As we can see based on the sharp downward slope in figure 6.16, the model is learning well from the training data, and there is little to no indication of overfitting; that is, there is no noticeable gap between the training and validation set losses.

Choosing the number of epochs

Earlier, when we initiated the training, we set the number of epochs to five. The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation, although an epoch number of five is usually a good starting point. If the model overfits after the first few epochs as a loss plot (see figure 6.16), you may need to reduce the number of epochs. Conversely, if the trendline suggests that the validation loss could improve with further training, you should increase the number of epochs. In this concrete case, five epochs is a reasonable number as there are no signs of early overfitting, and the validation loss is close to 0.

Using the same `plot_values` function, let's now plot the classification accuracies:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="accuracy"
)
```

Figure 6.17 graphs the resulting accuracy. The model achieves a relatively high training and validation accuracy after epochs 4 and 5. Importantly, we previously set `eval_iter=5`

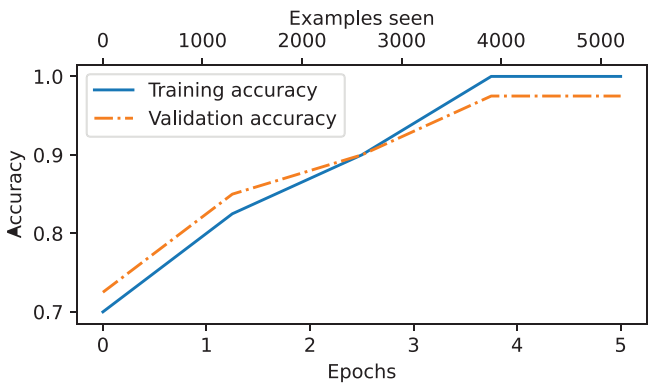


Figure 6.17 Both the training accuracy (solid line) and the validation accuracy (dashed line) increase substantially in the early epochs and then plateau, achieving almost perfect accuracy scores of 1.0. The close proximity of the two lines throughout the epochs suggests that the model does not overfit the training data very much.

when using the `train_classifier_simple` function, which means our estimations of training and validation performance are based on only five batches for efficiency during training.

Now we must calculate the performance metrics for the training, validation, and test sets across the entire dataset by running the following code, this time without defining the `eval_iter` value:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

The training and test set performances are almost identical. The slight discrepancy between the training and test set accuracies suggests minimal overfitting of the training data. Typically, the validation set accuracy is somewhat higher than the test set accuracy because the model development often involves tuning hyperparameters to perform well on the validation set, which might not generalize as effectively to the test set. This situation is common, but the gap could potentially be minimized by adjusting the model's settings, such as increasing the dropout rate (`drop_rate`) or the `weight_decay` parameter in the optimizer configuration.

6.8 Using the LLM as a spam classifier

Having fine-tuned and evaluated the model, we are now ready to classify spam messages (see figure 6.18). Let's use our fine-tuned GPT-based spam classification model. The following `classify_review` function follows data preprocessing steps similar to those we used in the `SpamDataset` implemented earlier. Then, after processing text into token IDs, the function uses the model to predict an integer class label, similar to what we implemented in section 6.6, and then returns the corresponding class name.

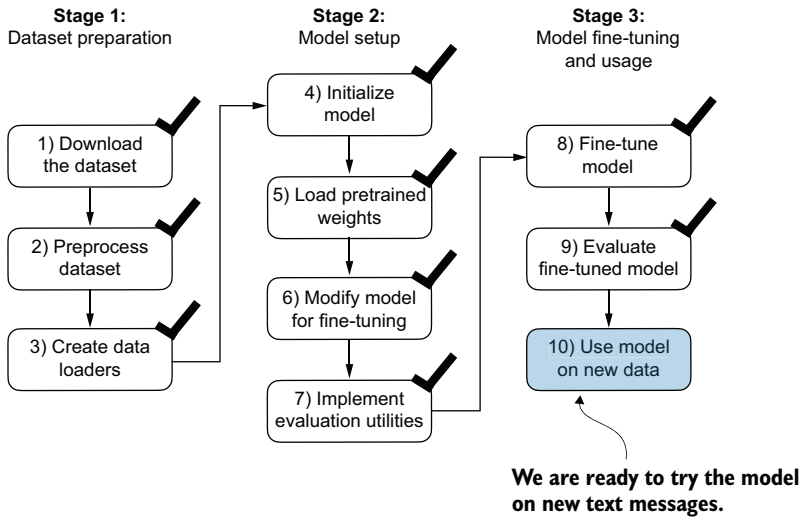


Figure 6.18 The three-stage process for classification fine-tuning our LLM. Step 10 is the final step of stage 3—using the fine-tuned model to classify new spam messages.

Listing 6.12 Using the model to classify new texts

```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[0]

    input_ids = input_ids[:min(
        max_length, supported_context_length
    )]

    input_ids += [pad_token_id] * (max_length - len(input_ids))

    input_tensor = torch.tensor(
        input_ids, device=device
    ).unsqueeze(0)

    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam"
```

Prepares inputs to the model

Truncates sequences if they are too long

Pads sequences to the longest sequence

Adds batch dimension

Models inference without gradient tracking

Logits of the last output token

Returns the classified result

Let's try this `classify_review` function on an example text:

```
text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The resulting model correctly predicts `"spam"`. Let's try another example:

```
text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(
    text_2, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The model again makes a correct prediction and returns a "not spam" label.

Finally, let's save the model in case we want to reuse the model later without having to train it again. We can use the `torch.save` method:

```
torch.save(model.state_dict(), "review_classifier.pth")
```

Once saved, the model can be loaded:

```
model_state_dict = torch.load("review_classifier.pth, map_location=device")
model.load_state_dict(model_state_dict)
```

Summary

- There are different strategies for fine-tuning LLMs, including classification fine-tuning and instruction fine-tuning.
- Classification fine-tuning involves replacing the output layer of an LLM via a small classification layer.
- In the case of classifying text messages as "spam" or "not spam," the new classification layer consists of only two output nodes. Previously, we used the number of output nodes equal to the number of unique tokens in the vocabulary (i.e., 50,256).
- Instead of predicting the next token in the text as in pretraining, classification fine-tuning trains the model to output a correct class label—for example, "spam" or "not spam."
- The model input for fine-tuning is text converted into token IDs, similar to pretraining.

- Before fine-tuning an LLM, we load the pretrained model as a base model.
- Evaluating a classification model involves calculating the classification accuracy (the fraction or percentage of correct predictions).
- Fine-tuning a classification model uses the same cross entropy loss function as when pretraining the LLM.

7 Fine-tuning to follow instructions

This chapter covers

- The instruction fine-tuning process of LLMs
- Preparing a dataset for supervised instruction fine-tuning
- Organizing instruction data in training batches
- Loading a pretrained LLM and fine-tuning it to follow human instructions
- Extracting LLM-generated instruction responses for evaluation
- Evaluating an instruction-fine-tuned LLM

Previously, we implemented the LLM architecture, carried out pretraining, and imported pretrained weights from external sources into our model. Then, we focused on fine-tuning our LLM for a specific classification task: distinguishing between spam and non-spam text messages. Now we'll implement the process for fine-tuning an LLM to follow human instructions, as illustrated in figure 7.1. Instruction fine-tuning is one of the main techniques behind developing LLMs for chatbot applications, personal assistants, and other conversational tasks.

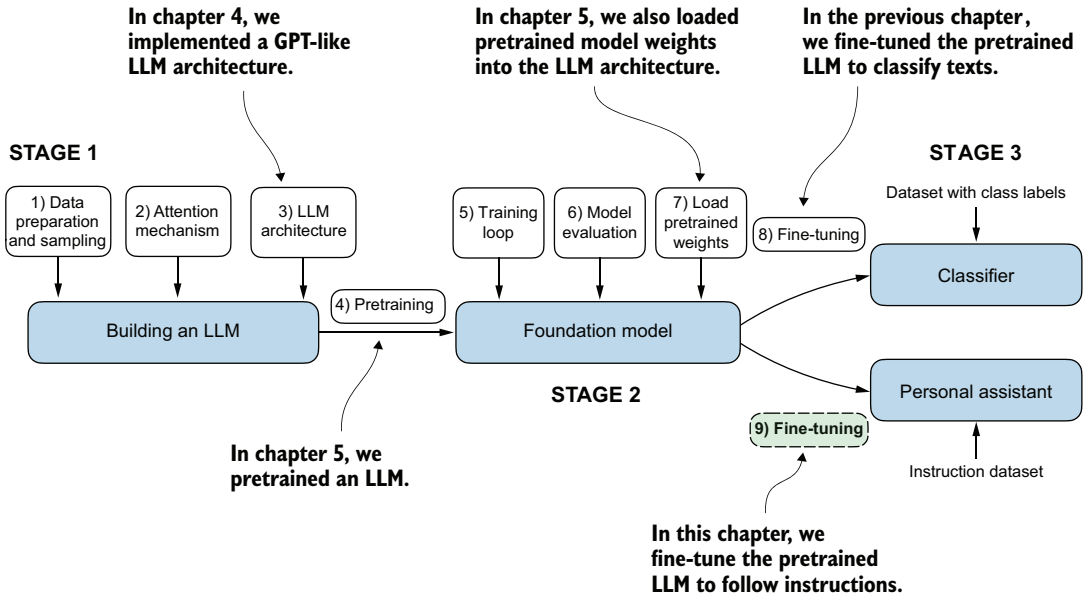


Figure 7.1 The three main stages of coding an LLM. This chapter focuses on step 9 of stage 3: fine-tuning a pretrained LLM to follow human instructions.

Figure 7.1 shows two main ways of fine-tuning an LLM: fine-tuning for classification (step 8) and fine-tuning an LLM to follow instructions (step 9). We implemented step 8 in chapter 6. Now we will fine-tune an LLM using an *instruction dataset*.

7.1 Introduction to instruction fine-tuning

We now know that pretraining an LLM involves a training procedure where it learns to generate one word at a time. The resulting pretrained LLM is capable of *text completion*, meaning it can finish sentences or write text paragraphs given a fragment as input. However, pretrained LLMs often struggle with specific instructions, such as "Fix the grammar in this text" or "Convert this text into passive voice." Later, we will examine a concrete example where we load the pretrained LLM as the basis for *instruction fine-tuning*, also known as *supervised instruction fine-tuning*.

Here, we focus on improving the LLM's ability to follow such instructions and generate a desired response, as illustrated in figure 7.2. Preparing the dataset is a key aspect of instruction fine-tuning. Then we'll complete all the steps in the three stages of the instruction fine-tuning process, beginning with the dataset preparation, as shown in figure 7.3.

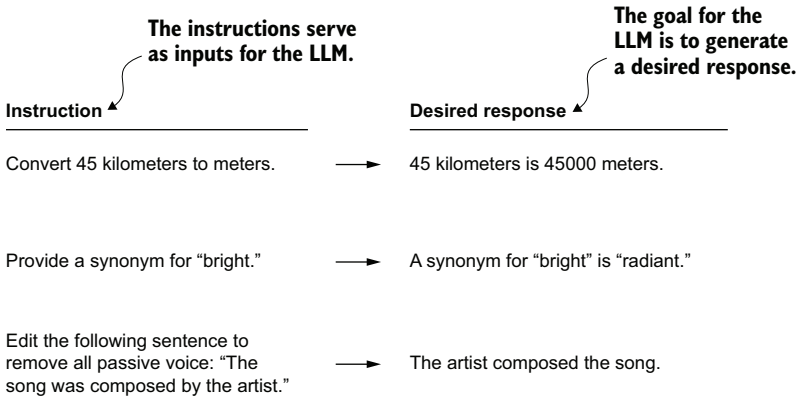


Figure 7.2 Examples of instructions that are processed by an LLM to generate desired responses

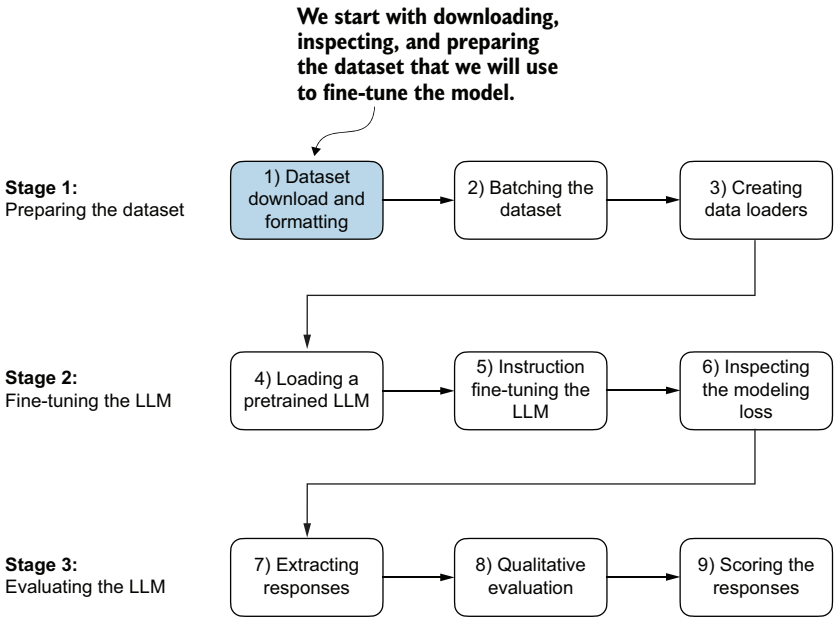


Figure 7.3 The three-stage process for instruction fine-tuning an LLM. Stage 1 involves dataset preparation, stage 2 focuses on model setup and fine-tuning, and stage 3 covers the evaluation of the model. We will begin with step 1 of stage 1: downloading and formatting the dataset.

7.2 Preparing a dataset for supervised instruction fine-tuning

Let's download and format the instruction dataset for instruction fine-tuning a pretrained LLM. The dataset consists of 1,100 *instruction–response pairs* similar to those in figure 7.2. This dataset was created specifically for this book, but interested readers can find alternative, publicly available instruction datasets in appendix B.

The following code implements and executes a function to download this dataset, which is a relatively small file (only 204 KB) in JSON format. JSON, or JavaScript Object Notation, mirrors the structure of Python dictionaries, providing a simple structure for data interchange that is both human readable and machine friendly.

Listing 7.1 Downloading the dataset

