    start_time = 时间.time()
torch.manual_seed(123) optimizer =
torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1)
num_epochs = 2


    训练损失，验证损失，已见令牌 = 简单训练模型()
    模型，训练加载器，验证加载器，优化器，设备，num_epochs=num_epochs,
eval_freq=5, eval_iter=5, 开始上下文=format_input(val_data[0]), 分词器
=tokenizer


    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。")
```

以下输出显示了两个 epoch 的训练进度，其中损失值的稳定下降表明了更好地遵循指令和生成适当回应的能力：

```
    第 1 集（步骤 000000）：训练损失 2.637，验证损失 2.626 第 1
集（步骤 000005）：训练损失 1.174，验证损失 1.103 第 1 集（步骤
000010）：训练损失 0.872，验证损失 0.944 第 1 集（步骤
000015）：训练损失 0.857，验证损失 0.906 ...
```

```
Ep 1 (Step 000115): Train loss 0.520, Val loss 0.665
Below is an instruction that describes a task. Write a response that
appropriately completes the request.  ### Instruction: Convert the
active sentence to passive: 'The chef cooks the meal every day.'
### Response: The meal is prepared every day by the chef.<|endoftext|>
The following is an instruction that describes a task.
Write a response that appropriately completes the request.
### Instruction: Convert the active sentence to passive:
Ep 2 (Step 000120): Train loss 0.438, Val loss 0.670
Ep 2 (Step 000125): Train loss 0.453, Val loss 0.685
Ep 2 (Step 000130): Train loss 0.448, Val loss 0.681
Ep 2 (Step 000135): Train loss 0.408, Val loss 0.677
...
Ep 2 (Step 000230): Train loss 0.300, Val loss 0.657
Below is an instruction that describes a task. Write a response
that appropriately completes the request.  ### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal
every day.'  ### Response: The meal is cooked every day by the
chef.<|endoftext|>The following is an instruction that describes
a task. Write a response that appropriately completes the request.
### Instruction: What is the capital of the United Kingdom
Training completed in 0.87 minutes.
```

The training output shows that the model is learning effectively, as we can tell based on the consistently decreasing training and validation loss values over the two epochs. This result suggests that the model is gradually improving its ability to understand and follow the provided instructions. (Since the model demonstrated effective learning within these two epochs, extending the training to a third epoch or more is not essential and may even be counterproductive as it could lead to increased overfitting.)

Moreover, the generated responses at the end of each epoch let us inspect the model's progress in correctly executing the given task in the validation set example. In this case, the model successfully converts the active sentence `"The chef cooks the meal every day."` into its passive voice counterpart: `"The meal is cooked every day by the chef."`

We will revisit and evaluate the response quality of the model in more detail later. For now, let's examine the training and validation loss curves to gain additional insights into the model's learning process. For this, we use the same `plot_losses` function we used for pretraining:

```
from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

From the loss plot shown in figure 7.17, we can see that the model's performance on both the training and validation sets improves substantially over the course of training. The rapid decrease in losses during the initial phase indicates that the model quickly learns meaningful patterns and representations from the data. Then, as training progresses to the second epoch, the losses continue to decrease but at a slower

以下是一个描述任务的指令。请适当完成请求。

指令：将主动句转换为被动句：'The chef cooks the meal every day.'
回复：The meal is cooked every day by the chef. 编写一个适当完成请求的回复。###
指示：英国的首都是哪里？训练完成用时 0.87 分钟。

训练输出显示模型正在有效学习，我们可以根据两个 epoch 内持续下降的训练和验证损失值来判断。这一结果表明，模型正在逐渐提高其理解和遵循提供指令的能力。（由于模型在这两个 epoch 内展示了有效的学习，将训练扩展到第三个 epoch 或更多不是必要的，甚至可能适得其反，因为它可能导致过拟合。）

此外，每个 epoch 结束时生成的响应使我们能够检查模型在验证集示例中正确执行给定任务的过程。在这种情况下，模型成功地将主动句子"The chef cooks the"转换为...

每天一餐。 每天由...烹饪的这顿饭

这位厨师。

我们将稍后更详细地回顾和评估该模型的响应质量。

目前，让我们分析训练和验证损失曲线，以深入了解模型的学习过程。为此，我们使用与预训练相同的 plot_losses 函数：

从第 05 章导入 plot_losses 函数，epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))，绘制 train_losses 的图像。

val_losses)

从图 7.17 所示的损失图中，我们可以看到，在训练过程中，模型在训练集和验证集上的性能显著提高。初始阶段损失的快速下降表明模型迅速从数据中学习到有意义的模式和表示。然后，随着训练进展到第二个时期，损失继续下降，但下降速度变慢。

rate, suggesting that the model is fine-tuning its learned representations and converging to a stable solution.

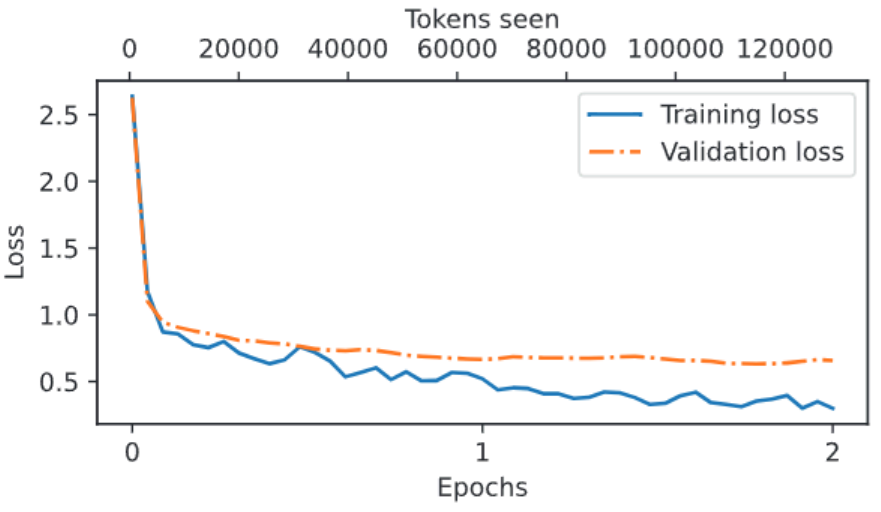

Figure 7.17 The training and validation loss trends over two epochs. The solid line represents the training loss, showing a sharp decrease before stabilizing, while the dotted line represents the validation loss, which follows a similar pattern.

While the loss plot in figure 7.17 indicates that the model is training effectively, the most crucial aspect is its performance in terms of response quality and correctness. So, next, let's extract the responses and store them in a format that allows us to evaluate and quantify the response quality.

> **Exercise 7.3 Fine-tuning on the original Alpaca dataset**
>
> The Alpaca dataset, by researchers at Stanford, is one of the earliest and most popular openly shared instruction datasets, consisting of 52,002 entries. As an alternative to the `instruction-data.json` file we use here, consider fine-tuning an LLM on this dataset. The dataset is available at https://mng.bz/NBnE.
>
> This dataset contains 52,002 entries, which is approximately 50 times more than those we used here, and most entries are longer. Thus, I highly recommend using a GPU to conduct the training, which will accelerate the fine-tuning process. If you encounter out-of-memory errors, consider reducing the `batch_size` from 8 to 4, 2, or even 1. Lowering the `allowed_max_length` from 1,024 to 512 or 256 can also help manage memory problems.

7.7 Extracting and saving responses

Having fine-tuned the LLM on the training portion of the instruction dataset, we are now ready to evaluate its performance on the held-out test set. First, we extract the model-generated responses for each input in the test dataset and collect them for manual analysis, and then we evaluate the LLM to quantify the quality of the responses, as highlighted in figure 7.18.

损失持续下降，但下降速度变慢，表明模型正在微调其学习到的表示，并收敛到一个稳定的解。

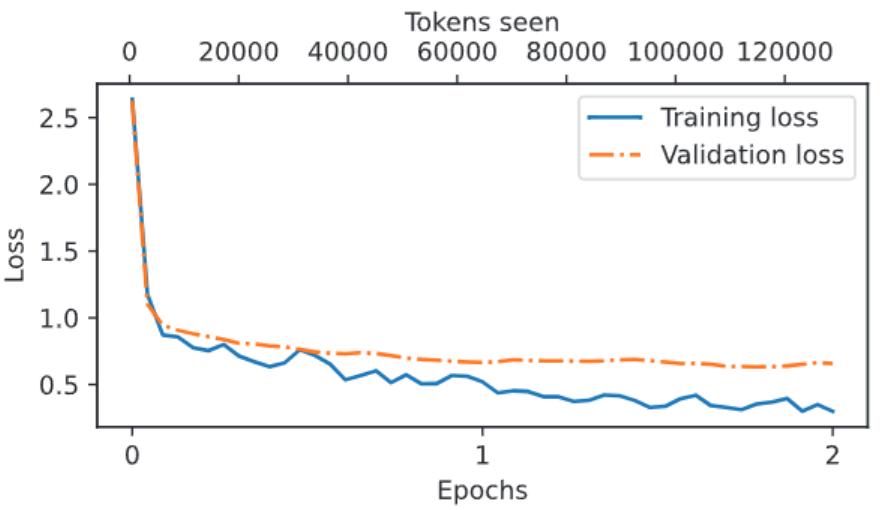

图 7.17 两个 epoch 的训练和验证损失趋势。实线表示训练损失，显示在稳定前急剧下降，而虚线表示验证损失，遵循相似的模式。

虽然图 7.17 中的损失曲线表明模型训练有效，但最关键的是其在响应质量和正确性方面的表现。因此，接下来，我们将提取响应并将它们存储在一个允许我们评估和量化响应质量的格式中。

练习 7.3 在原始 Alpaca 数据集上进行微调

斯坦福研究人员制作的羊驼数据集是早期且最受欢迎的开源指令数据集之一，包含 52,002 条条目。作为我们这里使用的 instruction-data.json 文件的替代方案，可以考虑在这个数据集上微调LLM。数据集可在 https://mng.bz/NBnE 获取。

该数据集包含 52,002 条记录，大约是我们在这里使用的 50 倍，且大多数记录更长。因此，我强烈建议使用 GPU 进行训练，这将加速微调过程。如果您遇到内存不足错误，请考虑将 batch_size 从 8 减少到 4、2 甚至 1。将允许的最大长度从 1,024 降低到 512 或 256 也可以帮助管理内存问题。

