Exercise 6.3

Rather than fine-tuning the last output token, we can fine-tune the first output token by changing `model(input_batch)[:, -1, :]` to `model(input_batch)[:, 0, :]` everywhere in the code.

As expected, since the first token contains less information than the last token, this change results in a substantially worse test accuracy of 75.00% (vs. the 95.67% in the main chapter).

Chapter 7

Exercise 7.1

The Phi-3 prompt format, which is shown in figure 7.4, looks like the following for a given example input:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

To use this template, we can modify the `format_input` function as follows:

```
def format_input(entry):
    instruction_text = (
        f"<|user|>\n{entry['instruction']}"
    )
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Lastly, we also have to update the way we extract the generated response when we collect the test set responses:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    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("<|assistant|>:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text
```

New: Adjust ###Response to <|assistant|>

Fine-tuning the model with the Phi-3 template is approximately 17% faster since it results in shorter model inputs. The score is close to 50, which is in the same ballpark as the score we previously achieved with the Alpaca-style prompts.

Exercise 7.2

To mask out the instructions as shown in figure 7.13, we need to make slight modifications to the `InstructionDataset` class and `custom_collate_fn` function. We can modify the `InstructionDataset` class to collect the lengths of the instructions, which we will use in the collate function to locate the instruction content positions in the targets when we code the collate function, as follows:

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []
        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)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input)
            )
            self.instruction_lengths.append(instruction_length)

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

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

Separate list for instruction lengths

Collects instruction lengths

Returns both instruction lengths and texts separately

Next, we update the `custom_collate_fn` where each `batch` is now a tuple containing `(instruction_length, item)` instead of just `item` due to the changes in the `InstructionDataset` dataset. In addition, we now mask the corresponding instruction tokens in the target ID list:

```
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 instruction_length, item in batch)
    inputs_lst, targets_lst = [], []

    for instruction_length, 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

        targets[:instruction_length-1] = -100

        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
```

batch is now a tuple.

Masks all input and instruction tokens in the targets

When evaluating a model fine-tuned with this instruction masking method, it performs slightly worse (approximately 4 points using the Ollama Llama 3 method from chapter 7). This is consistent with observations in the "Instruction Tuning With Loss Over Instructions" paper (https://arxiv.org/abs/2405.14394).

Exercise 7.3

To fine-tune the model on the original Stanford Alpaca dataset (https://github.com/tatsu-lab/stanford_alpaca), we just have to change the file URL from

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

to

```
url = "https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/
alpaca_data.json"
```

Note that the dataset contains 52,000 entries (50x more than in chapter 7), and the entries are longer than the ones we worked with in chapter 7.

Thus, it's highly recommended that the training be run on a GPU.