```
import json
import os
import urllib

def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

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


data = download_and_load_file(file_path, url)
print("Number of entries:", len(data))
```

The output of executing the preceding code is

```
Number of entries: 1100
```

The `data` list that we loaded from the JSON file contains the 1,100 entries of the instruction dataset. Let's print one of the entries to see how each entry is structured:

```
print("Example entry:\n", data[50])
```

The content of the example entry is

```
Example entry:
 {'instruction': 'Identify the correct spelling of the following word.',
  'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}
```

As we can see, the example entries are Python dictionary objects containing an `'instruction'`, `'input'`, and `'output'`. Let's take a look at another example:

```
print("Another example entry:\n", data[999])
```

Based on the contents of this entry, the `'input'` field may occasionally be empty:

```
Another example entry:
 {'instruction': "What is an antonym of 'complicated'?",
  'input': '',
  'output': "An antonym of 'complicated' is 'simple'."}
```

Instruction fine-tuning involves training a model on a dataset where the input-output pairs, like those we extracted from the JSON file, are explicitly provided. There are various methods to format these entries for LLMs. Figure 7.4 illustrates two different

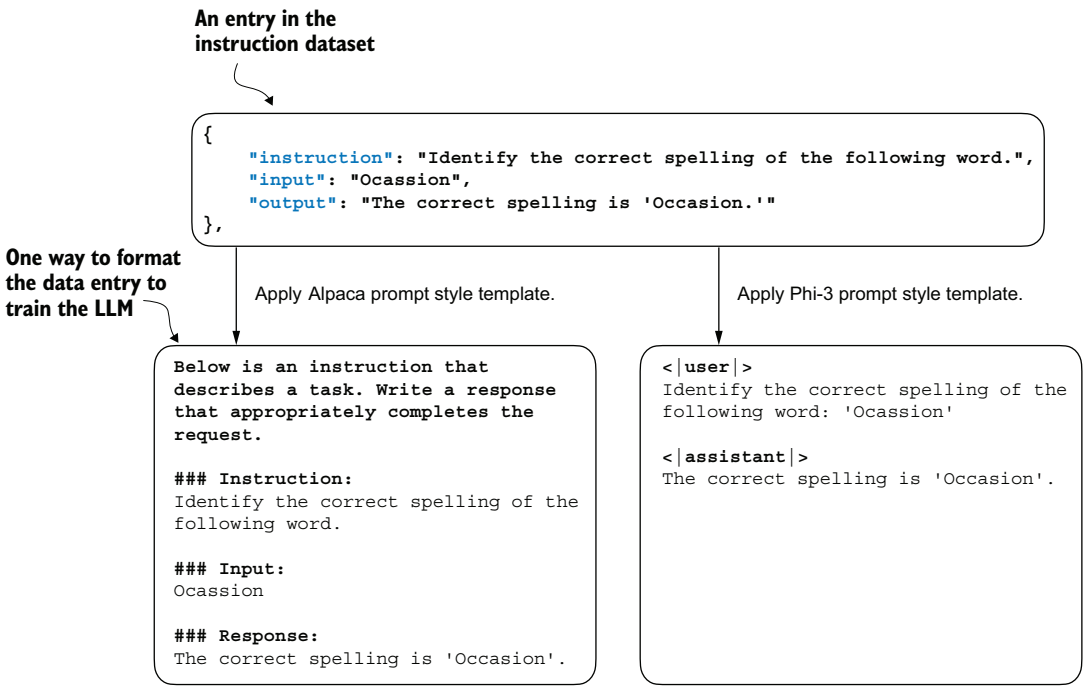


Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

example formats, often referred to as *prompt styles*, used in the training of notable LLMs such as Alpaca and Phi-3.

Alpaca was one of the early LLMs to publicly detail its instruction fine-tuning process. Phi-3, developed by Microsoft, is included to demonstrate the diversity in prompt styles. The rest of this chapter uses the Alpaca prompt style since it is one of the most popular ones, largely because it helped define the original approach to fine-tuning.

Exercise 7.1 Changing prompt styles

After fine-tuning the model with the Alpaca prompt style, try the Phi-3 prompt style shown in figure 7.4 and observe whether it affects the response quality of the model.

Let's define a `format_input` function that we can use to convert the entries in the `data` list into the Alpaca-style input format.

Listing 7.2 Implementing the prompt formatting function

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

This `format_input` function takes a dictionary `entry` as input and constructs a formatted string. Let's test it to dataset entry `data[50]`, which we looked at earlier:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

The formatted input looks like as follows:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
Identify the correct spelling of the following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion.'
```

Note that the `format_input` skips the optional `### Input:` section if the `'input'` field is empty, which we can test out by applying the `format_input` function to entry `data[999]` that we inspected earlier:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

The output shows that entries with an empty `'input'` field don't contain an `### Input:` section in the formatted input:

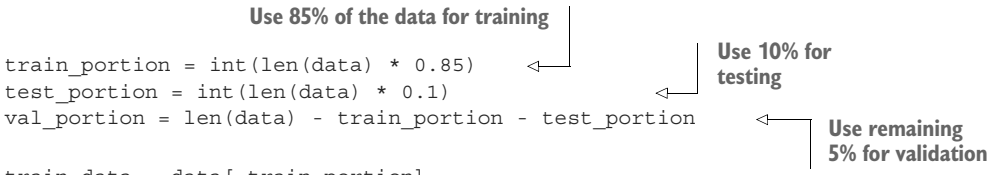
```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
What is an antonym of 'complicated'?

### Response:
An antonym of 'complicated' is 'simple'.
```

Before we move on to setting up the PyTorch data loaders in the next section, let's divide the dataset into training, validation, and test sets analogous to what we have done with the spam classification dataset in the previous chapter. The following listing shows how we calculate the portions.

Listing 7.3 Partitioning the dataset




```
train_portion = int(len(data) * 0.85)
test_portion = int(len(data) * 0.1)
val_portion = len(data) - train_portion - test_portion

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

This partitioning results in the following dataset sizes:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```

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.

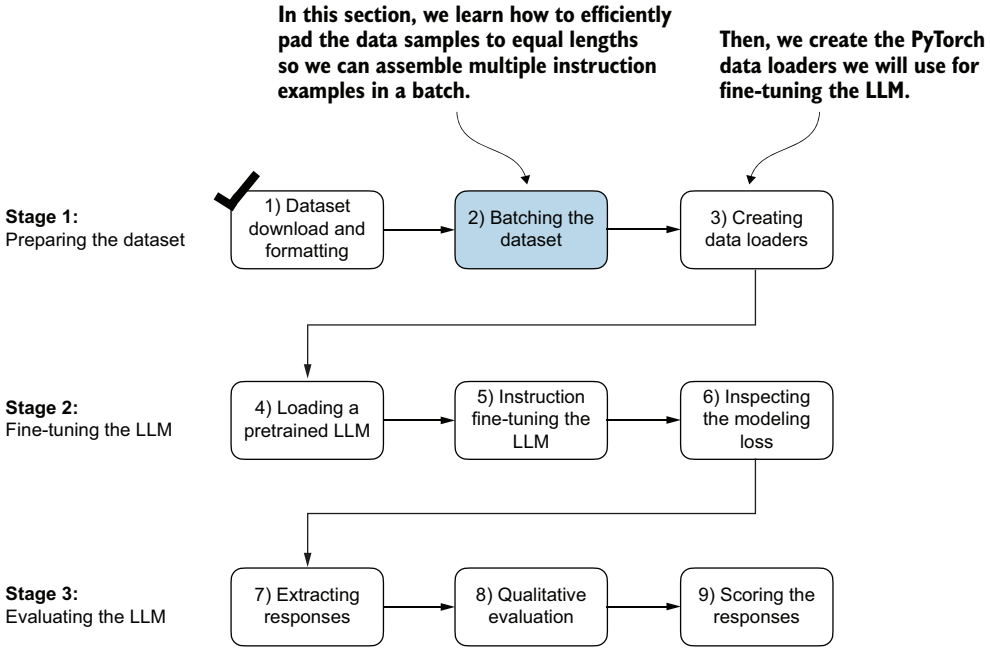


Figure 7.5 The three-stage process for instruction fine-tuning an LLM. Next, we look at step 2 of stage 1: assembling the training batches.

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`.

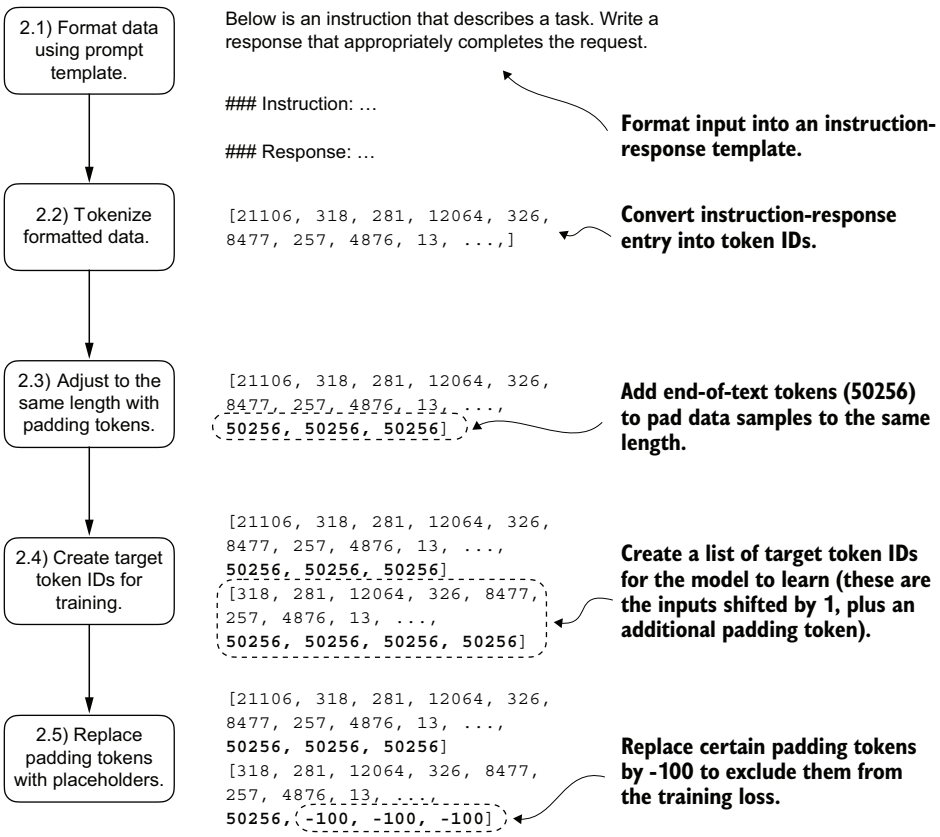


Figure 7.6 The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing `-100` placeholder tokens to mask padding tokens in the loss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

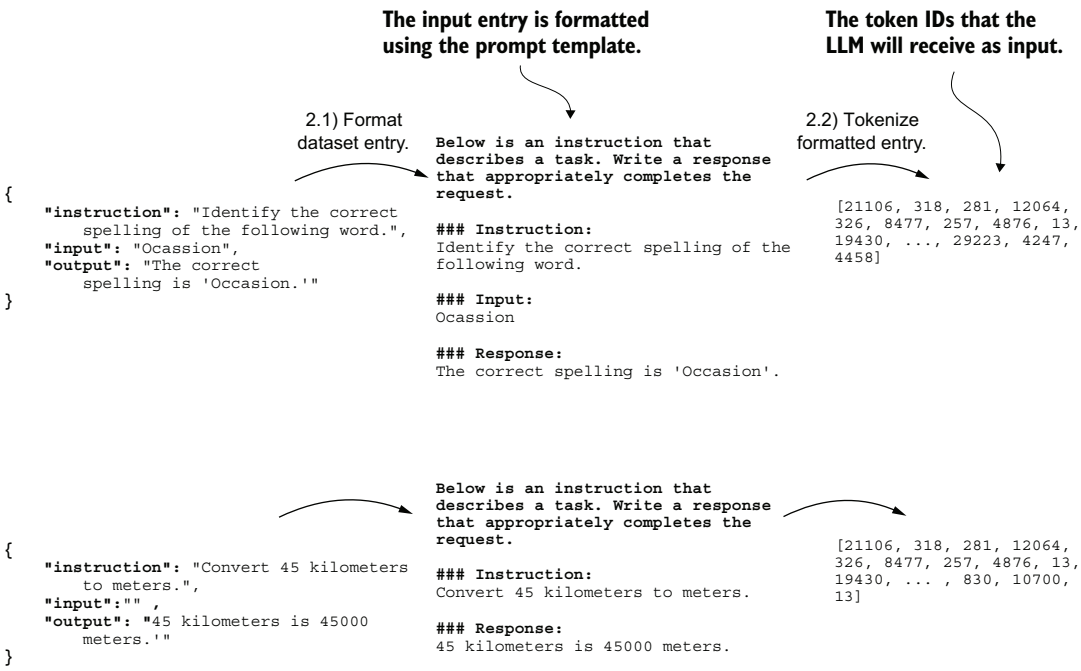


Figure 7.7 The first two steps involved in implementing the batching process. Entries are first formatted using a specific prompt template (2.1) and then tokenized (2.2), resulting in a sequence of token IDs that the model can process.

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

Similar to the approach used for classification fine-tuning, we want to accelerate training by collecting multiple training examples in a batch, which necessitates padding all inputs to a similar length. As with classification fine-tuning, we use the `<|endoftext|>` token as a padding token.

Instead of appending the `<|endoftext|>` tokens to the text inputs, we can append the token ID corresponding to `<|endoftext|>` to the pretokenized inputs directly. We can use the tokenizer's `.encode` method on an `<|endoftext|>` token to remind us which token ID we should use:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

The resulting token ID is `50256`.

Moving on to step 2.3 of the process (see figure 7.6), we adopt a more sophisticated approach by developing a custom collate function that we can pass to the data loader. This custom collate function pads the training examples in each batch to the same length while allowing different batches to have different lengths, as demonstrated in figure 7.8. This approach minimizes unnecessary padding by only extending sequences to match the longest one in each batch, not the whole dataset.

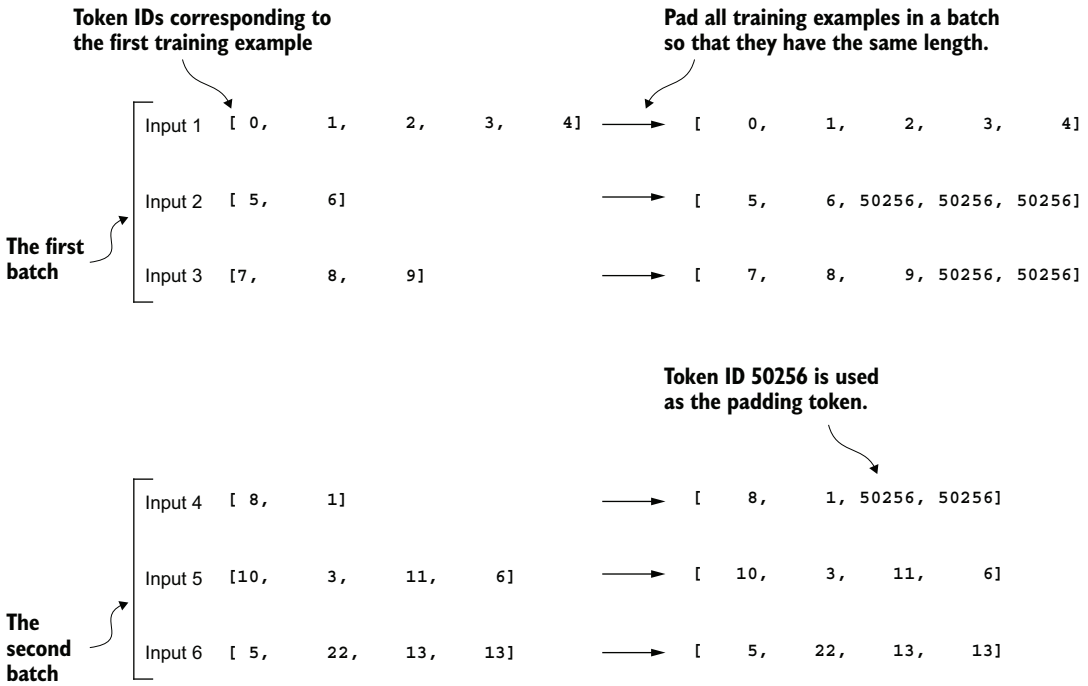


Figure 7.8 The padding of training examples in batches using token ID `50256` to ensure uniform length within each batch. Each batch may have different lengths, as shown by the first and second.

We can implement the padding process with a custom collate function:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst = []

    for 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])
        inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    return inputs_tensor
```