7.7 提取并保存响应

对指令数据集的训练部分进行微调后，我们现在准备评估其在保留测试集上的性能。首先，我们从测试数据集中的每个输入中提取模型生成的响应，并收集它们以进行人工分析，然后评估LLM以量化响应的质量，如图 7.18 所示。

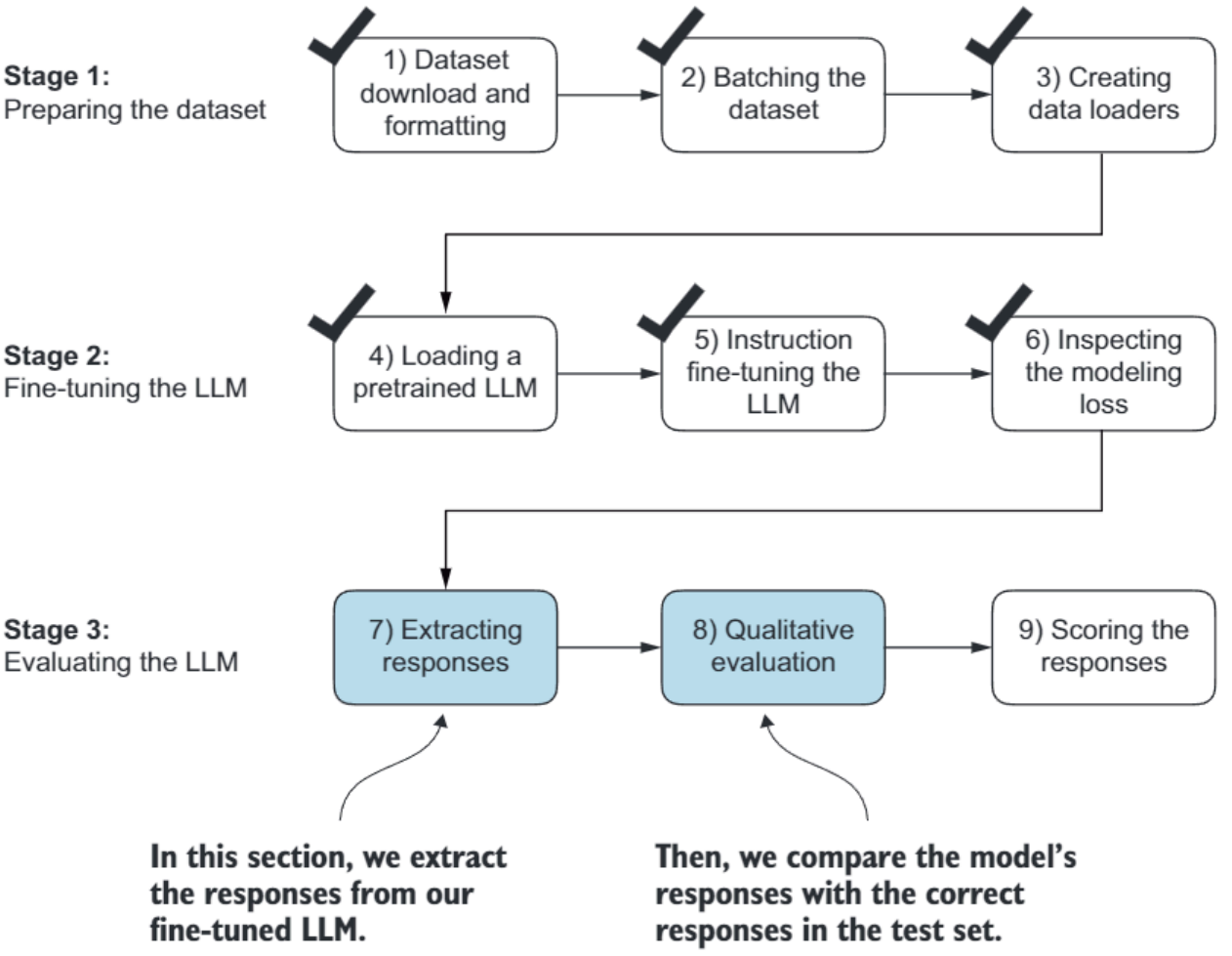

Figure 7.18 The three-stage process for instruction fine-tuning the LLM. In the first two steps of stage 3, we extract and collect the model responses on the held-out test dataset for further analysis and then evaluate the model to quantify the performance of the instruction-fine-tuned LLM.

To complete the response instruction step, we use the `generate` function. We then print the model responses alongside the expected test set answers for the first three test set entries, presenting them side by side for comparison:

```
torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
```

Iterates over the first three test set samples

Uses the generate function imported in section 7.5

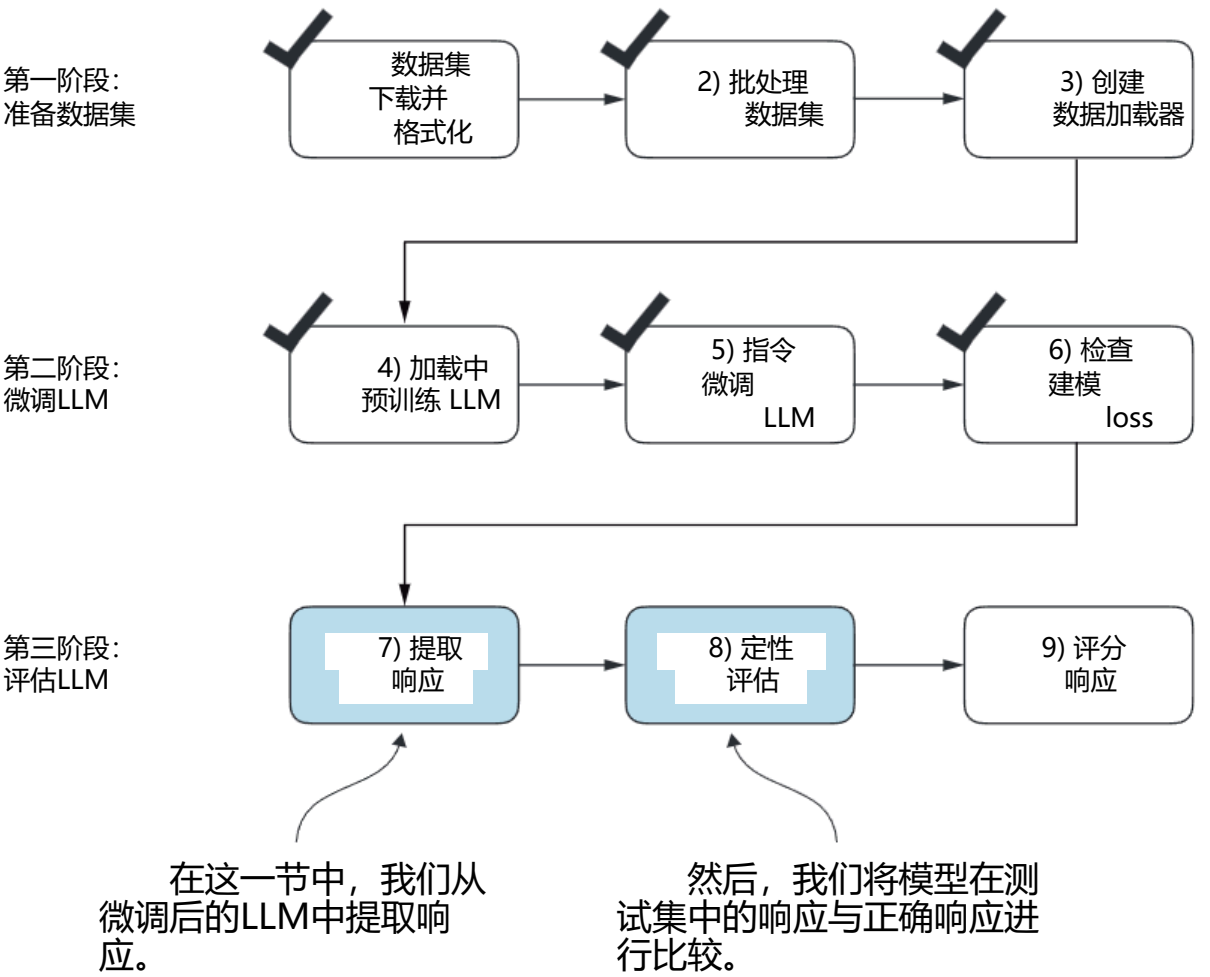

图 7.18 指令微调的三个阶段过程LLM。在第三阶段的前两步中，我们提取并收集在保留测试数据集上的模型响应，以进行进一步分析，然后评估模型以量化指令微调LLM的性能。

为了完成响应指令步骤，我们使用生成函数。然后，我们将模型响应与预期的测试集答案并排打印，对于前三个测试集条目，将它们并排呈现以供比较：

```
torch 手动设置随机种子(123)

for 条目 in 测试数据[:3]:
    输入文本格式化后，生成的 token ID 列表：输入文本格式化后，生成的 token ID 列表：
    model=model, idx=text_to_token_ids(input_text, tokenizer).to(device),
max_new_tokens=256, context_size=BASE_CONFIG["context_length"],
eos_id=50256 ) 生成文本 = token_ids_to_text(token_ids, tokenizer)

    response_text = (
    生成的文本[输入文本的长度:] .替换
("### 响应：") .去除空格() )
```

遍历前三个测试集样本

使用第 7.5 节中导入的 generate 函数

```
print(input_text)
print(f"\nCorrect response:\n>> {entry['output']}")
print(f"\nModel response:\n>> {response_text.strip()}")
print("-------------------------------------")
```

As mentioned earlier, the `generate` function returns the combined input and output text, so we use slicing and the `.replace()` method on the `generated_text` contents to extract the model's response. The instructions, followed by the given test set response and model response, are shown next.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Rewrite the sentence using a simile.

Input:

The car is very fast.

Correct response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

What type of cloud is typically associated with thunderstorms?

