```
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

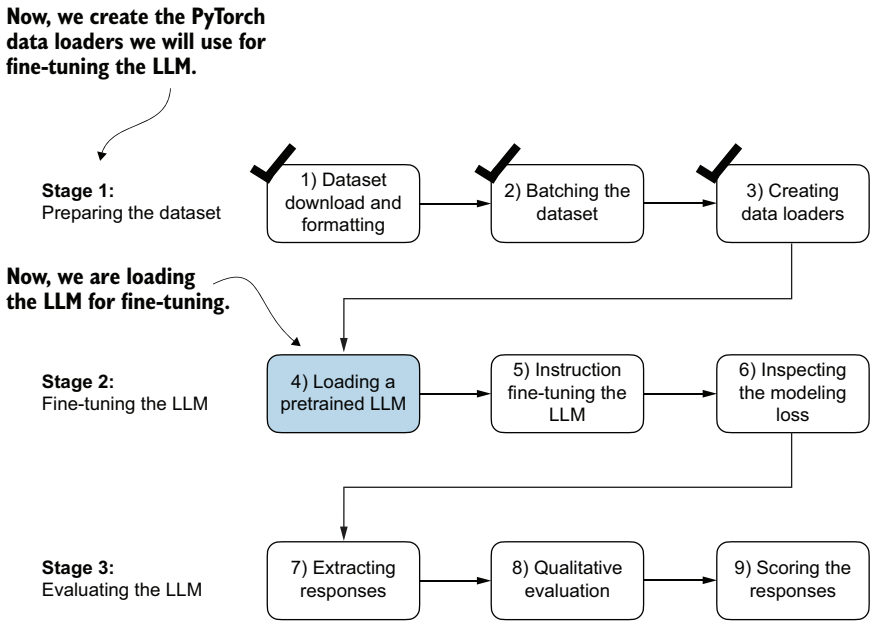

Figure 7.15 The three-stage process for instruction fine-tuning an LLM. After the dataset preparation, the process of fine-tuning an LLM for instruction-following begins with loading a pretrained LLM, which serves as the foundation for subsequent training.

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
```

```
[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)
```