Finds the longest sequence in the batch

Pads and prepares inputs

Removes extra padded token added earlier

Converts the list of inputs to a tensor and transfers it to the target device

The `custom_collate_draft_1` we implemented is designed to be integrated into a PyTorch `DataLoader`, but it can also function as a standalone tool. Here, we use it independently to test and verify that it operates as intended. Let's try it on three different inputs that we want to assemble into a batch, where each example gets padded to the same length:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
    inputs_3
)
print(custom_collate_draft_1(batch))
```

The resulting batch looks like the following:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
```

This output shows all inputs have been padded to the length of the longest input list, `inputs_1`, containing five token IDs.

We have just implemented our first custom collate function to create batches from lists of inputs. However, as we previously learned, we also need to create batches with the target token IDs corresponding to the batch of input IDs. These target IDs, as shown in figure 7.9, are crucial because they represent what we want the model to generate and what we need during training to calculate the loss for the weight updates. That is, we modify our custom collate function to return the target token IDs in addition to the input token IDs.

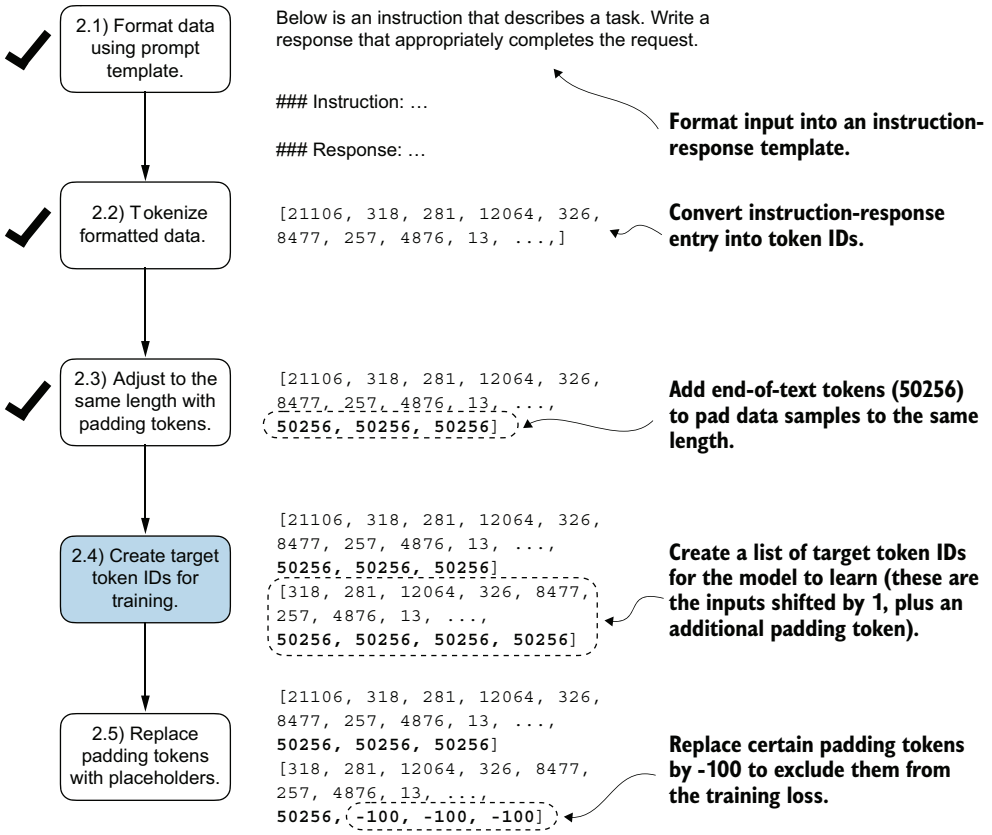


Figure 7.9 The five substeps involved in implementing the batching process. We are now focusing on step 2.4, the creation of target token IDs. This step is essential as it enables the model to learn and predict the tokens it needs to generate.

Similar to the process we used to pretrain an LLM, the target token IDs match the input token IDs but are shifted one position to the right. This setup, as shown in figure 7.10, allows the LLM to learn how to predict the next token in a sequence.

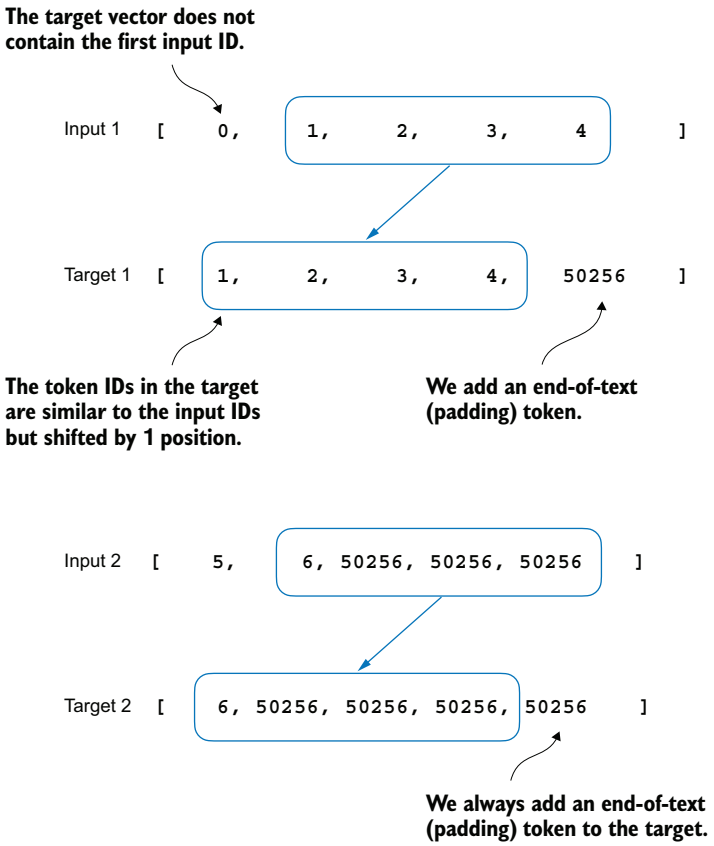


Figure 7.10 The input and target token alignment used in the instruction fine-tuning process of an LLM. For each input sequence, the corresponding target sequence is created by shifting the token IDs one position to the right, omitting the first token of the input, and appending an end-of-text token.

The following updated collate function generates the target token IDs from the input token IDs:

```
def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for 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:])
        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

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

Truncates the last token for inputs

Shifts +1 to the right for targets

Applied to the example `batch` consisting of three input lists we defined earlier, the new `custom_collate_draft_2` function now returns the input and the target batch:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
tensor([[    1,     2,     3,     4, 50256],
        [    6, 50256, 50256, 50256, 50256],
        [    8,     9, 50256, 50256, 50256]])
```

The first tensor represents inputs.

The second tensor represents the targets.

In the next step, we assign a `-100` placeholder value to all padding tokens, as highlighted in figure 7.11. This special value allows us to exclude these padding tokens from contributing to the training loss calculation, ensuring that only meaningful data influences model learning. We will discuss this process in more detail after we implement this modification. (When fine-tuning for classification, we did not have to worry about this since we only trained the model based on the last output token.)

However, note that we retain one end-of-text token, ID `50256`, in the target list, as depicted in figure 7.12. Retaining it allows the LLM to learn when to generate an end-of-text token in response to instructions, which we use as an indicator that the generated response is complete.

In the following listing, we modify our custom collate function to replace tokens with ID `50256` with `-100` in the target lists. Additionally, we introduce an `allowed_max_length` parameter to optionally limit the length of the samples. This adjustment will be useful if you plan to work with your own datasets that exceed the 1,024-token context size supported by the GPT-2 model.