Correct response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
    打印(input_text) 打印(f"\n 正确响应：\n>> {entry['output']}")
打印(f"\n 模型响应：\n>> {response_text.strip()}") 打印("----------
--------------------------")
```

如前所述，generate 函数返回输入和输出的组合文本，因此我们使用切片和.replace()方法对生成的文本内容进行处理，以提取模型的响应。接下来展示的是指令，以及给定的测试集响应和模型响应。

以下是一份描述任务的说明。撰写一个适当完成请求的回复。

指令：

用比喻重写这个句子。

输入：

这辆车非常快。

正确响应：

汽车的速度像闪电一样快。

模型响应：

汽车的速度像子弹一样快。

以下是一份描述任务的说明。撰写一个适当完成请求的回复。

指令：

什么是与雷暴通常相关的云类型?

正确响应：

雷暴通常与积雨云相关。

模型响应：

雷暴相关的云是积云。

以下是一份描述任务的说明。撰写一个适当完成请求的回复。

Instruction:

Name the author of 'Pride and Prejudice.'

Correct response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

As we can see based on the test set instructions, given responses, and the model's responses, the model performs relatively well. The answers to the first and last instructions are clearly correct, while the second answer is close but not entirely accurate. The model answers with "cumulus cloud" instead of "cumulonimbus," although it's worth noting that cumulus clouds can develop into cumulonimbus clouds, which are capable of producing thunderstorms.

Most importantly, model evaluation is not as straightforward as it is for completion fine-tuning, where we simply calculate the percentage of correct spam/non-spam class labels to obtain the classification's accuracy. In practice, instruction-fine-tuned LLMs such as chatbots are evaluated via multiple approaches:

- Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; https://arxiv.org/abs/2009.03300), which test the general knowledge of a model.
- Human preference comparison to other LLMs, such as LMSYS chatbot arena (https://arena.lmsys.org).
- Automated conversational benchmarks, where another LLM like GPT-4 is used to evaluate the responses, such as AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/).

In practice, it can be useful to consider all three types of evaluation methods: multiple-choice question answering, human evaluation, and automated metrics that measure conversational performance. However, since we are primarily interested in assessing conversational performance rather than just the ability to answer multiple-choice questions, human evaluation and automated metrics may be more relevant.

Conversational performance

Conversational performance of LLMs refers to their ability to engage in human-like communication by understanding context, nuance, and intent. It encompasses skills such as providing relevant and coherent responses, maintaining consistency, and adapting to different topics and styles of interaction.

指令：

《傲慢与偏见》的作者是谁。

正确响应：

简·奥斯汀

模型响应：

《傲慢与偏见》的作者是简·奥斯汀。

根据测试集说明、给定响应和模型的响应，我们可以看到模型表现相对较好。第一和最后一个指令的答案明显正确，而第二个答案接近但并不完全准确。模型回答为“积云”，而不是“积雨云”，尽管值得注意的是，积云可以发展成为积雨云，后者能够产生雷暴。

最重要的是，模型评估并不像完成微调那样简单，在完成微调中，我们只需计算正确垃圾邮件/非垃圾邮件类别标签的百分比来获得分类的准确率。在实践中，例如聊天机器人这样的指令微调LLMs通过多种方法进行评估：

- 简答和多项选择题基准，如衡量大规模多任务语言理解（MMLU；https://arxiv.org/abs/2009.03300），用于测试模型的常识。
- 人类对其他LLMs的偏好比较，例如 LMSYS 聊天机器人竞技场（https://arena.lmsys.org）。
- 自动化对话基准，其中使用另一个类似于 GPT-4 的LLM来评估响应，例如 AlpacaEval（https://tatsu-lab.github.io/alpaca_eval/）。

在实践中，考虑所有三种评估方法可能很有用：多项选择题回答、人工评估和衡量对话性能的自动化指标。然而，由于我们主要关注评估对话性能而不是仅仅回答多项选择题的能力，人工评估和自动化指标可能更为相关。

对话表现

LLMs的对话表现指的是他们通过理解语境、细微差别和意图进行类似人类交流的能力。它包括提供相关且连贯的回应、保持一致性以及适应不同主题和互动风格等技能。

Human evaluation, while providing valuable insights, can be relatively laborious and time-consuming, especially when dealing with a large number of responses. For instance, reading and assigning ratings to all 1,100 responses would require a significant amount of effort.

So, considering the scale of the task at hand, we will implement an approach similar to automated conversational benchmarks, which involves evaluating the responses automatically using another LLM. This method will allow us to efficiently assess the quality of the generated responses without the need for extensive human involvement, thereby saving time and resources while still obtaining meaningful performance indicators.

Let's employ an approach inspired by AlpacaEval, using another LLM to evaluate our fine-tuned model's responses. However, instead of relying on a publicly available benchmark dataset, we use our own custom test set. This customization allows for a more targeted and relevant assessment of the model's performance within the context of our intended use cases, represented in our instruction dataset.

To prepare the responses for this evaluation process, we append the generated model responses to the `test_set` dictionary and save the updated data as an `"instruction-data-with-response.json"` file for record keeping. Additionally, by saving this file, we can easily load and analyze the responses in separate Python sessions later on if needed.

The following code listing uses the `generate` method in the same manner as before; however, we now iterate over the entire `test_set`. Also, instead of printing the model responses, we add them to the `test_set` dictionary.

Listing 7.9 Generating test set responses

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)

    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file:
    json.dump(test_data, file, indent=4)
```

indent for pretty-printing

人类评估虽然能提供有价值的见解，但相对费力且耗时，尤其是在处理大量回复时。例如，阅读并给所有 1,100 个回复评分需要大量的努力。

因此，考虑到当前任务的规模，我们将实施一种类似于自动化对话基准的方法，该方法涉及使用另一个LLM自动评估响应。这种方法将使我们能够高效地评估生成响应的质量，无需大量人工参与，从而节省时间和资源，同时仍能获得有意义的性能指标。

让我们采用受 AlpacaEval 启发的方案，使用另一个LLM来评估我们微调模型的响应。然而，我们不是依赖于公开的基准数据集，而是使用我们自己的定制测试集。这种定制化允许在目标使用案例的上下文中，针对模型性能进行更精确和相关的评估，这些案例由我们的指令数据集表示。

为准备此评估过程的响应，我们将生成的模型响应追加到 test_set 字典中，并将更新后的数据保存为“instruction-data-with-response.json”文件以备记录。此外，通过保存此文件，我们可以在需要时轻松地在单独的 Python 会话中加载和分析响应。

以下代码列表使用与之前相同的方式调用 generate 方法；然而，我们现在遍历整个 test_set。此外，我们不再打印模型响应，而是将它们添加到 test_set 字典中。

列表 7.9 生成测试集响应

```
from  tqdm  导入     tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
     输入文本: input_格式化输入（entry）

     token_ids = generate(
     model=model, idx=text_to_token_ids(input_text, tokenizer).to(device),
 max_new_tokens=256, context_size=BASE_CONFIG["context_length"],
 eos_id=50256 ) generated_text = token_ids_to_text(token_ids, tokenizer)




     response_text = (
     生成的文本长度减去输入文本长度后的部分，替换掉"###
 Response:"，然后去除空白字符。测试数据中第 i 个元素
 的"model_response"字段被设置为 response_text。



with open("instruction-data-with-response.json", "w") as 文件:
     json.dump(test_data, file,       缩进=4）
```

缩进以实现美观打印

Processing the dataset takes about 1 minute on an A100 GPU and 6 minutes on an M3 MacBook Air:

```
100%|██████████| 110/110 [01:05<00:00,  1.68it/s]
```

Let's verify that the responses have been correctly added to the `test_set` dictionary by examining one of the entries:

```
print(test_data[0])
```

The output shows that the `model_response` has been added correctly:

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

Finally, we save the model as `gpt2-medium355M-sft.pth` file to be able to reuse it in future projects:

```
import re

file_name = f"{re.sub(r'[ ()]', '', CHOOSE_MODEL) }-sft.pth"
torch.save(model.state_dict(), file_name)
print(f"Model saved as {file_name}")
```

Removes white spaces and parentheses from file name

The saved model can then be loaded via `model.load_state_dict(torch.load("gpt2-medium355M-sft.pth"))`.

7.8 Evaluating the fine-tuned LLM

Previously, we judged the performance of an instruction-fine-tuned model by looking at its responses on three examples of the test set. While this gives us a rough idea of how well the model performs, this method does not scale well to larger amounts of responses. So, we implement a method to automate the response evaluation of the fine-tuned LLM using another, larger LLM, as highlighted in figure 7.19.

To evaluate test set responses in an automated fashion, we utilize an existing instruction-fine-tuned 8-billion-parameter Llama 3 model developed by Meta AI. This model can be run locally using the open source Ollama application (https://ollama.com).

> **NOTE** Ollama is an efficient application for running LLMs on a laptop. It serves as a wrapper around the open source llama.cpp library (https://github.com/ggerganov/llama.cpp), which implements LLMs in pure C/C++ to maximize efficiency. However, Ollama is only a tool for generating text using LLMs (inference) and does not support training or fine-tuning LLMs.

处理数据集在 A100 GPU 上大约需要 1 分钟，在 M3 MacBook Air 上需要 6 分钟

```
100%|███████████| 110/[01005<00:00,     1.68it/s]
```

让我们通过检查其中一个条目来验证响应是否已正确添加到 test_set 字典中：

```
打印(test_data[0])
```

输出显示模型响应已正确添加：

```
将句子用比喻重新改写。这辆车非常快。
  'output': '这辆车和闪电一样快。', 'model_response':
'这辆车和闪电一样快。'                                子弹。'}
```

最后，我们将模型保存为 gpt2-medium355M-sft.pth 文件，以便在未来的项目中重复使用：

```
导入    re

file_name = f"{re.sub(r'[ ()]', '', CHOOSE_MODEL)}-sft.pth"
torch.save(model.state_dict(), file_name) 打印(f"模型已保存为
{file_name}")
```

移除空白文件名中的括号

保存的模型可以通过 model.load_state_dict(torch.load("gpt2"))加载。
-medium355M-sft.pth")).

