If you encounter out-of-memory errors, consider reducing the batch size from 8 to 4, 2, or 1. In addition to lowering the batch size, you may also want to consider lowering the `allowed_max_length` from 1024 to 512 or 256.

Below are a few examples from the Alpaca dataset, including the generated model responses:

Exercise 7.4

To instruction fine-tune the model using LoRA, use the relevant classes and functions from appendix E:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Next, add the following lines of code below the model loading code in section 7.5:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

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

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
model.to(device)
```

Note that, on an Nvidia L4 GPU, the fine-tuning with LoRA takes 1.30 min to run on an L4. On the same GPU, the original code takes 1.80 minutes to run. So, LoRA is approximately 28% faster in this case. The score, evaluated with the Ollama Llama 3 method from chapter 7, is around 50, which is in the same ballpark as the original model.

Appendix A

Exercise A.1

The optional Python Setup Tips document (https://github.com/rasbt/LLMs-from-scratch/tree/main/setup/01_optional-python-setup-preferences) contains additional recommendations and tips if you need additional help to set up your Python environment.

Licensed to Jason Walsh <manning@jwalsh.net>

Exercise A.2

The the optional "Installing Libraries Used In This Book" document (https://github.com/rasbt/LLMs-from-scratch/tree/main/setup/02_installing-python-libraries) contains utilities to check whether your environment is set up correctly.

Exercise A.3

The network has two inputs and two outputs. In addition, there are 2 hidden layers with 30 and 20 nodes, respectively. Programmatically, we can calculate the number of parameters as follows:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

This returns

```
752
```

We can also calculate this manually as follows:

- *First hidden layer:* 2 inputs times 30 hidden units plus 30 bias units
- *Second hidden layer:* 30 incoming units times 20 nodes plus 20 bias units
- *Output layer:* 20 incoming nodes times 2 output nodes plus 2 bias units

Then, adding all the parameters in each layer results in 2×30+30 + 30×20+20 + 20×2+2 = 752.

Exercise A.4

The exact run-time results will be specific to the hardware used for this experiment. In my experiments, I observed significant speed-ups even for small matrix multiplications when using a Google Colab instance connected to a V100 GPU:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

On the CPU this resulted in

```
63.8 µs ± 8.7 µs per loop
```

When executed on a GPU

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

Licensed to Jason Walsh <manning@jwalsh.net>

The result was

```
13.8 µs ± 425 ns per loop
```

In this case, on a V100, the computation was approximately four times faster.

Licensed to Jason Walsh <manning@jwalsh.net>