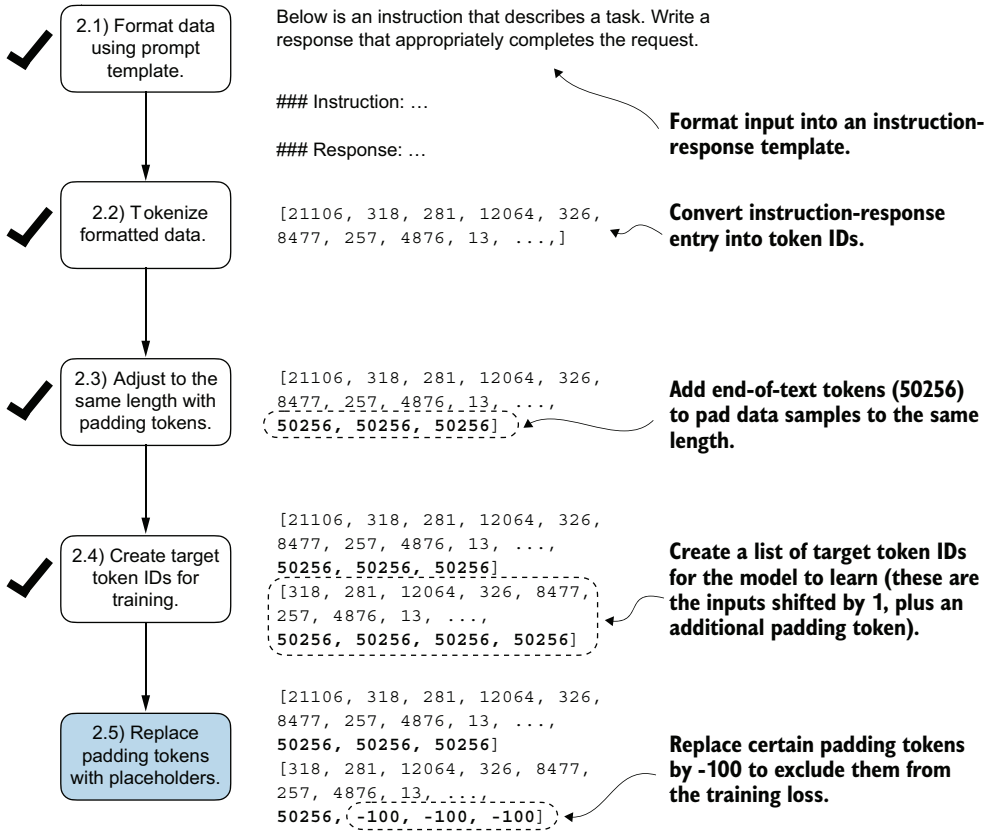


Figure 7.11 The five substeps involved in implementing the batching process. After creating the target sequence by shifting token IDs one position to the right and appending an end-of-text token, in step 2.5, we replace the end-of-text padding tokens with a placeholder value (`-100`).

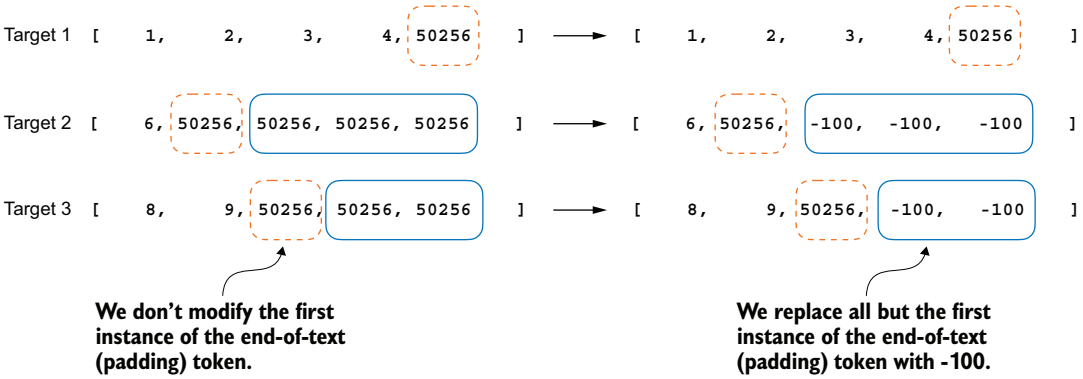


Figure 7.12 Step 2.4 in the token replacement process in the target batch for the training data preparation. We replace all but the first instance of the end-of-text token, which we use as padding, with the placeholder value `-100`, while keeping the initial end-of-text token in each target sequence.

Listing 7.5 Implementing a custom batch collate function

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

    for 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

        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
```

Pads sequences to max_length

Truncates the last token for inputs

Shifts +1 to the right for targets

Replaces all but the first padding tokens in targets by ignore_index

Optionally truncates to the maximum sequence length

Again, let's try the collate function on the sample batch that we created earlier to check that it works as intended:

```
inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)
```

The results are as follows, where the first tensor represents the inputs and the second tensor represents the targets:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
```

```
tensor([[    1,     2,     3,     4, 50256],
        [    6, 50256,  -100,  -100,  -100],
        [    8,     9, 50256,  -100,  -100]])
```

The modified collate function works as expected, altering the target list by inserting the token ID `-100`. What is the logic behind this adjustment? Let's explore the underlying purpose of this modification.

For demonstration purposes, consider the following simple and self-contained example where each output logit corresponds to a potential token from the model's vocabulary. Here's how we might calculate the cross entropy loss (introduced in chapter 5) during training when the model predicts a sequence of tokens, which is similar to what we did when we pretrained the model and fine-tuned it for classification:

```
logits_1 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5]]
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

predictions for 1st token

predictions for 2nd token

The loss value calculated by the previous code is `1.1269`:

```
tensor(1.1269)
```

As we would expect, adding an additional token ID affects the loss calculation:

```
logits_2 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5],
     [-0.5, 1.5]]
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

New third token ID prediction

After adding the third token, the loss value is `0.7936`.

So far, we have carried out some more or less obvious example calculations using the cross entropy loss function in PyTorch, the same loss function we used in the training functions for pretraining and fine-tuning for classification. Now let's get to the interesting part and see what happens if we replace the third target token ID with `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

The resulting output is

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

The resulting loss on these three training examples is identical to the loss we calculated from the two training examples earlier. In other words, the cross entropy loss function ignored the third entry in the `targets_3` vector, the token ID corresponding to `-100`. (Interested readers can try to replace the `-100` value with another token ID that is not `0` or `1`; it will result in an error.)

So what's so special about `-100` that it's ignored by the cross entropy loss? The default setting of the cross entropy function in PyTorch is `cross_entropy(..., ignore_index=-100)`. This means that it ignores targets labeled with `-100`. We take advantage of this `ignore_index` to ignore the additional end-of-text (padding) tokens that we used to pad the training examples to have the same length in each batch. However, we want to keep one `50256` (end-of-text) token ID in the targets because it helps the LLM to learn to generate end-of-text tokens, which we can use as an indicator that a response is complete.

In addition to masking out padding tokens, it is also common to mask out the target token IDs that correspond to the instruction, as illustrated in figure 7.13. By masking out the LLM's target token IDs corresponding to the instruction, the cross entropy loss is only computed for the generated response target IDs. Thus, the model is trained to focus on generating accurate responses rather than memorizing instructions, which can help reduce overfitting.

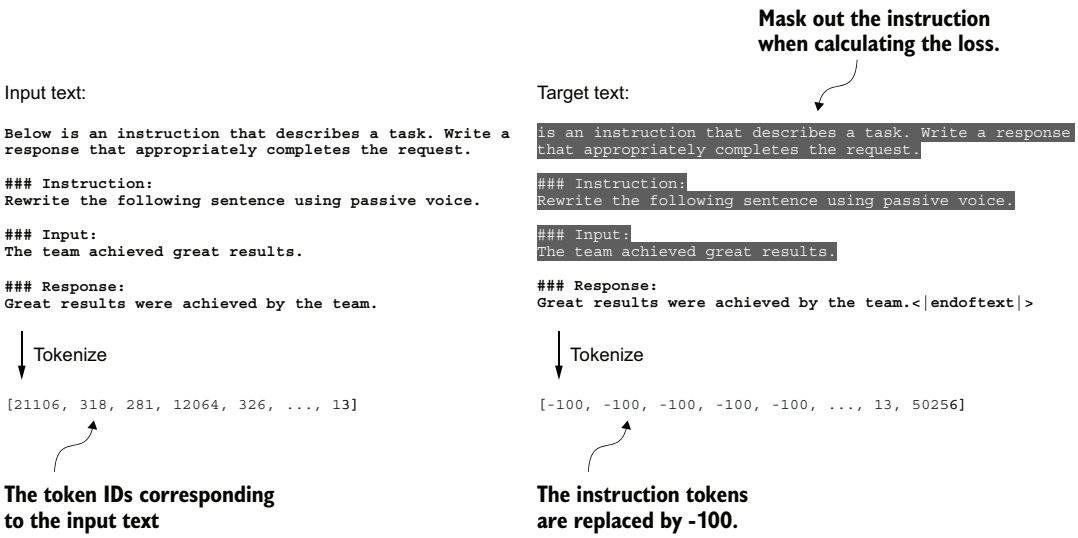


Figure 7.13 Left: The formatted input text we tokenize and then feed to the LLM during training. Right: The target text we prepare for the LLM where we can optionally mask out the instruction section, which means replacing the corresponding token IDs with the `-100` `ignore_index` value.

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

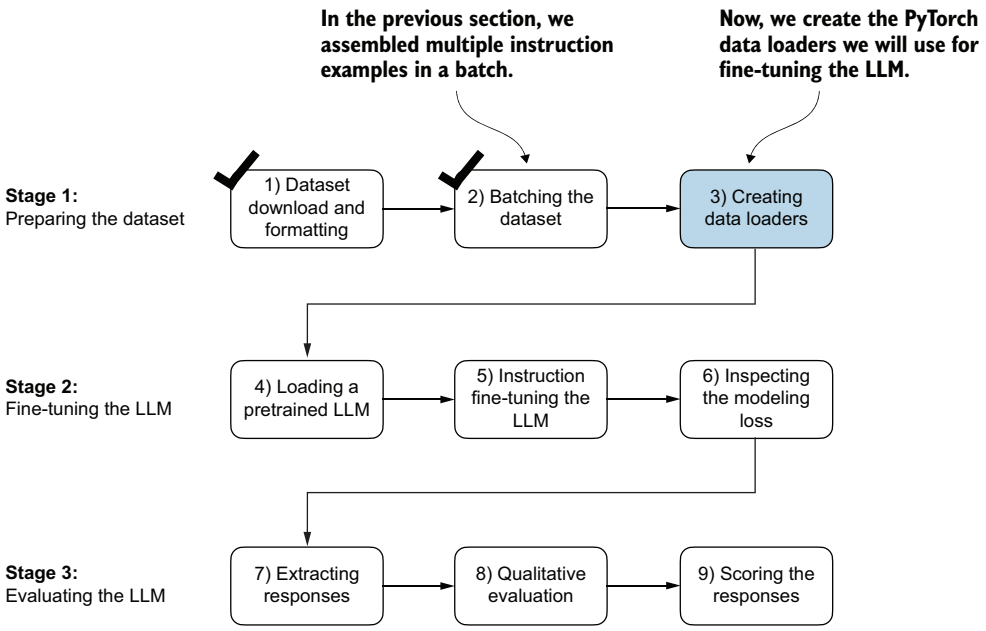


Figure 7.14 The three-stage process for instruction fine-tuning an LLM. Thus far, we have prepared the dataset and implemented a custom collate function to batch the instruction dataset. Now, we can create and apply the data loaders to the training, validation, and test sets needed for the LLM instruction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

```
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

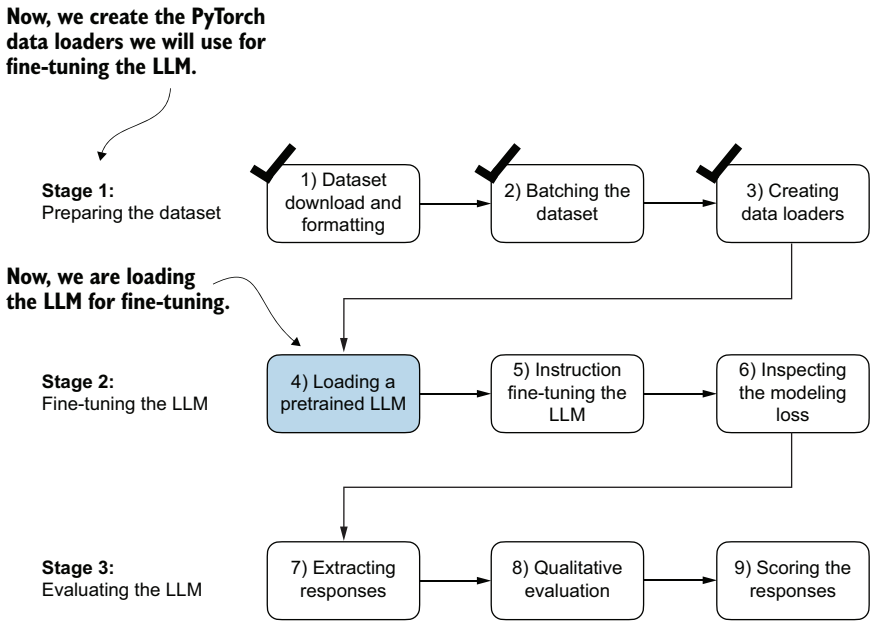


Figure 7.15 The three-stage process for instruction fine-tuning an LLM. After the dataset preparation, the process of fine-tuning an LLM for instruction-following begins with loading a pretrained LLM, which serves as the foundation for subsequent training.

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)
```

This code removes the input text from the beginning of the `generated_text`, leaving us with only the model's generated response. The `strip()` function is then applied to remove any leading or trailing whitespace characters. The output is

```
### Response:

The chef cooks the meal every day.

### Instruction:

Convert the active sentence to passive: 'The chef cooks the
```

This output shows that the pretrained model is not yet capable of correctly following the given instruction. While it does create a Response section, it simply repeats the original input sentence and part of the instruction, failing to convert the active sentence to passive voice as requested. So, let's now implement the fine-tuning process to improve the model's ability to comprehend and appropriately respond to such requests.