7.8 评估经过微调的LLM

之前，我们通过观察测试集三个示例的响应来判断指令微调模型的性能。虽然这让我们对模型的表现有一个大致的了解，但这种方法并不适用于大量响应。因此，我们实现了一种方法来自动评估微调的LLM的响应，使用另一个更大的LLM，如图 7.19 所示。

为了以自动化的方式评估测试集的响应，我们使用由 Meta AI 开发的现有指令微调的 80 亿参数 Llama 3 模型。此模型可以使用开源的 Ollama 应用程序（https://ollama.com）在本地运行。

Ollama 是一个在笔记本电脑上运行 LLMs 的高效应用程序。它作为开源库 llama.cpp（https://github.com/ggerganov/llama.cpp）的包装器，该库使用纯 C/C++ 实现 LLMs 以最大化效率。然而，Ollama 只是一个用于生成文本的 LLMs（推理）工具，不支持训练或微调 LLMs。

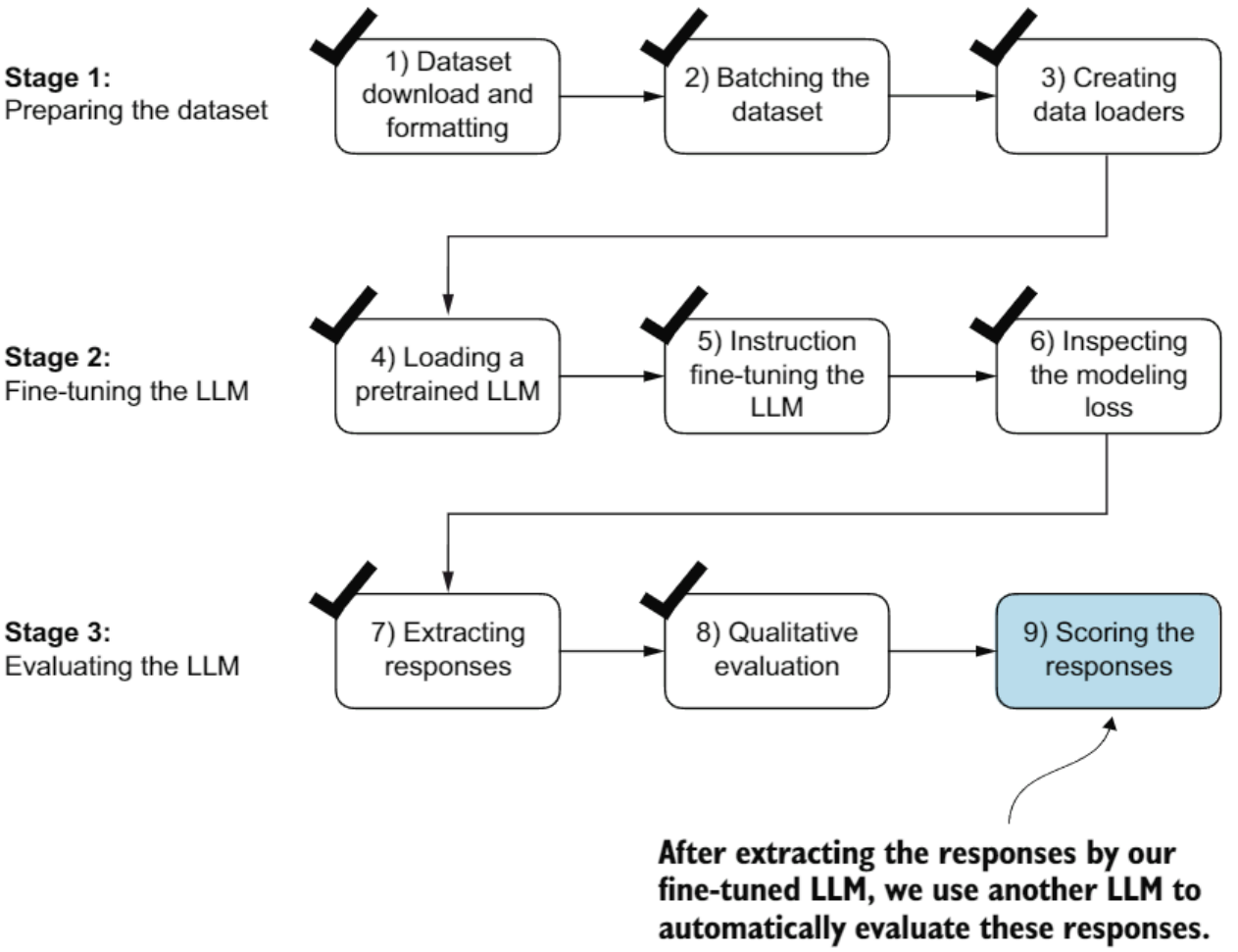

Figure 7.19 The three-stage process for instruction fine-tuning the LLM. In this last step of the instruction-fine-tuning pipeline, we implement a method to quantify the performance of the fine-tuned model by scoring the responses it generated for the test.

Using larger LLMs via web APIs

The 8-billion-parameter Llama 3 model is a very capable LLM that runs locally. However, it's not as capable as large proprietary LLMs such as GPT-4 offered by OpenAI. For readers interested in exploring how to utilize GPT-4 through the OpenAI API to assess generated model responses, an optional code notebook is available within the supplementary materials accompanying this book at https://mng.bz/BgEv.

To execute the following code, install Ollama by visiting https://ollama.com and follow the provided instructions for your operating system:

- *For macOS and Windows users*—Open the downloaded Ollama application. If prompted to install command-line usage, select Yes.
- *For Linux users*—Use the installation command available on the Ollama website.

Before implementing the model evaluation code, let's first download the Llama 3 model and verify that Ollama is functioning correctly by using it from the command-line terminal. To use Ollama from the command line, you must either start the Ollama application or run `ollama serve` in a separate terminal, as shown in figure 7.20.

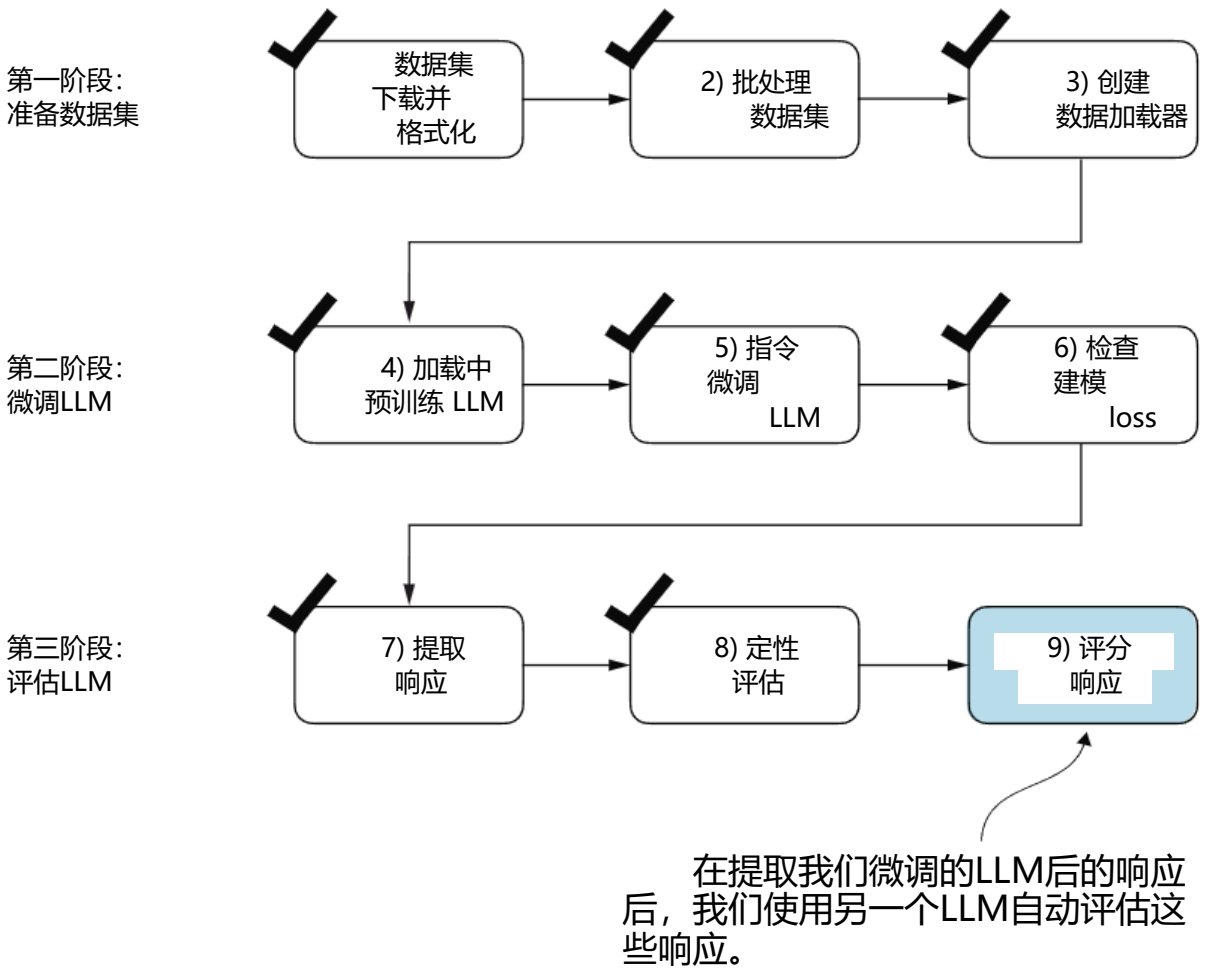

图 7.19 指令微调的三个阶段过程LLM。在这个指令微调管道的最后一步，我们通过评分测试中生成的响应来量化微调模型的表现。

使用更大的LLMs通过 Web API

8 亿参数的 Llama 3 模型是一个非常强大的本地运行模型。然而，它并不像 OpenAI 提供的 GPT-4 等大型专有模型那样强大。对于有兴趣探索如何通过 OpenAI API 利用 GPT-4 来评估生成模型响应的读者，本书附带的补充材料中提供了一个可选的代码笔记本，可在 https://mng.bz/BgEv 找到。

执行以下代码，请访问 https://ollama.com 安装 Ollama，并按照您操作系统的说明进行操作：

- 对于 macOS 和 Windows 用户——打开下载的 Ollama 应用程序。如果提示安装命令行使用，请选择是。
- 对于 Linux 用户——使用 Ollama 网站上提供的安装命令。

在实现模型评估代码之前，我们首先下载 Llama 3 模型，并通过命令行终端使用它来验证 Ollama 是否正常工作。要从命令行使用 Ollama，您必须启动 Ollama 应用程序或在单独的终端中运行 ollama serve，如图 7.20 所示。

