

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.

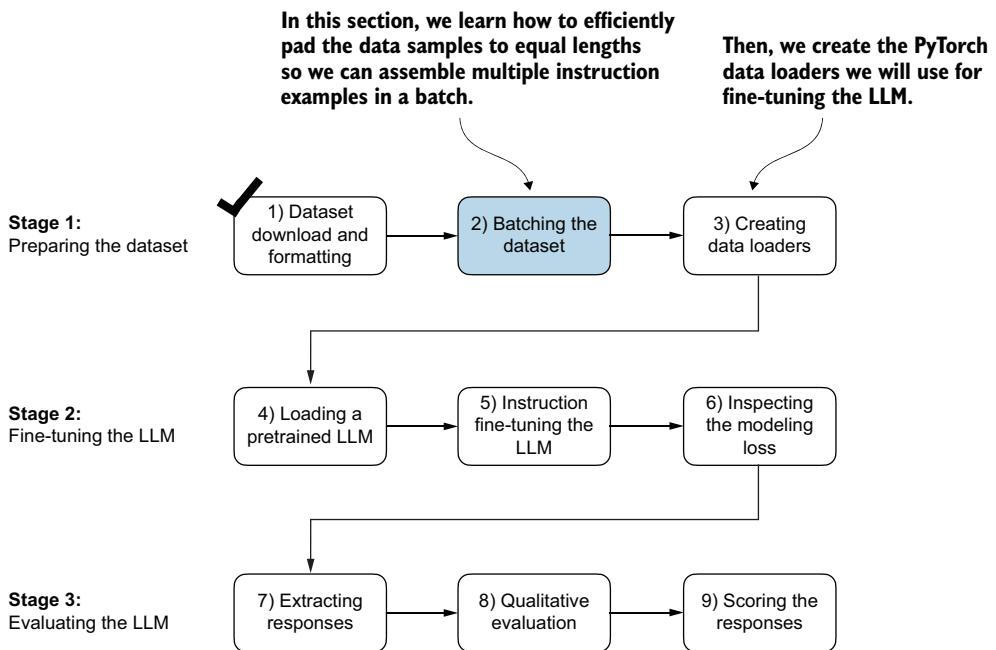


Figure 7.5 The three-stage process for instruction fine-tuning an LLM. Next, we look at step 2 of stage 1: assembling the training batches.

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

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`.

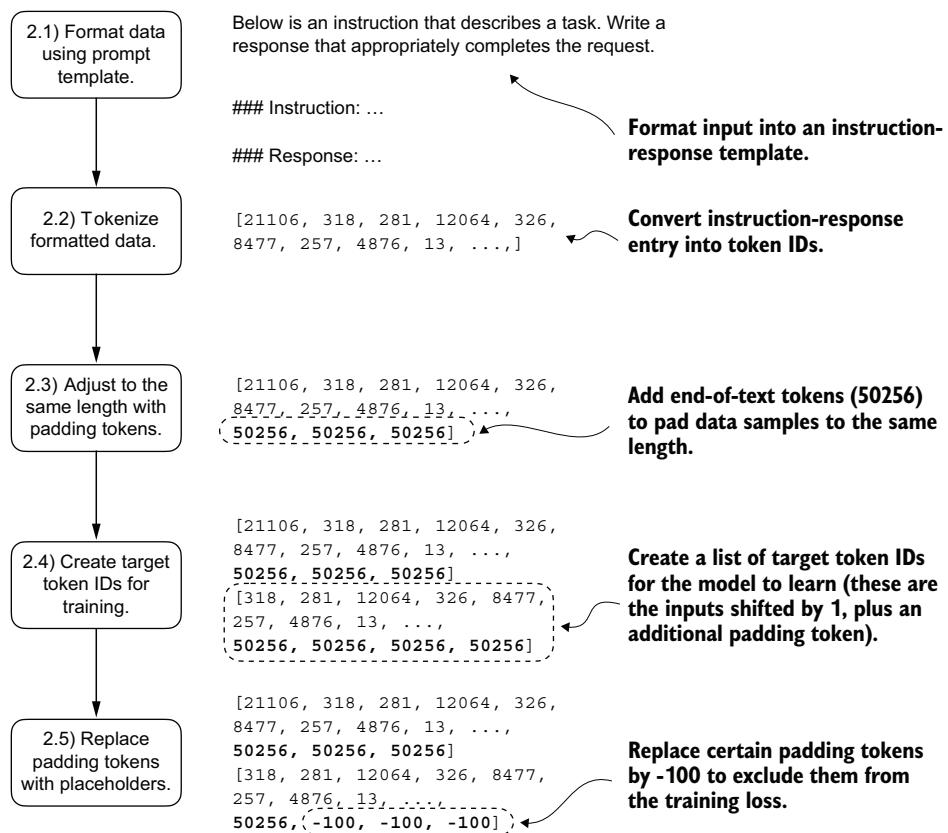


Figure 7.6 The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing -100 placeholder tokens to mask padding tokens in the loss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

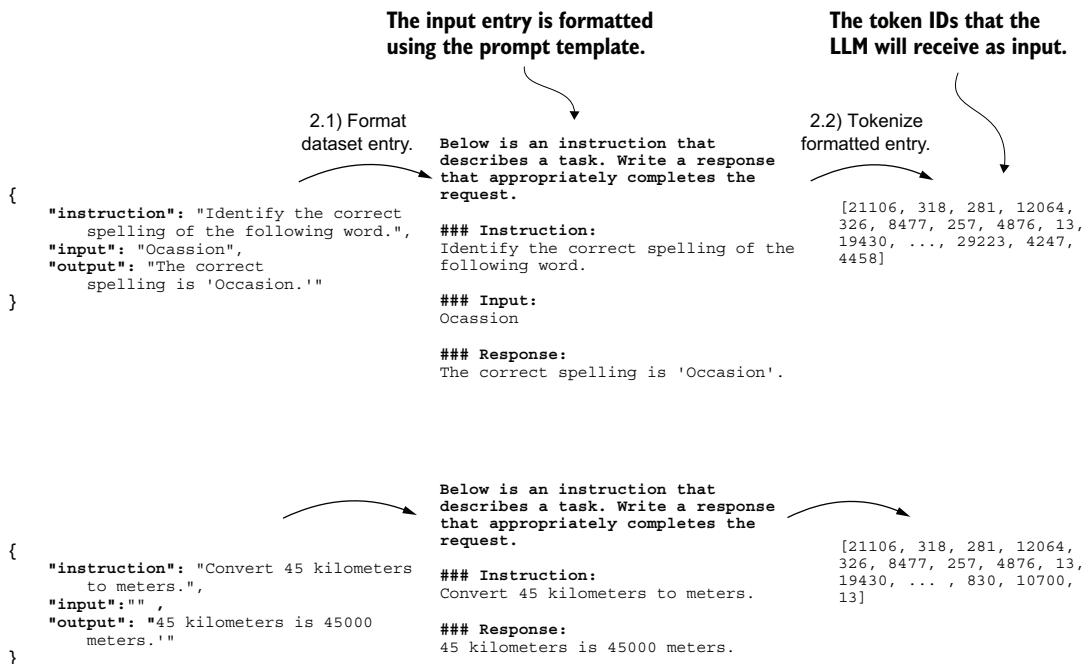


Figure 7.7 The first two steps involved in implementing the batching process. Entries are first formatted using a specific prompt template (2.1) and then tokenized (2.2), resulting in a sequence of token IDs that the model can process.

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:
            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)
```

↑ Pretokenizes texts