7.6 Fine-tuning the LLM on instruction data

It's time to fine-tune the LLM for instructions (figure 7.16). We will take the loaded pretrained model in the previous section and further train it using the previously prepared instruction dataset prepared earlier in this chapter. We already did all the hard work when we implemented the instruction dataset processing at the beginning of

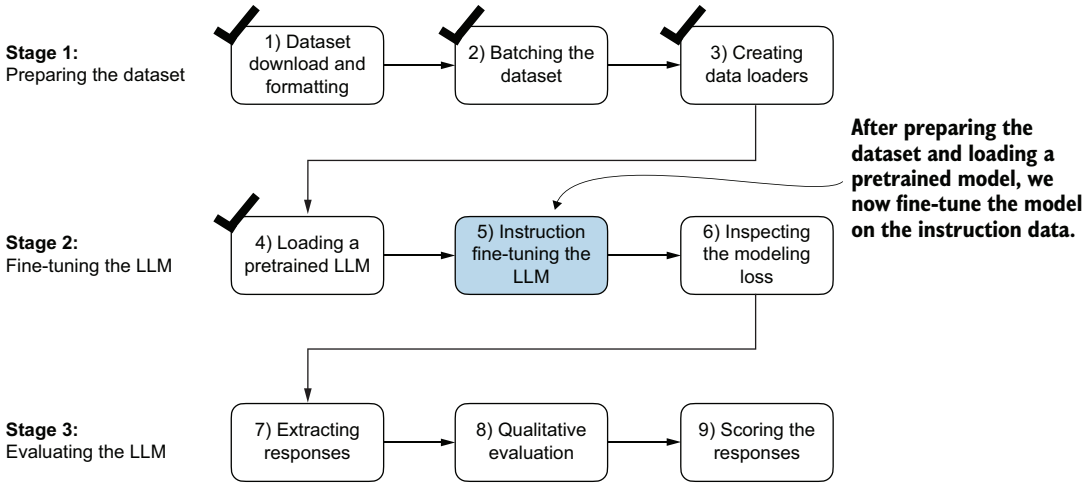


Figure 7.16 The three-stage process for instruction fine-tuning an LLM. In step 5, we train the pretrained model we previously loaded on the instruction dataset we prepared earlier.

this chapter. For the fine-tuning process itself, we can reuse the loss calculation and training functions implemented in chapter 5:

```
from chapter05 import (
    calc_loss_loader,
    train_model_simple
)
```

Before we begin training, let's calculate the initial loss for the training and validation sets:

```
model.to(device)
torch.manual_seed(123)

with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(
        val_loader, model, device, num_batches=5
)

print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

The initial loss values are as follows; as previously, our goal is to minimize the loss:

```
Training loss: 3.825908660888672
Validation loss: 3.7619335651397705
```

Dealing with hardware limitations

Using and training a larger model like GPT-2 medium (355 million parameters) is more computationally intensive than the smaller GPT-2 model (124 million parameters). If you encounter problems due to hardware limitations, you can switch to the smaller model by changing `CHOOSE_MODEL = "gpt2-medium (355M)"` to `CHOOSE_MODEL = "gpt2-small (124M)"` (see section 7.5). Alternatively, to speed up the model training, consider using a GPU. The following supplementary section in this book's code repository lists several options for using cloud GPUs: https://mng.bz/EOEq.

The following table provides reference run times for training each model on various devices, including CPUs and GPUs, for GPT-2. Running this code on a compatible GPU requires no code changes and can significantly speed up training. For the results shown in this chapter, I used the GPT-2 medium model and trained it on an A100 GPU.

Model name	Device	Run time for two epochs
gpt2-medium (355M)	CPU (M3 MacBook Air)	15.78 minutes
gpt2-medium (355M)	GPU (NVIDIA L4)	1.83 minutes
gpt2-medium (355M)	GPU (NVIDIA A100)	0.86 minutes
gpt2-small (124M)	CPU (M3 MacBook Air)	5.74 minutes
gpt2-small (124M)	GPU (NVIDIA L4)	0.69 minutes
gpt2-small (124M)	GPU (NVIDIA A100)	0.39 minutes

With the model and data loaders prepared, we can now proceed to train the model. The code in listing 7.8 sets up the training process, including initializing the optimizer, setting the number of epochs, and defining the evaluation frequency and starting context to evaluate generated LLM responses during training based on the first validation set instruction (`val_data[0]`) we looked at in section 7.5.

Listing 7.8 Instruction fine-tuning the pretrained LLM

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1
)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

The following output displays the training progress over two epochs, where a steady decrease in losses indicates improving ability to follow instructions and generate appropriate responses:

```
Ep 1 (Step 000000): Train loss 2.637, Val loss 2.626
Ep 1 (Step 000005): Train loss 1.174, Val loss 1.103
Ep 1 (Step 000010): Train loss 0.872, Val loss 0.944
Ep 1 (Step 000015): Train loss 0.857, Val loss 0.906
...
```

```
Ep 1 (Step 000115): Train loss 0.520, Val loss 0.665
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.'
### Response: The meal is prepared every day by the chef.<|endoftext|>
The following is an instruction that describes a task.
Write a response that appropriately completes the request.
### Instruction: Convert the active sentence to passive:
Ep 2 (Step 000120): Train loss 0.438, Val loss 0.670
Ep 2 (Step 000125): Train loss 0.453, Val loss 0.685
Ep 2 (Step 000130): Train loss 0.448, Val loss 0.681
Ep 2 (Step 000135): Train loss 0.408, Val loss 0.677
...
Ep 2 (Step 000230): Train loss 0.300, Val loss 0.657
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.'  ### Response: The meal is cooked every day by the
chef.<|endoftext|>The following is an instruction that describes
a task. Write a response that appropriately completes the request.
### Instruction: What is the capital of the United Kingdom
Training completed in 0.87 minutes.
```

The training output shows that the model is learning effectively, as we can tell based on the consistently decreasing training and validation loss values over the two epochs. This result suggests that the model is gradually improving its ability to understand and follow the provided instructions. (Since the model demonstrated effective learning within these two epochs, extending the training to a third epoch or more is not essential and may even be counterproductive as it could lead to increased overfitting.)

Moreover, the generated responses at the end of each epoch let us inspect the model's progress in correctly executing the given task in the validation set example. In this case, the model successfully converts the active sentence `"The chef cooks the meal every day."` into its passive voice counterpart: `"The meal is cooked every day by the chef."`

We will revisit and evaluate the response quality of the model in more detail later. For now, let's examine the training and validation loss curves to gain additional insights into the model's learning process. For this, we use the same `plot_losses` function we used for pretraining:

```
from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

From the loss plot shown in figure 7.17, we can see that the model's performance on both the training and validation sets improves substantially over the course of training. The rapid decrease in losses during the initial phase indicates that the model quickly learns meaningful patterns and representations from the data. Then, as training progresses to the second epoch, the losses continue to decrease but at a slower

rate, suggesting that the model is fine-tuning its learned representations and converging to a stable solution.

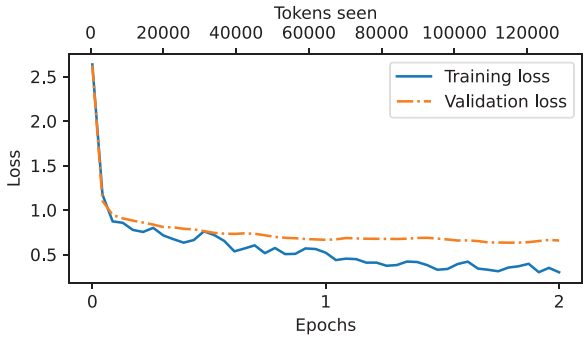


Figure 7.17 The training and validation loss trends over two epochs. The solid line represents the training loss, showing a sharp decrease before stabilizing, while the dotted line represents the validation loss, which follows a similar pattern.

While the loss plot in figure 7.17 indicates that the model is training effectively, the most crucial aspect is its performance in terms of response quality and correctness. So, next, let's extract the responses and store them in a format that allows us to evaluate and quantify the response quality.

Exercise 7.3 Fine-tuning on the original Alpaca dataset

The Alpaca dataset, by researchers at Stanford, is one of the earliest and most popular openly shared instruction datasets, consisting of 52,002 entries. As an alternative to the `instruction-data.json` file we use here, consider fine-tuning an LLM on this dataset. The dataset is available at https://mng.bz/NBnE.

This dataset contains 52,002 entries, which is approximately 50 times more than those we used here, and most entries are longer. Thus, I highly recommend using a GPU to conduct the training, which will accelerate the fine-tuning process. If you encounter out-of-memory errors, consider reducing the `batch_size` from 8 to 4, 2, or even 1. Lowering the `allowed_max_length` from 1,024 to 512 or 256 can also help manage memory problems.

7.7 Extracting and saving responses

Having fine-tuned the LLM on the training portion of the instruction dataset, we are now ready to evaluate its performance on the held-out test set. First, we extract the model-generated responses for each input in the test dataset and collect them for manual analysis, and then we evaluate the LLM to quantify the quality of the responses, as highlighted in figure 7.18.

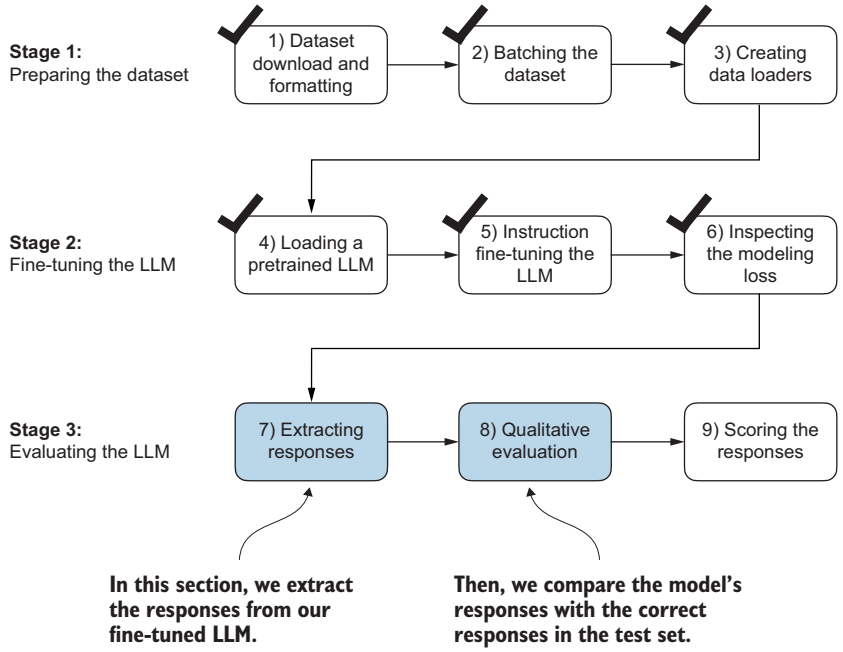


Figure 7.18 The three-stage process for instruction fine-tuning the LLM. In the first two steps of stage 3, we extract and collect the model responses on the held-out test dataset for further analysis and then evaluate the model to quantify the performance of the instruction-fine-tuned LLM.

To complete the response instruction step, we use the `generate` function. We then print the model responses alongside the expected test set answers for the first three test set entries, presenting them side by side for comparison:

```
torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    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("### Response:", "")
        .strip()
    )
```

Iterates over the first three test set samples

Uses the generate function imported in section 7.5

```
print(input_text)
print(f"\nCorrect response:\n>> {entry['output']}")
print(f"\nModel response:\n>> {response_text.strip()}")
print("---------------------------------------")
```

As mentioned earlier, the `generate` function returns the combined input and output text, so we use slicing and the `.replace()` method on the `generated_text` contents to extract the model's response. The instructions, followed by the given test set response and model response, are shown next.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Rewrite the sentence using a simile.

Input:

The car is very fast.

Correct response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

What type of cloud is typically associated with thunderstorms?

Correct response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Name the author of 'Pride and Prejudice.'

Correct response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

As we can see based on the test set instructions, given responses, and the model's responses, the model performs relatively well. The answers to the first and last instructions are clearly correct, while the second answer is close but not entirely accurate. The model answers with "cumulus cloud" instead of "cumulonimbus," although it's worth noting that cumulus clouds can develop into cumulonimbus clouds, which are capable of producing thunderstorms.

Most importantly, model evaluation is not as straightforward as it is for classification fine-tuning, where we simply calculate the percentage of correct spam/non-spam class labels to obtain the classification's accuracy. In practice, instruction-fine-tuned LLMs such as chatbots are evaluated via multiple approaches:

- Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; https://arxiv.org/abs/2009.03300), which test the general knowledge of a model.
- Human preference comparison to other LLMs, such as LMSYS chatbot arena (https://arena.lmsys.org).
- Automated conversational benchmarks, where another LLM like GPT-4 is used to evaluate the responses, such as AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/).