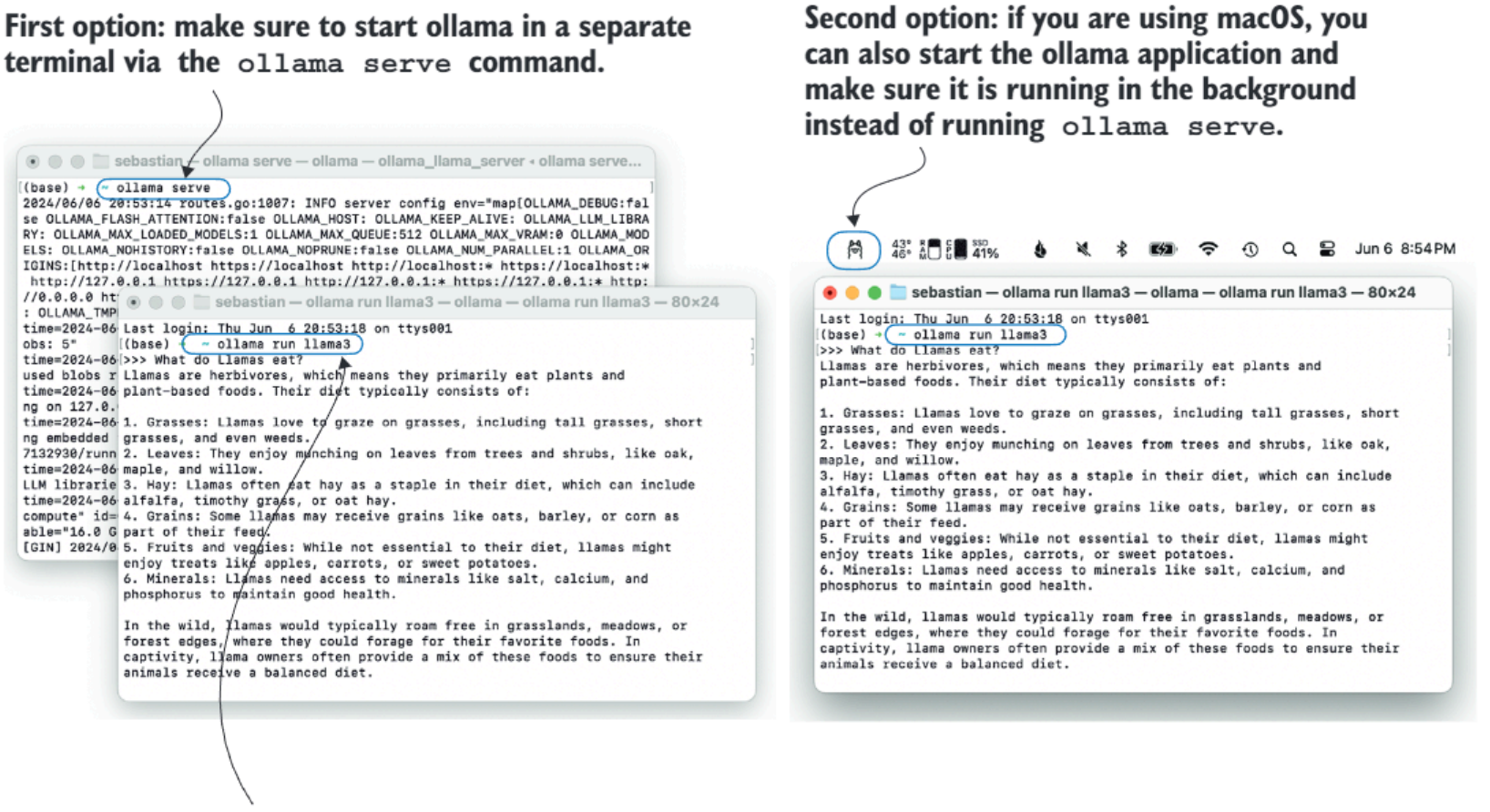

Figure 7.20 Two options for running Ollama. The left panel illustrates starting Ollama using `ollama serve`. The right panel shows a second option in macOS, running the Ollama application in the background instead of using the `ollama serve` command to start the application.

With the Ollama application or `ollama serve` running in a different terminal, execute the following command on the command line (not in a Python session) to try out the 8-billion-parameter Llama 3 model:

```
ollama run llama3
```

The first time you execute this command, this model, which takes up 4.7 GB of storage space, will be automatically downloaded. The output looks like the following:

```
pulling manifest
pulling 6a0746a1ec1a... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕| 4.7 GB
pulling 4fa551d4f938... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  12 KB
pulling 8ab4849b038c... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  254 B
pulling 577073ffcc6c... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  110 B
pulling 3f8eb4da87fa... 100% |▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕|  485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

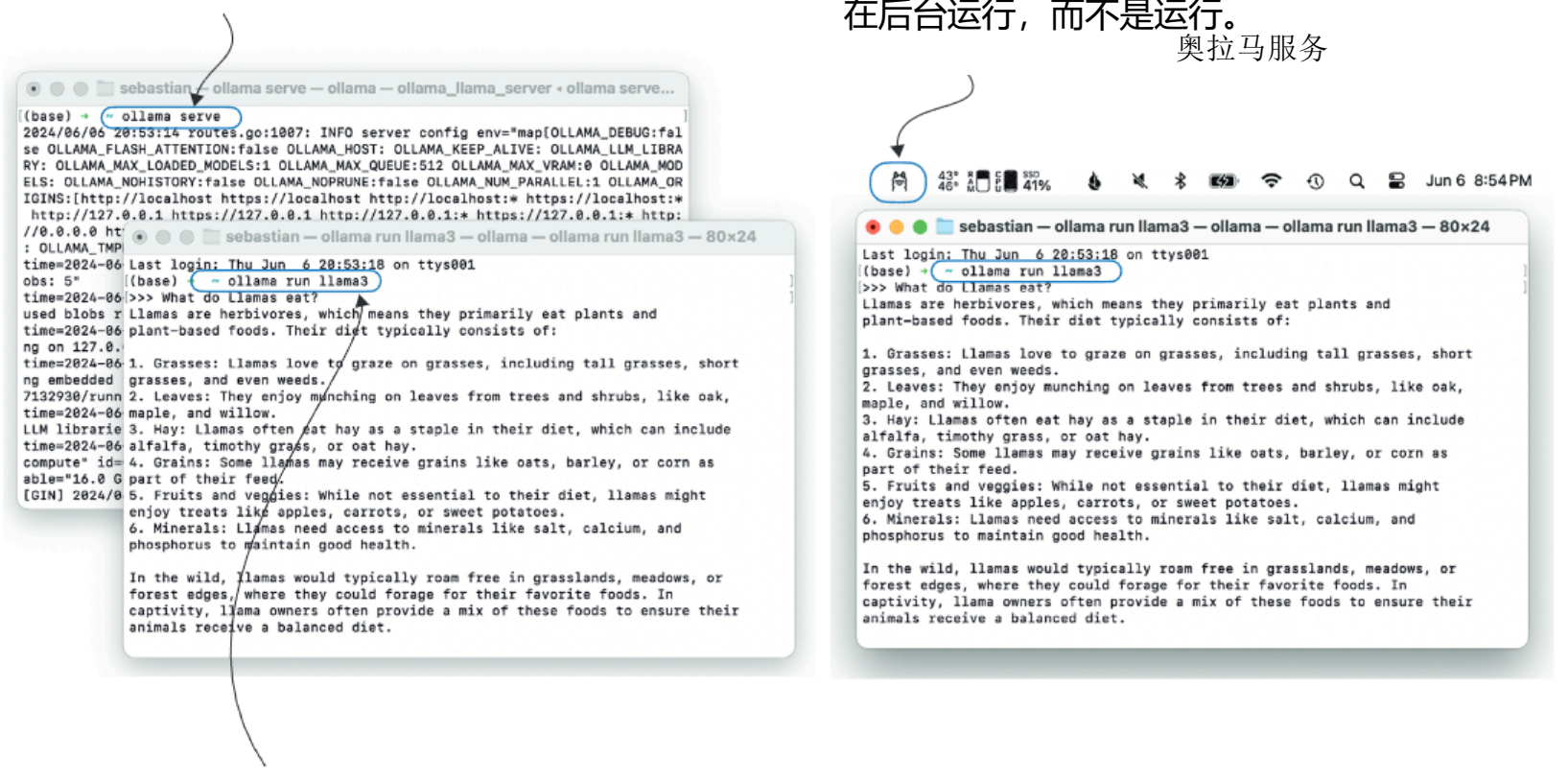

图 7.20 运行 Ollama 的两种选项。左侧面板展示了使用 ollama serve 启动 Ollama 的过程。右侧面板显示了在 macOS 中，通过在后台运行 Ollama 应用程序而不是使用 ollama serve 命令来启动应用程序的另一种选项。

使用 Ollama 应用程序或 ollama serve 在不同的终端中运行，请在命令行（非 Python 会话）中执行以下命令以尝试使用 80 亿参数的 Llama 3 模型：

```
驴马 run 拉马 3
```

第一次执行此命令时，将自动下载占用 4.7GB 存储空间的此模型。输出如下所示：

```
拉取清单 拉取 6a0746a1ec1a... 100% |
████████████████████████████████| 4.7 GB 拉取 4fa551d4f938... 100%
|████████████████████████████████| 12 KB 拉取 8ab4849b038c... 100%
|████████████████████████████████| 254 B 拉取 577073ffcc6c... 100%
|████████████████████████████████| 110 B 拉取 3f8eb4da87fa... 100%
|████████████████████████████████| 485 B 验证 sha256 校验和 编写清单 删除任何未使用的层 成功
```

Alternative Ollama models

The `llama3` in the `ollama run llama3` command refers to the instruction-fine-tuned 8-billion-parameter Llama 3 model. Using Ollama with the `llama3` model requires approximately 16 GB of RAM. If your machine does not have sufficient RAM, you can try using a smaller model, such as the 3.8-billion-parameter `phi3` model via `ollama run llama3`, which only requires around 8 GB of RAM.

For more powerful computers, you can also use the larger 70-billion-parameter Llama 3 model by replacing `llama3` with `llama3:70b`. However, this model requires significantly more computational resources.

Once the model download is complete, we are presented with a command-line interface that allows us to interact with the model. For example, try asking the model, "What do llamas eat?"

