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

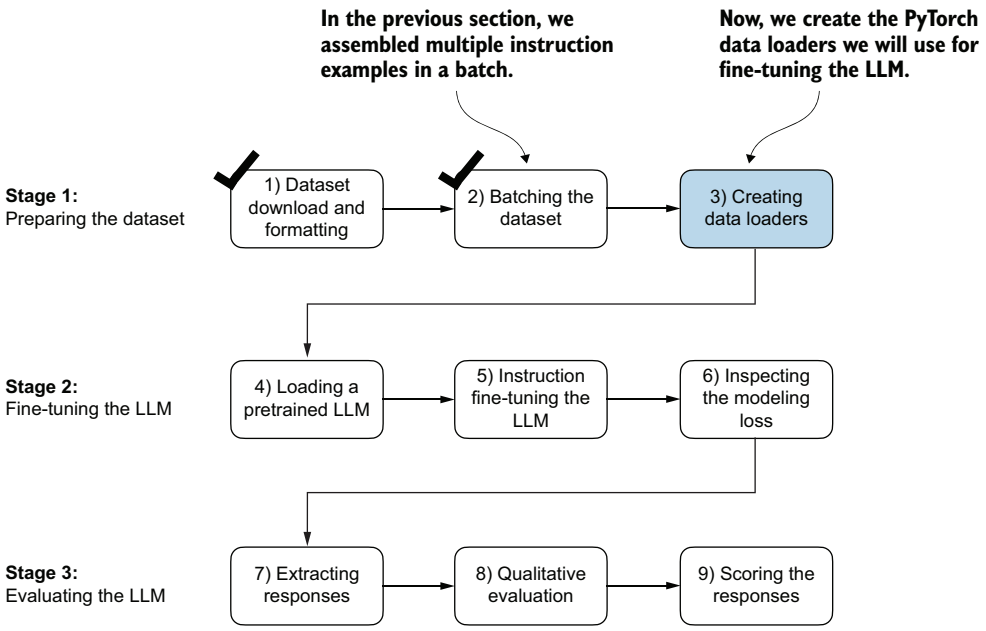

Figure 7.14 The three-stage process for instruction fine-tuning an LLM. Thus far, we have prepared the dataset and implemented a custom collate function to batch the instruction dataset. Now, we can create and apply the data loaders to the training, validation, and test sets needed for the LLM instruction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

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.

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])
...
```