In practice, it can be useful to consider all three types of evaluation methods: multiple-choice question answering, human evaluation, and automated metrics that measure conversational performance. However, since we are primarily interested in assessing conversational performance rather than just the ability to answer multiple-choice questions, human evaluation and automated metrics may be more relevant.

Conversational performance

Conversational performance of LLMs refers to their ability to engage in human-like communication by understanding context, nuance, and intent. It encompasses skills such as providing relevant and coherent responses, maintaining consistency, and adapting to different topics and styles of interaction.

Human evaluation, while providing valuable insights, can be relatively laborious and time-consuming, especially when dealing with a large number of responses. For instance, reading and assigning ratings to all 1,100 responses would require a significant amount of effort.

So, considering the scale of the task at hand, we will implement an approach similar to automated conversational benchmarks, which involves evaluating the responses automatically using another LLM. This method will allow us to efficiently assess the quality of the generated responses without the need for extensive human involvement, thereby saving time and resources while still obtaining meaningful performance indicators.

Let's employ an approach inspired by AlpacaEval, using another LLM to evaluate our fine-tuned model's responses. However, instead of relying on a publicly available benchmark dataset, we use our own custom test set. This customization allows for a more targeted and relevant assessment of the model's performance within the context of our intended use cases, represented in our instruction dataset.

To prepare the responses for this evaluation process, we append the generated model responses to the `test_set` dictionary and save the updated data as an `"instruction-data-with-response.json"` file for record keeping. Additionally, by saving this file, we can easily load and analyze the responses in separate Python sessions later on if needed.

The following code listing uses the `generate` method in the same manner as before; however, we now iterate over the entire `test_set`. Also, instead of printing the model responses, we add them to the `test_set` dictionary.

Listing 7.9 Generating test set responses

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)

    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("### Response:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file:
    json.dump(test_data, file, indent=4)
```

indent for pretty-printing

Processing the dataset takes about 1 minute on an A100 GPU and 6 minutes on an M3 MacBook Air:

```
100%|██████████| 110/110 [01:05<00:00,  1.68it/s]
```

Let's verify that the responses have been correctly added to the `test_set` dictionary by examining one of the entries:

```
print(test_data[0])
```

The output shows that the `model_response` has been added correctly:

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

Finally, we save the model as `gpt2-medium355M-sft.pth` file to be able to reuse it in future projects:

```
import re

file_name = f"{re.sub(r'[ ()]', '', CHOOSE_MODEL) }-sft.pth"
torch.save(model.state_dict(), file_name)
print(f"Model saved as {file_name}")
```

Removes white spaces and parentheses from file name

The saved model can then be loaded via `model.load_state_dict(torch.load("gpt2-medium355M-sft.pth"))`.

7.8 Evaluating the fine-tuned LLM

Previously, we judged the performance of an instruction-fine-tuned model by looking at its responses on three examples of the test set. While this gives us a rough idea of how well the model performs, this method does not scale well to larger amounts of responses. So, we implement a method to automate the response evaluation of the fine-tuned LLM using another, larger LLM, as highlighted in figure 7.19.

To evaluate test set responses in an automated fashion, we utilize an existing instruction-fine-tuned 8-billion-parameter Llama 3 model developed by Meta AI. This model can be run locally using the open source Ollama application (https://ollama.com).

> **NOTE** Ollama is an efficient application for running LLMs on a laptop. It serves as a wrapper around the open source llama.cpp library (https://github.com/ggerganov/llama.cpp), which implements LLMs in pure C/C++ to maximize efficiency. However, Ollama is only a tool for generating text using LLMs (inference) and does not support training or fine-tuning LLMs.

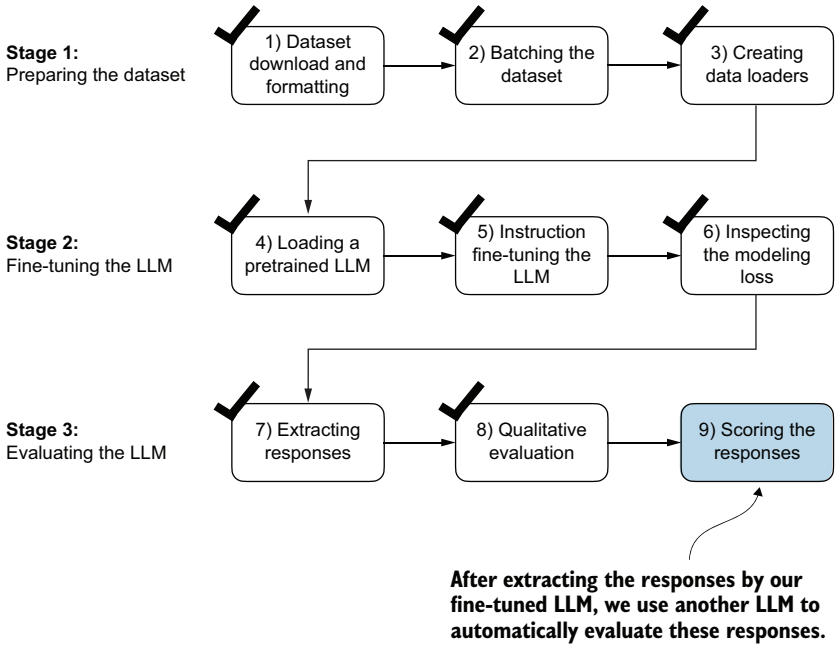


Figure 7.19 The three-stage process for instruction fine-tuning the LLM. In this last step of the instruction-fine-tuning pipeline, we implement a method to quantify the performance of the fine-tuned model by scoring the responses it generated for the test.

Using larger LLMs via web APIs

The 8-billion-parameter Llama 3 model is a very capable LLM that runs locally. However, it's not as capable as large proprietary LLMs such as GPT-4 offered by OpenAI. For readers interested in exploring how to utilize GPT-4 through the OpenAI API to assess generated model responses, an optional code notebook is available within the supplementary materials accompanying this book at https://mng.bz/BgEv.

To execute the following code, install Ollama by visiting https://ollama.com and follow the provided instructions for your operating system:

- *For macOS and Windows users*—Open the downloaded Ollama application. If prompted to install command-line usage, select Yes.
- *For Linux users*—Use the installation command available on the Ollama website.

Before implementing the model evaluation code, let's first download the Llama 3 model and verify that Ollama is functioning correctly by using it from the command-line terminal. To use Ollama from the command line, you must either start the Ollama application or run `ollama serve` in a separate terminal, as shown in figure 7.20.

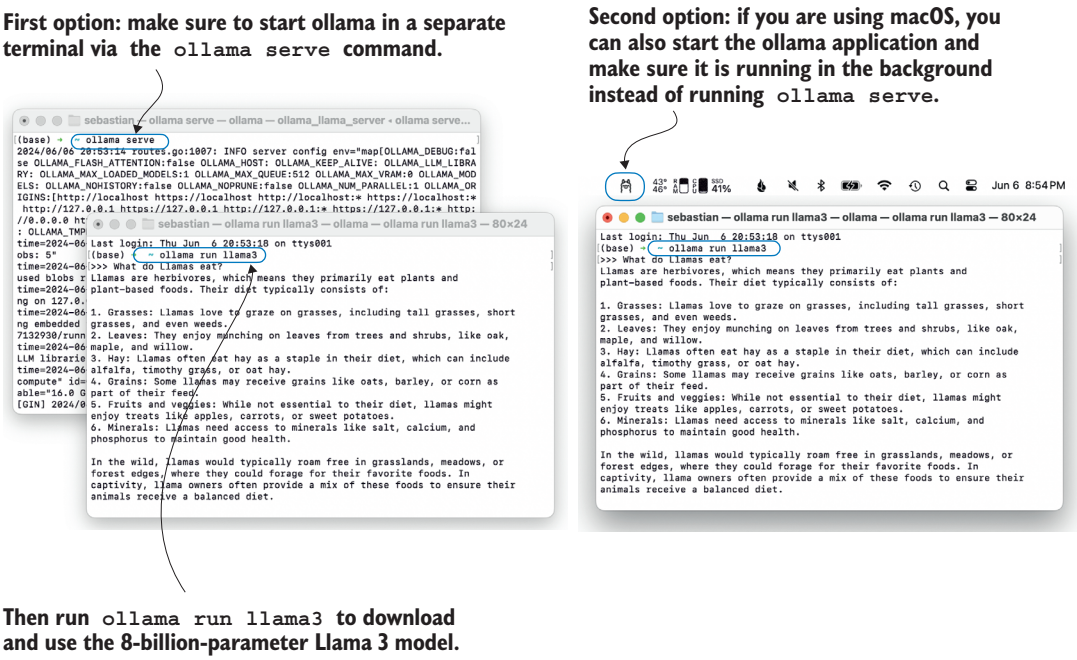


Figure 7.20 Two options for running Ollama. The left panel illustrates starting Ollama using `ollama serve`. The right panel shows a second option in macOS, running the Ollama application in the background instead of using the `ollama serve` command to start the application.

With the Ollama application or `ollama serve` running in a different terminal, execute the following command on the command line (not in a Python session) to try out the 8-billion-parameter Llama 3 model:

```
ollama run llama3
```

The first time you execute this command, this model, which takes up 4.7 GB of storage space, will be automatically downloaded. The output looks like the following:

```
pulling manifest
pulling 6a0746a1ec1a... 100% |████████████████| 4.7 GB
pulling 4fa551d4f938... 100% |████████████████|  12 KB
pulling 8ab4849b038c... 100% |████████████████|  254 B
pulling 577073ffcc6c... 100% |████████████████|  110 B
pulling 3f8eb4da87fa... 100% |████████████████|  485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

Alternative Ollama models

The `llama3` in the `ollama run llama3` command refers to the instruction-fine-tuned 8-billion-parameter Llama 3 model. Using Ollama with the `llama3` model requires approximately 16 GB of RAM. If your machine does not have sufficient RAM, you can try using a smaller model, such as the 3.8-billion-parameter `phi3` model via `ollama run llama3`, which only requires around 8 GB of RAM.

For more powerful computers, you can also use the larger 70-billion-parameter Llama 3 model by replacing `llama3` with `llama3:70b`. However, this model requires significantly more computational resources.

Once the model download is complete, we are presented with a command-line interface that allows us to interact with the model. For example, try asking the model, "What do llamas eat?"

```
>>> What do llamas eat?
Llamas are ruminant animals, which means they have a four-chambered
stomach and eat plants that are high in fiber. In the wild,
llamas typically feed on:

1. Grasses: They love to graze on various types of grasses, including tall
grasses, wheat, oats, and barley.
```

Note that the response you see might differ since Ollama is not deterministic as of this writing.

You can end this `ollama run llama3` session using the input `/bye`. However, make sure to keep the `ollama serve` command or the Ollama application running for the remainder of this chapter.

The following code verifies that the Ollama session is running properly before we use Ollama to evaluate the test set responses:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(["name"]):
        if process_name in proc.info["name"]:
            running = True
            break
    return running

ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError(
        "Ollama not running. Launch ollama before proceeding."
)
print("Ollama running:", check_if_running("ollama"))
```

Ensure that the output from executing the previous code displays `Ollama running: True`. If it shows `False`, verify that the `ollama serve` command or the Ollama application is actively running.

Running the code in a new Python session

If you already closed your Python session or if you prefer to execute the remaining code in a different Python session, use the following code, which loads the instruction and response data file we previously created and redefines the `format_input` function we used earlier (the `tqdm` progress bar utility is used later):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )


    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

An alternative to the `ollama run` command for interacting with the model is through its REST API using Python. The `query_model` function shown in the following listing demonstrates how to use the API.

Listing 7.10 Querying a local Ollama model

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {                    <-- Creates the data payload as a dictionary
        "model": model,
        "messages": [
            {"role": "user", "content": prompt}
        ],
        "options": {            <-- Settings for deterministic responses
            "seed": 123,
```

```
            "temperature": 0,
            "num_ctx": 2048
        }
    }

    payload = json.dumps(data).encode("utf-8")
    request = urllib.request.Request(
        url,
        data=payload,
        method="POST"
    )

    request.add_header("Content-Type", "application/json")

    response_data = ""
    with urllib.request.urlopen(request) as response:
        while True:
            line = response.readline().decode("utf-8")
            if not line:
                break
            response_json = json.loads(line)
            response_data += response_json["message"]["content"]

    return response_data
```

Converts the dictionary to a JSON-formatted string and encodes it to bytes

Creates a request object, setting the method to POST and adding necessary headers

Sends the request and captures the response

Before running the subsequent code cells in this notebook, ensure that Ollama is still running. The previous code cells should print `"Ollama running: True"` to confirm that the model is active and ready to receive requests.

The following is an example of how to use the `query_model` function we just implemented:

```
model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)
```

The resulting response is as follows:

```
Llamas are ruminant animals, which means they have a four-chambered
stomach that allows them to digest plant-based foods. Their diet
typically consists of:

1. Grasses: Llamas love to graze on grasses, including tall grasses,
short grasses, and even weeds.
...
```

Using the `query_model` function defined earlier, we can evaluate the responses generated by our fine-tuned model that prompts the Llama 3 model to rate our fine-tuned model's responses on a scale from 0 to 100 based on the given test set response as reference.