```
>>> What do llamas eat?
Llamas are ruminant animals, which means they have a four-chambered
stomach and eat plants that are high in fiber. In the wild,
llamas typically feed on:

1. Grasses: They love to graze on various types of grasses, including tall
grasses, wheat, oats, and barley.
```

Note that the response you see might differ since Ollama is not deterministic as of this writing.

You can end this `ollama run llama3` session using the input `/bye`. However, make sure to keep the `ollama serve` command or the Ollama application running for the remainder of this chapter.

The following code verifies that the Ollama session is running properly before we use Ollama to evaluate the test set responses:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(["name"]):
        if process_name in proc.info["name"]:
            running = True
            break
    return running

ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError(
        "Ollama not running. Launch ollama before proceeding."
)
print("Ollama running:", check_if_running("ollama"))
```

替代 Ollama 模型

llama3 在 ollama 运行 llama3 命令指的是指令微调的 80 亿参数 Llama 3 模型。使用 ollama 与 llama3 模型需要大约 16GB 的 RAM。如果您的机器没有足够的 RAM，您可以尝试使用较小的模型，例如通过 ollama 使用 3.8 亿参数的 phi3 模型

运行 llama3，它只需要大约 8 GB 的 RAM。

对于更强大的计算机，您还可以通过将 llama3 替换为 llama3:70b 使用更大的 70 亿参数的 Llama 3 模型。然而，此模型需要显著更多的计算资源。

一旦模型下载完成，我们将看到一个命令行界面，允许我们与模型交互。例如，试着问模型，“羊驼吃什么？”

```
羊驼吃什么？羊驼是反刍动物，这意味着它们有一个四室胃，并且吃富含纤维的植物。
在野外，羊驼通常以以下植物为食：

草类：它们喜欢在各种各样的草地上吃草，包括高草、小麦、燕麦和大麦。
```

请注意，您看到的响应可能会有所不同，因为截至本文写作时，Ollama 并非确定性算法。

您可以使用输入/bye 结束 ollama run llama3 会话。然而，请确保在本章剩余部分保持 ollama serve 命令或 Ollama 应用程序运行。

以下代码在用 Ollama 评估测试集响应之前，验证 Ollama 会话是否运行正常：

```
导入     psutil

def 检查是否正在运行(process_name):
     运行 = False for proc in
 psutil.process_iter(["名称"]):
          如果进程名称在 proc.info["name"]中：
               运行 = True
               break
     返回运行中断

ollama_running = 驴马检查是否正在运行("ollama")

如果不 ollama_running:
     引发 RuntimeError(
          Ollama 未运行。启动                驴马      在之前    进行中。
)
打印("Ollama")    运行中:检查是否正在运行("ollama"))
```

Ensure that the output from executing the previous code displays `Ollama running: True`. If it shows `False`, verify that the `ollama serve` command or the Ollama application is actively running.

Running the code in a new Python session

If you already closed your Python session or if you prefer to execute the remaining code in a different Python session, use the following code, which loads the instruction and response data file we previously created and redefines the `format_input` function we used earlier (the `tqdm` progress bar utility is used later):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )
    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

An alternative to the `ollama run` command for interacting with the model is through its REST API using Python. The `query_model` function shown in the following listing demonstrates how to use the API.

Listing 7.10 Querying a local Ollama model

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {                    <-- Creates the data payload as a dictionary
        "model": model,
        "messages": [
            {"role": "user", "content": prompt}
        ],
        "options": {            <-- Settings for deterministic responses
            "seed": 123,
```

确保执行上一段代码的输出显示 Ollama 运行：True。如果显示 False，请验证 ollama serve 命令或 Ollama 应用程序是否正在运行。

运行新 Python 会话中的代码

如果您已经关闭了 Python 会话或您希望在不同的 Python 会话中执行剩余的代码，请使用以下代码，该代码加载我们之前创建的指令和响应数据文件，并重新定义了我们之前使用的 format_input 函数（稍后使用 tqdm 进度条实用工具）：

```
    import json from tqdm
import tqdm

    file_path = "instruction-data-with-response.json" 使用
open(file_path, "r") 作为文件
        测试数据     = json.load(文件)

    def 格式化输入(entry):
        instruction_text = (
        以下是一个描述任务的说明。请写一个适当完成请求的回复。### 说明：
     {entry['instruction']}


        输入文本：
input_text = (
### 输入：
{entry['input']}"
if entry["input"] else ""
```

一个与 ollama 运行命令交互模型的替代方法是使用其 REST API 并通过 Python。以下列表中所示的 query_model 函数演示了如何使用 API。

列表 7.10 查询本地 Ollama 模型

```
    导入     urllib.request

    def 查询模型(
         提示
    model="llama3",
url="http://localhost:11434/api/chat" ):

          data = {                                  创建数据负载作
               "model": 模型                        为字典
               "messages": [
               [{"角色": "用户", "内容": "提示"}], "选项":
           {

                                                    设置确定性响应
                    "seed": 123,
```

```
        "temperature": 0,
        "num_ctx": 2048
    }
}

payload = json.dumps(data).encode("utf-8")
request = urllib.request.Request(
    url,
    data=payload,
    method="POST"
)

request.add_header("Content-Type", "application/json")

response_data = ""
with urllib.request.urlopen(request) as response:
    while True:
        line = response.readline().decode("utf-8")
        if not line:
            break
        response_json = json.loads(line)
        response_data += response_json["message"]["content"]

return response_data
```

Converts the dictionary to a JSON-formatted string and encodes it to bytes

Creates a request object, setting the method to POST and adding necessary headers

Sends the request and captures the response

Before running the subsequent code cells in this notebook, ensure that Ollama is still running. The previous code cells should print `"Ollama running: True"` to confirm that the model is active and ready to receive requests.

The following is an example of how to use the `query_model` function we just implemented:

```
model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)
```

The resulting response is as follows:

```
Llamas are ruminant animals, which means they have a four-chambered
stomach that allows them to digest plant-based foods. Their diet
typically consists of:

1. Grasses: Llamas love to graze on grasses, including tall grasses,
short grasses, and even weeds.
...
```

Using the `query_model` function defined earlier, we can evaluate the responses generated by our fine-tuned model that prompts the Llama 3 model to rate our fine-tuned model's responses on a scale from 0 to 100 based on the given test set response as reference.

```
            "温度": 0
            "num_ctx": 2048
        }
    }


    payload = json.dumps(data).encode("utf-8")
request = urllib.request.Request()
        url,
        data=payload,
        method="POST"
    )


    request.add_header("Content-Type", "application/json")


    response_data = "" 使用 urllib.request.urlopen(request)
作为 response:
        while True:
            line = response.readline().decode("utf-8") 如果
         not line:
                break
中断
            response_json = json.loads(line)
         response_data        += response_json["消息"]["内容"]


    返回 response_data
```

将字典转换为JSON 格式的字符串并将其编码为字节

创建一个请求对象，设置POST 方法添加必要的标题

发送请求和捕捉响应

在运行本笔记本中的后续代码单元格之前，请确保 Ollama 仍在运行。前面的代码单元格应打印“Ollama running: True”，以确认模型处于活动状态并准备好接收请求。

以下是我们刚刚实现的 query_model 函数的一个示例：

```
model = "llama3"
result = 查询模型("羊驼吃什么？", 模型) 打印(result)
```

结果响应如下：

```
    羊驼是反刍动物，这意味着它们有一个四室胃，这使得它们能够消化植物性食物。它们
的饮食通常包括：


    草类：羊驼喜欢在草地上吃草，包括高草、矮草甚至杂草。

    ...
```

使用之前定义的 query_model 函数，我们可以评估由我们的微调模型生成的响应，该模型提示 Llama 3 模型根据给定的测试集响应作为参考，在 0 到 100 的范围内对我们的微调模型响应进行评分。

First, we apply this approach to the first three examples from the test set that we previously examined:

```
for entry in test_data[:3]:
    prompt = (
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']}`, "
        f"score the model response `{entry['model_response']}`"
        f" on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])
    print("\nModel response:")
    print(">>", entry["model_response"])
    print("\nScore:")
    print(">>", query_model(prompt))
    print("\n-------------------------")
```

This code prints outputs similar to the following (as of this writing, Ollama is not fully deterministic, so the generated texts may vary):

Dataset response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Score:

>> I'd rate the model response "The car is as fast as a bullet." an 85 out of 100.

Here's why:

The response uses a simile correctly, comparing the speed of the car to something else (in this case, a bullet).

The comparison is relevant and makes sense, as bullets are known for their high velocity.

The phrase "as fast as" is used correctly to introduce the simile.

The only reason I wouldn't give it a perfect score is that some people might find the comparison slightly less vivid or evocative than others. For example, comparing something to lightning (as in the original response) can be more dramatic and attention grabbing. However, "as fast as a bullet" is still a strong and effective simile that effectively conveys the idea of the car's speed.

Overall, I think the model did a great job!

首先，我们将这种方法应用于我们之前检查过的测试集的前三个示例：

```
for 条目 in 测试数据[:3]:
    prompt = （
    给定输入 `{format_input(entry)}` ，正确输出 `{entry['output']}` ，对模型响应 `{entry['model_response']}` 进行评分，评分范围从 0 到 100，100 为最佳得分。 ） 打印("\n 数据集响应：") 打印(">>", entry['output']) 打印("\n 模型响应：") 打印(">>", entry["model_response"]) 打印("\n 评分：") 打印(">>", query_model(prompt)) 打印("\n-------------------------")
```

这段代码输出的结果类似于以下内容（截至本文写作时，Ollama 并非完全确定性的，因此生成的文本可能会有所不同）：

数据集响应：

汽车的速度像闪电一样快。

模型响应：

汽车的速度像子弹一样快。

得分：

我给这个模型响应“这辆车快得像子弹一样。”打 85 分。

这是因为：

该回应正确地使用了比喻，将汽车的速度与其他事物（在这种情况下，是一颗子弹）进行比较。
比较是相关且合理的，因为子弹以其高速而闻名。

该短语“像……一样快”正确地用于引入明喻。

我不会给它打满分的原因是有些人可能觉得这种比较稍微不够生动或富有感染力，与其他人相比。例如，将某物比作闪电（如原始回复中所述）可能更具戏剧性和吸引力。然而，“像子弹一样快”仍然是一个强大而有效的比喻，有效地传达了汽车速度的概念。

总体来说，我认为这个模型做得很好！

Dataset response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Score:

>> I'd score this model response as 40 out of 100.

Here's why:

The model correctly identifies that thunderstorms are related to clouds (correctly identifying the type of phenomenon).

However, it incorrectly specifies the type of cloud associated with thunderstorms. Cumulus clouds are not typically associated with thunderstorms; cumulonimbus clouds are.

The response lacks precision and accuracy in its description.

Overall, while the model attempts to address the instruction, it provides an incorrect answer, which is a significant error.

Dataset response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

Score:

>> I'd rate my own response as 95 out of 100. Here's why:

The response accurately answers the question by naming the author of *Pride and Prejudice* as Jane Austen.

The response is concise and clear, making it easy to understand.

There are no grammatical errors or ambiguities that could lead to confusion.

The only reason I wouldn't give myself a perfect score is that the response is slightly redundant—it's not necessary to rephrase the question in the answer. A more concise response would be simply "Jane Austen."

数据集响应：

雷暴通常与积雨云相关。

模型响应：

雷暴相关的云是积云。

得分：

我给这个模型响应打 40 分，满分 100 分。

这是因为：

该模型正确识别出雷暴与云有关（正确识别现象类型）。
然而，它错误地指定了与雷暴相关的云的类型。积云通常不与雷暴相关；积雨云才是。

响应在描述中缺乏精确性和准确性。

总体而言，虽然该模型试图解决指令，但它提供了一个错误的答案，这是一个重大错误。

数据集响应：

简·奥斯汀

模型响应：

《傲慢与偏见》的作者是简·奥斯汀。

得分：

我的回答我给打 95 分，原因如下：

该回复准确地回答了问题，通过指出《傲慢与偏见》的作者为简·奥斯汀。

响应简洁明了，易于理解。

没有语法错误或歧义可能导致混淆。

我不会给自己打满分的原因是回答稍微有些冗余——在回答中不必重新措辞问题。更简洁的回答就是“简·奥斯汀”。

The generated responses show that the Llama 3 model provides reasonable evaluations and is capable of assigning partial points when a model's answer is not entirely correct. For instance, if we consider the evaluation of the "cumulus cloud" answer, the model acknowledges the partial correctness of the response.

The previous prompt returns highly detailed evaluations in addition to the score. We can modify the prompt to just generate integer scores ranging from 0 to 100, where 100 represents the best possible score. This modification allows us to calculate an average score for our model, which serves as a more concise and quantitative assessment of its performance. The `generate_model_scores` function shown in the following listing uses a modified prompt telling the model to `"Respond with the integer number only."`

Listing 7.11 Evaluating the instruction fine-tuning LLM

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `{format_input(entry)}` "
            f"and correct output `{entry['output']}`, "
            f"score the model response `{entry[json_key]}`"
            f" on a scale from 0 to 100, where 100 is the best score. "
            f"Respond with the integer number only."
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Modified instruction line to only return the score

Let's now apply the `generate_model_scores` function to the entire `test_data` set, which takes about 1 minute on a M3 Macbook Air:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

The results are as follows:

```
Scoring entries: 100%|████████████████████████| 110/110
[01:10<00:00,  1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

The evaluation output shows that our fine-tuned model achieves an average score above 50, which provides a useful benchmark for comparison against other models

生成的回复显示，Llama 3 模型提供了合理的评估，并且能够在模型答案不完全正确时分配部分分数。例如，如果我们考虑对“积云”答案的评估，模型承认了响应的部分正确性。

上一条提示不仅返回分数，还返回高度详细的评估。

我们可以修改提示，使其仅生成 0 到 100 之间的整数分数，其中 100 代表最佳可能分数。这种修改使我们能够计算模型的平均分数，这作为对其性能的更简洁和量化的评估。以下列表中所示的 generate_model_scores 函数使用修改后的提示，告诉模型“以分数形式回复”。

仅限整数。

列表 7.11 评估指令微调 LLM

```
def 生成模型得分(json_data, json_key, 模型="llama3")
    scores = []
    for entry 在 tqdm(json_data, desc="评分条目"):
        prompt = (
        给定输入 `{format_input(entry)}` ，以及正确的输出 `{entry['output']}` ，
     对模型的响应 `{entry[json_key]}` 在 0 到 100 的范围内进行评分，其中 100 是最
     佳分数。仅以整数形式回复。 ) 评分 = 查询模型(prompt, model) try:
        scores.append(int(score))
    except ValueError: 翻译为：
    scores.append(int(score)) 除以
    ValueError:
        无法转换分数：{分数} 继续

    返回     分数
```

修改后指令行仅返回分数

现在将 generate_model_scores 函数应用于整个 test_data 数据集，这个过程在 M3 Macbook Air 上大约需要 1 分钟：

```
scores = generate_model_scores(test_data, "model_response") 打印(f"分
数数量: {len(scores)}, 共 {len(test_data)}") 打印(f"平均分数:
{sum(scores)/len(scores):.2f}\n")
```

以下为结果：

```
评分条目：100%|
██████████████████████████████████████| 110/110
[01:10<00:00,  1.56it/s] 评分数量：110/110 平均分：50.32
```

评估输出显示，我们的微调模型平均得分超过 50，这为与其他模型进行比较提供了一个有用的基准

or for experimenting with different training configurations to improve the model's performance.

It's worth noting that Ollama is not entirely deterministic across operating systems at the time of this writing, which means that the scores you obtain might vary slightly from the previous scores. To obtain more robust results, you can repeat the evaluation multiple times and average the resulting scores.

To further improve our model's performance, we can explore various strategies, such as

- Adjusting the hyperparameters during fine-tuning, such as the learning rate, batch size, or number of epochs
- Increasing the size of the training dataset or diversifying the examples to cover a broader range of topics and styles
- Experimenting with different prompts or instruction formats to guide the model's responses more effectively
- Using a larger pretrained model, which may have greater capacity to capture complex patterns and generate more accurate responses

NOTE For reference, when using the methodology described herein, the Llama 3 8B base model, without any fine-tuning, achieves an average score of 58.51 on the test set. The Llama 3 8B instruct model, which has been fine-tuned on a general instruction-following dataset, achieves an impressive average score of 82.6.

Exercise 7.4 Parameter-efficient fine-tuning with LoRA

To instruction fine-tune an LLM more efficiently, modify the code in this chapter to use the low-rank adaptation method (LoRA) from appendix E. Compare the training run time and model performance before and after the modification.

7.9 Conclusions

This chapter marks the conclusion of our journey through the LLM development cycle. We have covered all the essential steps, including implementing an LLM architecture, pretraining an LLM, and fine-tuning it for specific tasks, as summarized in figure 7.21. Let's discuss some ideas for what to look into next.

7.9.1 What's next?

While we covered the most essential steps, there is an optional step that can be performed after instruction fine-tuning: preference fine-tuning. Preference fine-tuning is particularly useful for customizing a model to better align with specific user preferences. If you are interested in exploring this further, see the `04_preference-tuning-with-dpo` folder in this book's supplementary GitHub repository at https://mng.bz/dZwD.

该文本提供了一个有用的基准，用于与其他模型进行比较，或用于尝试不同的训练配置以改进模型性能。

值得注意的是，在撰写本文时，Ollama 在各个操作系统上并非完全确定，这意味着您获得的分数可能与之前的分数略有不同。为了获得更稳健的结果，您可以多次重复评估并平均最终分数。

为进一步提升我们模型的表现，我们可以探索各种策略，例如

- 调整微调过程中的超参数，例如学习率、批量大小或训练轮数
- 增加训练数据集的大小或多样化示例以涵盖更广泛的主题和风格
- 尝试使用不同的提示或指令格式来更有效地引导模型的响应
- 使用更大的预训练模型，可能具有更大的能力来捕捉复杂模式和生成更准确的响应

注意：本方法所述方法中，未经微调的 Llama 3 8B 基础模型在测试集上平均得分为 58.51。经过在通用指令遵循数据集上微调的 Llama 3 8B 指令模型，实现了令人印象深刻的平均得分 82.6。

练习 7.4 使用 LoRA 进行参数高效的微调

为了更有效地微调LLM，修改本章中的代码以使用附录 E 中的低秩自适应方法（LoRA）。比较修改前后的训练运行时间和模型性能。

7.9 结论

本章标志着我们通过LLM开发周期的旅程的结束。我们涵盖了所有基本步骤，包括实现LLM架构、预训练LLM以及针对特定任务进行微调，如图 7.21 所示。让我们讨论一下接下来要关注的一些想法。

7.9.1 下一步是什么?

虽然我们涵盖了最基本步骤，但还有一个可选步骤可以在指令微调后执行：偏好微调。偏好微调特别有助于定制模型以更好地符合特定用户偏好。如果您想进一步探索，请参阅本书补充 GitHub 仓库中的 04_preference-tuningwith-dpo 文件夹，网址为 https://mng.bz/dZwD。

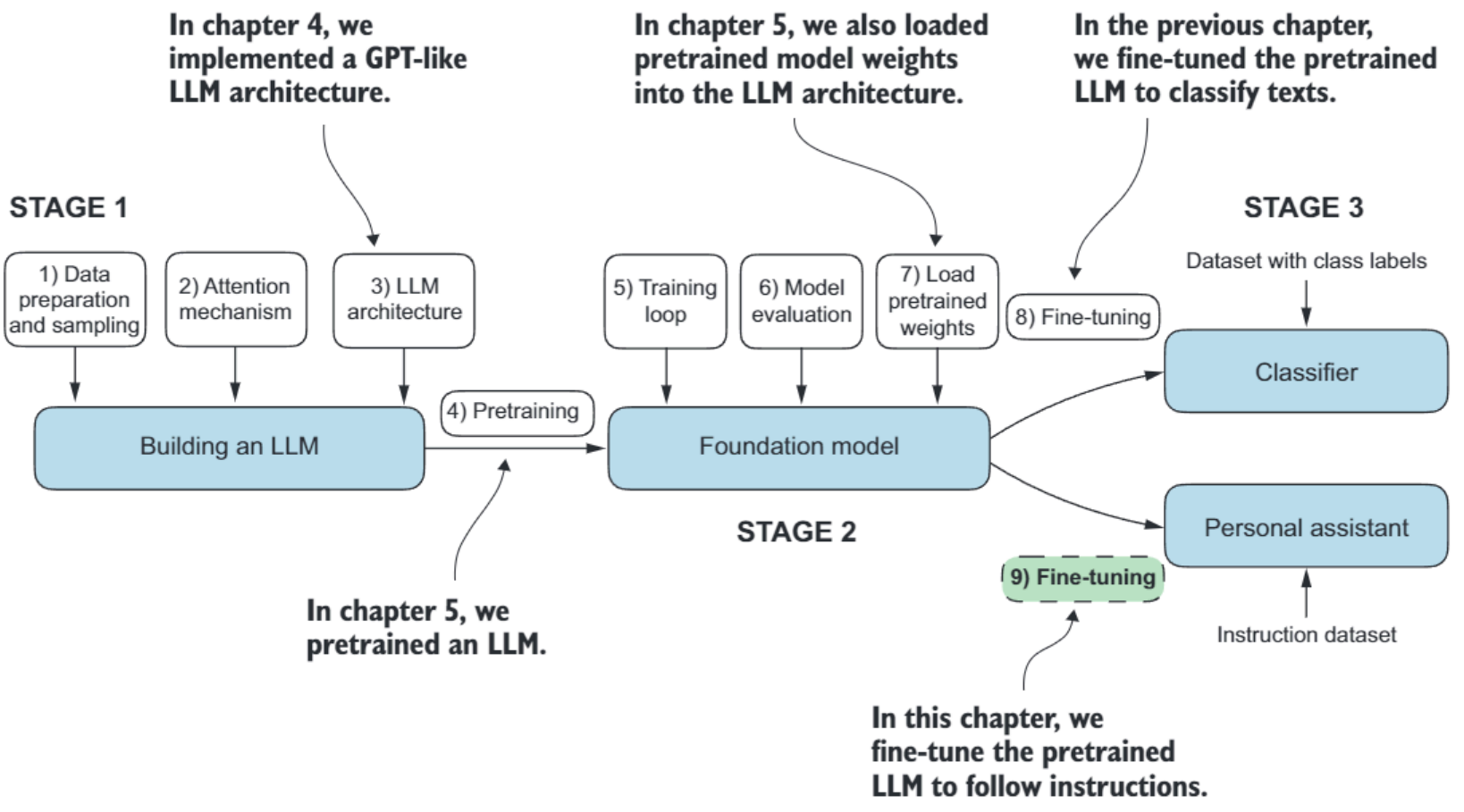

Figure 7.21 The three main stages of coding an LLM.

In addition to the main content covered in this book, the GitHub repository also contains a large selection of bonus material that you may find valuable. To learn more about these additional resources, visit the Bonus Material section on the repository's README page: https://mng.bz/r12g.

7.9.2 Staying up to date in a fast-moving field

The fields of AI and LLM research are evolving at a rapid (and, depending on who you ask, exciting) pace. One way to keep up with the latest advancements is to explore recent research papers on arXiv at https://arxiv.org/list/cs.LG/recent. Additionally, many researchers and practitioners are very active in sharing and discussing the latest developments on social media platforms like X (formerly Twitter) and Reddit. The subreddit r/LocalLLaMA, in particular, is a good resource for connecting with the community and staying informed about the latest tools and trends. I also regularly share insights and write about the latest in LLM research on my blog, available at https://magazine.sebastianraschka.com and https://sebastianraschka.com/blog/.

7.9.3 Final words

I hope you have enjoyed this journey of implementing an LLM from the ground up and coding the pretraining and fine-tuning functions from scratch. In my opinion, building an LLM from scratch is the most effective way to gain a deep understanding of how LLMs work. I hope that this hands-on approach has provided you with valuable insights and a solid foundation in LLM development.

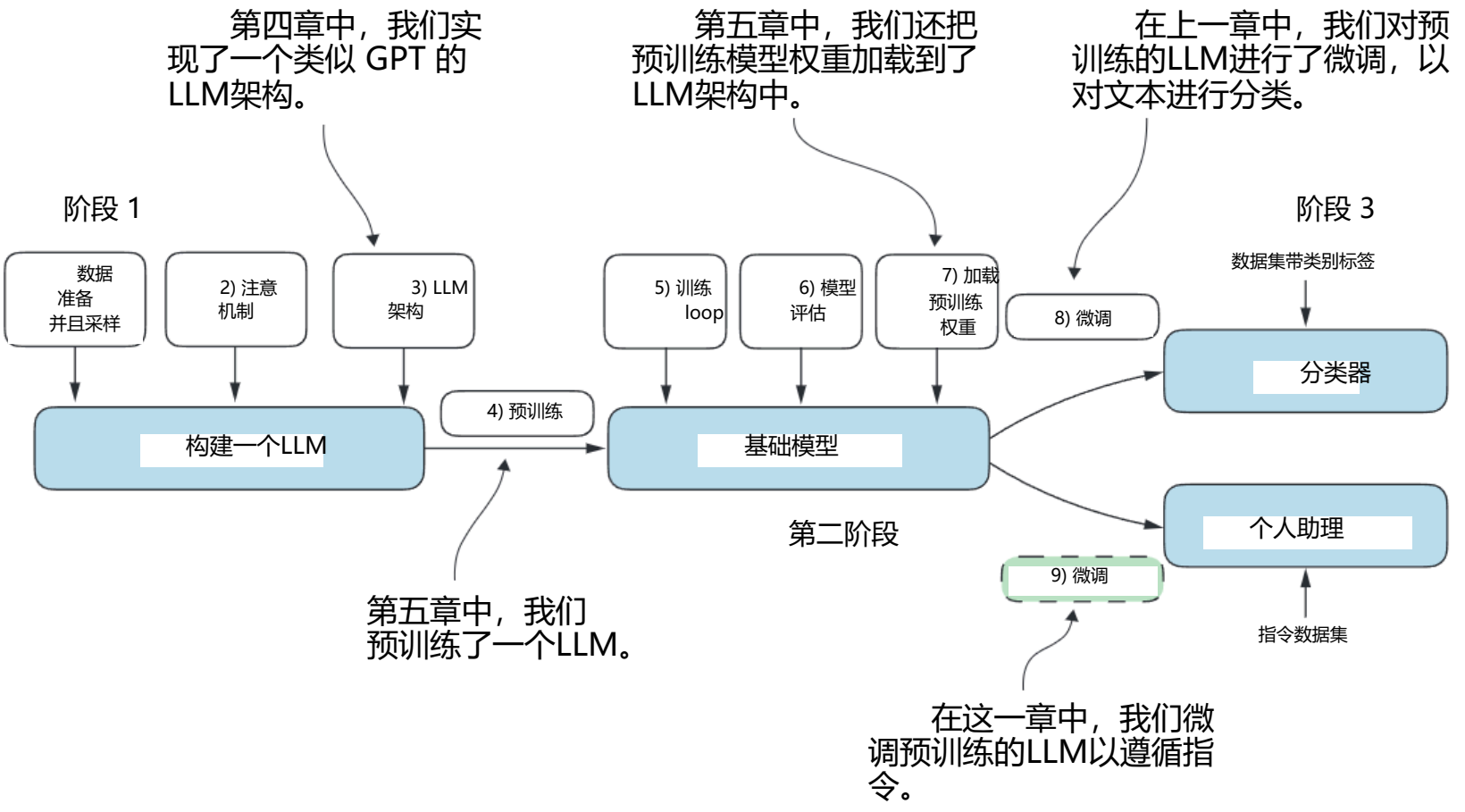

图 7.21 编码的三个主要阶段为：LLM。

除了本书涵盖的主要内容外，GitHub 仓库还包含大量您可能认为有价值的额外材料。要了解更多关于这些额外资源的信息，请访问仓库的 README 页面上的“额外材料”部分：https://mng.bz/r12g。

7.9.2 保持对快速发展的领域的最新了解

人工智能和LLM研究领域正在迅速（并且，取决于你问谁，令人兴奋）地发展。跟上最新进展的一种方式是探索 arXiv 上的最新研究论文，网址为 https://arxiv.org/list/cs.LG/recent。此外，许多研究人员和从业者都在社交媒体平台如 X（前身为 Twitter）和 Reddit 上非常活跃地分享和讨论最新进展。特别是 r/LocalLLaMA 这个 subreddit 是一个很好的资源，可以与社区建立联系并了解最新的工具和趋势。我还在我的博客上定期分享见解并撰写关于LLM研究的最新动态，博客可在 https://magazine.sebastianraschka.com 和 https://sebastianraschka.com/blog/上找到。

7.9.3 最后的话

我希望您已经享受了从零开始实现一个LLM的旅程，并从头开始编写预训练和微调函数。在我看来，从头开始构建一个LLM是最有效的方法来深入理解LLMs的工作原理。我希望这种动手方法为您提供了宝贵的见解和在LLM开发方面的坚实基础。

While the primary purpose of this book is educational, you may be interested in utilizing different and more powerful LLMs for real-world applications. For this, I recommend exploring popular tools such as Axolotl (https://github.com/OpenAccess-AI-Collective/axolotl) or LitGPT (https://github.com/Lightning-AI/litgpt), which I am actively involved in developing.

Thank you for joining me on this learning journey, and I wish you all the best in your future endeavors in the exciting field of LLMs and AI!

Summary

- The instruction-fine-tuning process adapts a pretrained LLM to follow human instructions and generate desired responses.
- Preparing the dataset involves downloading an instruction-response dataset, formatting the entries, and splitting it into train, validation, and test sets.
- Training batches are constructed using a custom collate function that pads sequences, creates target token IDs, and masks padding tokens.
- We load a pretrained GPT-2 medium model with 355 million parameters to serve as the starting point for instruction fine-tuning.
- The pretrained model is fine-tuned on the instruction dataset using a training loop similar to pretraining.
- Evaluation involves extracting model responses on a test set and scoring them (for example, using another LLM).
- The Ollama application with an 8-billion-parameter Llama model can be used to automatically score the fine-tuned model's responses on the test set, providing an average score to quantify performance.

尽管这本书的主要目的是教育性的，您可能会对利用不同且更强大的LLMs进行实际应用感兴趣。为此，我建议探索流行的工具，如 Axolotl（https://github.com/OpenAccess-AI-Collective/axolotl）或 LitGPT（https://github.com/Lightning-AI/litgpt），我正在积极参与其开发。

感谢您加入我的学习之旅，并祝您在激动人心的LLMs和 AI 领域未来的努力一切顺利！

摘要

- 指令微调过程将预训练的LLM适配以遵循人类指令并生成期望的响应。
- 准备数据集涉及下载指令-响应数据集、格式化条目并将其分为训练集、验证集和测试集。
- 训练批次使用自定义的 collate 函数构建，该函数填充序列，创建目标标记 ID，并掩码填充标记。
- 我们加载一个预训练的 GPT-2 中型模型，该模型包含 355 百万个参数，作为指令微调的起点。
- 预训练模型在指令数据集上使用类似于预训练的训练循环进行微调。
- 评估涉及从测试集中提取模型响应并对它们进行评分（例如，使用另一个LLM）。
- Ollama 应用程序使用 8 亿参数的 Llama 模型，可以自动评分测试集上微调模型的响应，提供平均分以量化性能。