First, we apply this approach to the first three examples from the test set that we previously examined:

```
for entry in test_data[:3]:
    prompt = (
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']}`, "
        f"score the model response `{entry['model_response']}`"
        f" on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])
    print("\nModel response:")
    print(">>", entry["model_response"])
    print("\nScore:")
    print(">>", query_model(prompt))
    print("\n-------------------------")
```

This code prints outputs similar to the following (as of this writing, Ollama is not fully deterministic, so the generated texts may vary):

Dataset response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Score:

>> I'd rate the model response "The car is as fast as a bullet." an 85 out of 100.

Here's why:

The response uses a simile correctly, comparing the speed of the car to something else (in this case, a bullet).

The comparison is relevant and makes sense, as bullets are known for their high velocity.

The phrase "as fast as" is used correctly to introduce the simile.

The only reason I wouldn't give it a perfect score is that some people might find the comparison slightly less vivid or evocative than others. For example, comparing something to lightning (as in the original response) can be more dramatic and attention grabbing. However, "as fast as a bullet" is still a strong and effective simile that effectively conveys the idea of the car's speed.

Overall, I think the model did a great job!

Dataset response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Score:

>> I'd score this model response as 40 out of 100.

Here's why:

The model correctly identifies that thunderstorms are related to clouds (correctly identifying the type of phenomenon).

However, it incorrectly specifies the type of cloud associated with thunderstorms. Cumulus clouds are not typically associated with thunderstorms; cumulonimbus clouds are.

The response lacks precision and accuracy in its description.

Overall, while the model attempts to address the instruction, it provides an incorrect answer, which is a significant error.

Dataset response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

Score:

>> I'd rate my own response as 95 out of 100. Here's why:

The response accurately answers the question by naming the author of *Pride and Prejudice* as Jane Austen.

The response is concise and clear, making it easy to understand.

There are no grammatical errors or ambiguities that could lead to confusion.

The only reason I wouldn't give myself a perfect score is that the response is slightly redundant—it's not necessary to rephrase the question in the answer. A more concise response would be simply "Jane Austen."

The generated responses show that the Llama 3 model provides reasonable evaluations and is capable of assigning partial points when a model's answer is not entirely correct. For instance, if we consider the evaluation of the "cumulus cloud" answer, the model acknowledges the partial correctness of the response.

The previous prompt returns highly detailed evaluations in addition to the score. We can modify the prompt to just generate integer scores ranging from 0 to 100, where 100 represents the best possible score. This modification allows us to calculate an average score for our model, which serves as a more concise and quantitative assessment of its performance. The `generate_model_scores` function shown in the following listing uses a modified prompt telling the model to `"Respond with the integer number only."`

Listing 7.11 Evaluating the instruction fine-tuning LLM

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `{format_input(entry)}` "
            f"and correct output `{entry['output']}`, "
            f"score the model response `{entry[json_key]}`"
            f" on a scale from 0 to 100, where 100 is the best score. "
            f"Respond with the integer number only."
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Modified instruction line to only return the score

Let's now apply the `generate_model_scores` function to the entire `test_data` set, which takes about 1 minute on a M3 Macbook Air:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

The results are as follows:

```
Scoring entries: 100%|████████████████████████| 110/110
[01:10<00:00,  1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

The evaluation output shows that our fine-tuned model achieves an average score above 50, which provides a useful benchmark for comparison against other models

or for experimenting with different training configurations to improve the model's performance.

It's worth noting that Ollama is not entirely deterministic across operating systems at the time of this writing, which means that the scores you obtain might vary slightly from the previous scores. To obtain more robust results, you can repeat the evaluation multiple times and average the resulting scores.

To further improve our model's performance, we can explore various strategies, such as

- Adjusting the hyperparameters during fine-tuning, such as the learning rate, batch size, or number of epochs
- Increasing the size of the training dataset or diversifying the examples to cover a broader range of topics and styles
- Experimenting with different prompts or instruction formats to guide the model's responses more effectively
- Using a larger pretrained model, which may have greater capacity to capture complex patterns and generate more accurate responses

NOTE For reference, when using the methodology described herein, the Llama 3 8B base model, without any fine-tuning, achieves an average score of 58.51 on the test set. The Llama 3 8B instruct model, which has been fine-tuned on a general instruction-following dataset, achieves an impressive average score of 82.6.

Exercise 7.4 Parameter-efficient fine-tuning with LoRA

To instruction fine-tune an LLM more efficiently, modify the code in this chapter to use the low-rank adaptation method (LoRA) from appendix E. Compare the training run time and model performance before and after the modification.

7.9 Conclusions

This chapter marks the conclusion of our journey through the LLM development cycle. We have covered all the essential steps, including implementing an LLM architecture, pretraining an LLM, and fine-tuning it for specific tasks, as summarized in figure 7.21. Let's discuss some ideas for what to look into next.

7.9.1 What's next?

While we covered the most essential steps, there is an optional step that can be performed after instruction fine-tuning: preference fine-tuning. Preference fine-tuning is particularly useful for customizing a model to better align with specific user preferences. If you are interested in exploring this further, see the `04_preference-tuning-with-dpo` folder in this book's supplementary GitHub repository at https://mng.bz/dZwD.

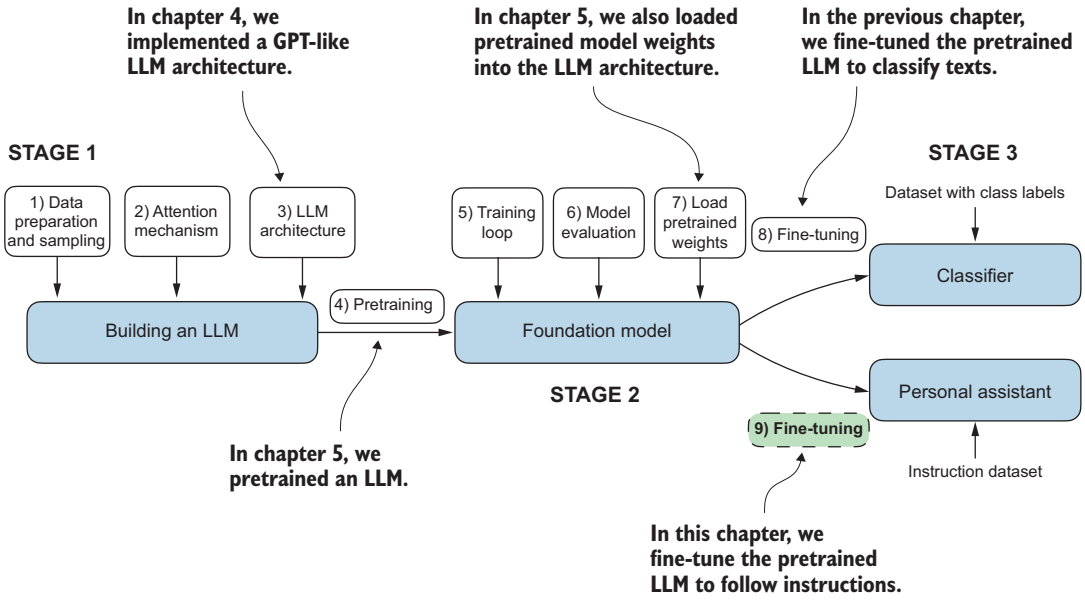


Figure 7.21 The three main stages of coding an LLM.

In addition to the main content covered in this book, the GitHub repository also contains a large selection of bonus material that you may find valuable. To learn more about these additional resources, visit the Bonus Material section on the repository's README page: https://mng.bz/r12g.

7.9.2 Staying up to date in a fast-moving field

The fields of AI and LLM research are evolving at a rapid (and, depending on who you ask, exciting) pace. One way to keep up with the latest advancements is to explore recent research papers on arXiv at https://arxiv.org/list/cs.LG/recent. Additionally, many researchers and practitioners are very active in sharing and discussing the latest developments on social media platforms like X (formerly Twitter) and Reddit. The subreddit r/LocalLLaMA, in particular, is a good resource for connecting with the community and staying informed about the latest tools and trends. I also regularly share insights and write about the latest in LLM research on my blog, available at https://magazine.sebastianraschka.com and https://sebastianraschka.com/blog/.

7.9.3 Final words

I hope you have enjoyed this journey of implementing an LLM from the ground up and coding the pretraining and fine-tuning functions from scratch. In my opinion, building an LLM from scratch is the most effective way to gain a deep understanding of how LLMs work. I hope that this hands-on approach has provided you with valuable insights and a solid foundation in LLM development.

While the primary purpose of this book is educational, you may be interested in utilizing different and more powerful LLMs for real-world applications. For this, I recommend exploring popular tools such as Axolotl (https://github.com/OpenAccess-AI-Collective/axolotl) or LitGPT (https://github.com/Lightning-AI/litgpt), which I am actively involved in developing.

Thank you for joining me on this learning journey, and I wish you all the best in your future endeavors in the exciting field of LLMs and AI!

Summary

- The instruction-fine-tuning process adapts a pretrained LLM to follow human instructions and generate desired responses.
- Preparing the dataset involves downloading an instruction-response dataset, formatting the entries, and splitting it into train, validation, and test sets.
- Training batches are constructed using a custom collate function that pads sequences, creates target token IDs, and masks padding tokens.
- We load a pretrained GPT-2 medium model with 355 million parameters to serve as the starting point for instruction fine-tuning.
- The pretrained model is fine-tuned on the instruction dataset using a training loop similar to pretraining.
- Evaluation involves extracting model responses on a test set and scoring them (for example, using another LLM).
- The Ollama application with an 8-billion-parameter Llama model can be used to automatically score the fine-tuned model's responses on the test set, providing an average score to quantify performance.

appendix A
Introduction to PyTorch

This appendix is designed to equip you with the necessary skills and knowledge to put deep learning into practice and implement large language models (LLMs) from scratch. PyTorch, a popular Python-based deep learning library, will be our primary tool for this book. I will guide you through setting up a deep learning workspace armed with PyTorch and GPU support.

Then you'll learn about the essential concept of tensors and their usage in PyTorch. We will also delve into PyTorch's automatic differentiation engine, a feature that enables us to conveniently and efficiently use backpropagation, which is a crucial aspect of neural network training.

This appendix is meant as a primer for those new to deep learning in PyTorch. While it explains PyTorch from the ground up, it's not meant to be an exhaustive coverage of the PyTorch library. Instead, we'll focus on the PyTorch fundamentals we will use to implement LLMs. If you are already familiar with deep learning, you may skip this appendix and directly move on to chapter 2.

A.1 What is PyTorch?

PyTorch (https://pytorch.org/) is an open source Python-based deep learning library. According to *Papers With Code* (https://paperswithcode.com/trends), a platform that tracks and analyzes research papers, PyTorch has been the most widely used deep learning library for research since 2019 by a wide margin. And, according to the *Kaggle Data Science and Machine Learning Survey 2022* (https://www.kaggle.com/c/kaggle-survey-2022), the number of respondents using PyTorch is approximately 40%, which grows every year.

One of the reasons PyTorch is so popular is its user-friendly interface and efficiency. Despite its accessibility, it doesn't compromise on flexibility, allowing advanced users to tweak lower-level aspects of their models for customization and

optimization. In short, for many practitioners and researchers, PyTorch offers just the right balance between usability and features.

A.1.1 The three core components of PyTorch

PyTorch is a relatively comprehensive library, and one way to approach it is to focus on its three broad components, summarized in figure A.1.

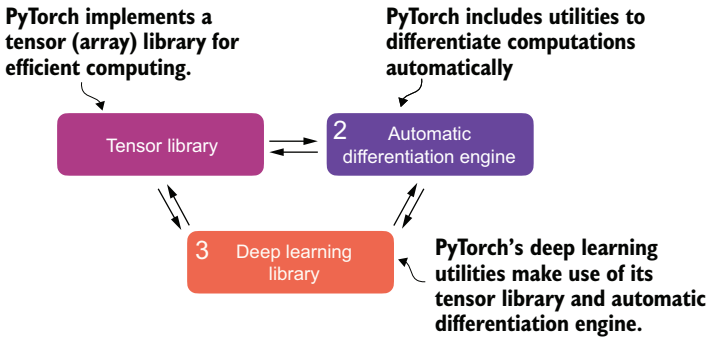


Figure A.1 PyTorch's three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to implement and train deep neural network models.

First, PyTorch is a *tensor library* that extends the concept of the array-oriented programming library NumPy with the additional feature that accelerates computation on GPUs, thus providing a seamless switch between CPUs and GPUs. Second, PyTorch is an *automatic differentiation engine*, also known as autograd, that enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization. Finally, PyTorch is a *deep learning library*. It offers modular, flexible, and efficient building blocks, including pretrained models, loss functions, and optimizers, for designing and training a wide range of deep learning models, catering to both researchers and developers.

A.1.2 Defining deep learning

In the news, LLMs are often referred to as AI models. However, LLMs are also a type of deep neural network, and PyTorch is a deep learning library. Sound confusing? Let's take a brief moment and summarize the relationship between these terms before we proceed.

AI is fundamentally about creating computer systems capable of performing tasks that usually require human intelligence. These tasks include understanding natural language, recognizing patterns, and making decisions. (Despite significant progress, AI is still far from achieving this level of general intelligence.)

Machine learning represents a subfield of AI, as illustrated in figure A.2, that focuses on developing and improving learning algorithms. The key idea behind machine learning is to enable computers to learn from data and make predictions or decisions without being explicitly programmed to perform the task. This involves developing algorithms that can identify patterns, learn from historical data, and improve their performance over time with more data and feedback.

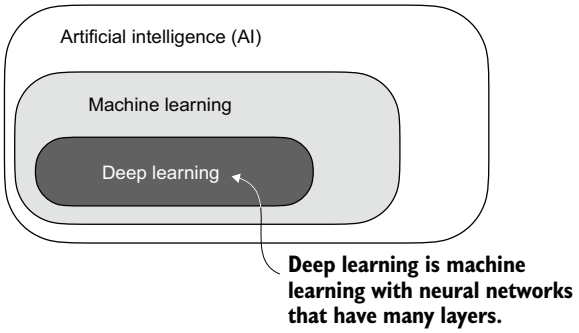


Figure A.2 Deep learning is a subcategory of machine learning focused on implementing deep neural networks. Machine learning is a subcategory of AI that is concerned with algorithms that learn from data. AI is the broader concept of machines being able to perform tasks that typically require human intelligence.

Machine learning has been integral in the evolution of AI, powering many of the advancements we see today, including LLMs. Machine learning is also behind technologies like recommendation systems used by online retailers and streaming services, email spam filtering, voice recognition in virtual assistants, and even self-driving cars. The introduction and advancement of machine learning have significantly enhanced AI's capabilities, enabling it to move beyond strict rule-based systems and adapt to new inputs or changing environments.

Deep learning is a subcategory of machine learning that focuses on the training and application of deep neural networks. These deep neural networks were originally inspired by how the human brain works, particularly the interconnection between many neurons. The "deep" in deep learning refers to the multiple hidden layers of artificial neurons or nodes that allow them to model complex, nonlinear relationships in the data. Unlike traditional machine learning techniques that excel at simple pattern recognition, deep learning is particularly good at handling unstructured data like images, audio, or text, so it is particularly well suited for LLMs.

The typical predictive modeling workflow (also referred to as *supervised learning*) in machine learning and deep learning is summarized in figure A.3.

Using a learning algorithm, a model is trained on a training dataset consisting of examples and corresponding labels. In the case of an email spam classifier, for example, the training dataset consists of emails and their "spam" and "not spam" labels that a human identified. Then the trained model can be used on new observations (i.e., new emails) to predict their unknown label ("spam" or "not spam"). Of course, we also want to add a model evaluation between the training and inference stages to

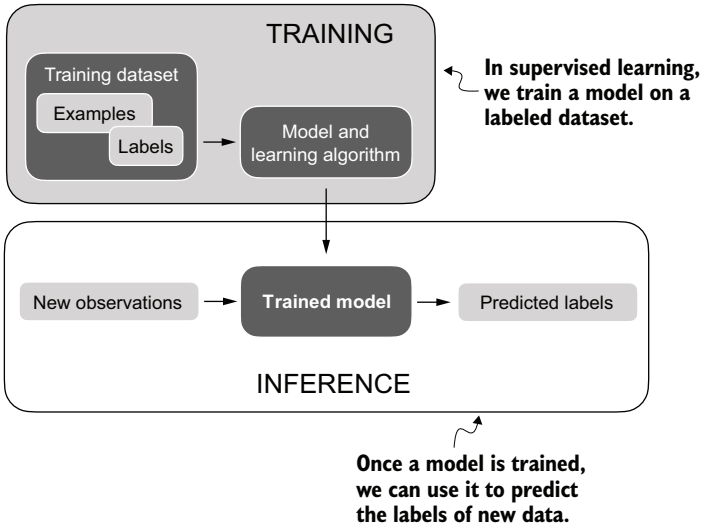


Figure A.3 The supervised learning workflow for predictive modeling consists of a training stage where a model is trained on labeled examples in a training dataset. The trained model can then be used to predict the labels of new observations.

ensure that the model satisfies our performance criteria before using it in a real-world application.

If we train LLMs to classify texts, the workflow for training and using LLMs is similar to that depicted in figure A.3. If we are interested in training LLMs to generate texts, which is our main focus, figure A.3 still applies. In this case, the labels during pretraining can be derived from the text itself (the next-word prediction task introduced in chapter 1). The LLM will generate entirely new text (instead of predicting labels), given an input prompt during inference.

A.1.3 Installing PyTorch

PyTorch can be installed just like any other Python library or package. However, since PyTorch is a comprehensive library featuring CPU- and GPU-compatible codes, the installation may require additional explanation.

> **Python version**
> Many scientific computing libraries do not immediately support the newest version of Python. Therefore, when installing PyTorch, it's advisable to use a version of Python that is one or two releases older. For instance, if the latest version of Python is 3.13, using Python 3.11 or 3.12 is recommended.

For instance, there are two versions of PyTorch: a leaner version that only supports CPU computing and a full version that supports both CPU and GPU computing. If your machine has a CUDA-compatible GPU that can be used for deep learning (ideally, an NVIDIA T4, RTX 2080 Ti, or newer), I recommend installing the GPU version. Regardless, the default command for installing PyTorch in a code terminal is:

```
pip install torch
```

Suppose your computer supports a CUDA-compatible GPU. In that case, it will automatically install the PyTorch version that supports GPU acceleration via CUDA, assuming the Python environment you're working on has the necessary dependencies (like `pip`) installed.

NOTE As of this writing, PyTorch has also added experimental support for AMD GPUs via ROCm. See https://pytorch.org for additional instructions.

To explicitly install the CUDA-compatible version of PyTorch, it's often better to specify the CUDA you want PyTorch to be compatible with. PyTorch's official website (https://pytorch.org) provides the commands to install PyTorch with CUDA support for different operating systems. Figure A.4 shows a command that will also install PyTorch, as well as the `torchvision` and `torchaudio` libraries, which are optional for this book.

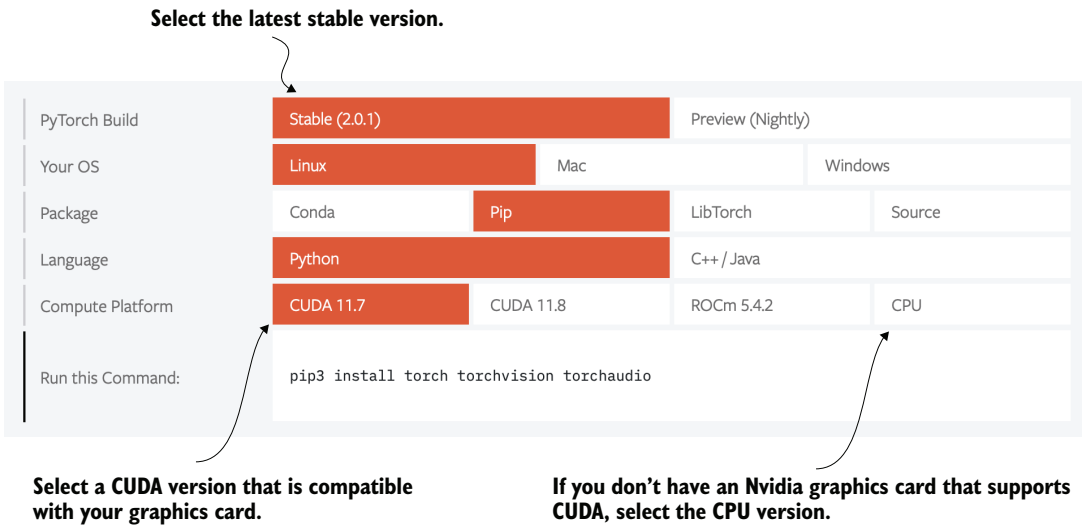


Figure A.4 Access the PyTorch installation recommendation on https://pytorch.org to customize and select the installation command for your system.

I use PyTorch 2.4.0 for the examples, so I recommend that you use the following command to install the exact version to guarantee compatibility with this book:

```
pip install torch==2.4.0
```

However, as mentioned earlier, given your operating system, the installation command might differ slightly from the one shown here. Thus, I recommend that you visit https://pytorch.org and use the installation menu (see figure A.4) to select the installation command for your operating system. Remember to replace `torch` with `torch==2.4.0` in the command.

To check the version of PyTorch, execute the following code in PyTorch:

```
import torch
torch.__version__
```

This prints

```
'2.4.0'
```

> **PyTorch and Torch**
>
> The Python library is named PyTorch primarily because it's a continuation of the Torch library but adapted for Python (hence, "PyTorch"). "Torch" acknowledges the library's roots in Torch, a scientific computing framework with wide support for machine learning algorithms, which was initially created using the Lua programming language.

If you are looking for additional recommendations and instructions for setting up your Python environment or installing the other libraries used in this book, visit the supplementary GitHub repository of this book at https://github.com/rasbt/LLMs-from-scratch.

After installing PyTorch, you can check whether your installation recognizes your built-in NVIDIA GPU by running the following code in Python:

```
import torch
torch.cuda.is_available()
```

This returns

```
True
```

If the command returns `True`, you are all set. If the command returns `False`, your computer may not have a compatible GPU, or PyTorch does not recognize it. While GPUs are not required for the initial chapters in this book, which are focused on implementing LLMs for educational purposes, they can significantly speed up deep learning–related computations.

If you don't have access to a GPU, there are several cloud computing providers where users can run GPU computations against an hourly cost. A popular Jupyter notebook–like environment is Google Colab (https://colab.research.google.com), which provides time-limited access to GPUs as of this writing. Using the Runtime menu, it is possible to select a GPU, as shown in the screenshot in figure A.5.

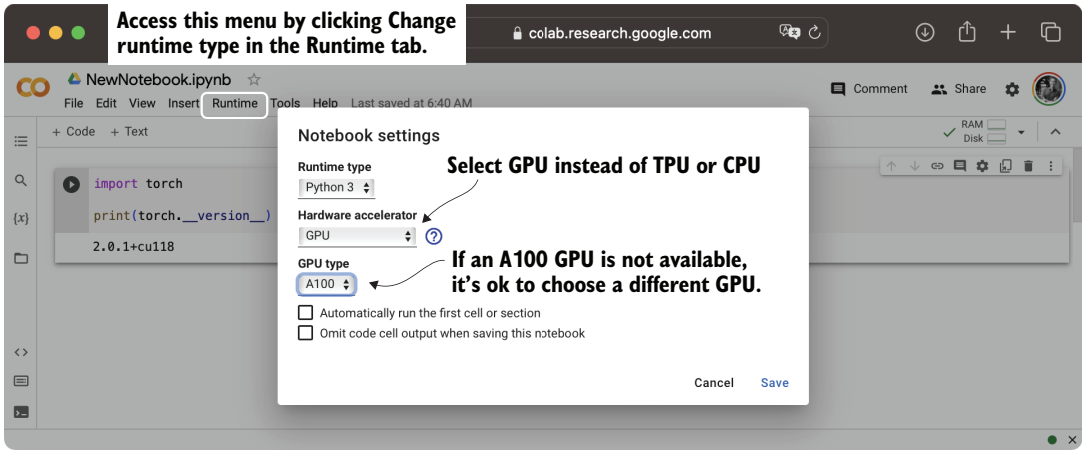


Figure A.5 Select a GPU device for Google Colab under the Runtime/Change Runtime Type menu.

PyTorch on Apple Silicon

If you have an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models), you can use its capabilities to accelerate PyTorch code execution. To use your Apple Silicon chip for PyTorch, you first need to install PyTorch as you normally would. Then, to check whether your Mac supports PyTorch acceleration with its Apple Silicon chip, you can run a simple code snippet in Python:

```
print(torch.backends.mps.is_available())
```

If it returns `True`, it means that your Mac has an Apple Silicon chip that can be used to accelerate PyTorch code.

Exercise A.1

Install and set up PyTorch on your computer

Exercise A.2

Run the supplementary code at https://mng.bz/o05v that checks whether your environment is set up correctly.

A.2 Understanding tensors

Tensors represent a mathematical concept that generalizes vectors and matrices to potentially higher dimensions. In other words, tensors are mathematical objects that can be characterized by their order (or rank), which provides the number of dimensions. For example, a scalar (just a number) is a tensor of rank 0, a vector is a tensor of rank 1, and a matrix is a tensor of rank 2, as illustrated in figure A.6.

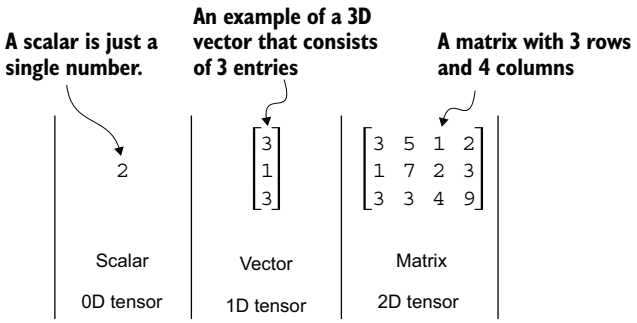


Figure A.6 Tensors with different ranks. Here 0D corresponds to rank 0, 1D to rank 1, and 2D to rank 2. A three-dimensional vector, which consists of three elements, is still a rank 1 tensor.

From a computational perspective, tensors serve as data containers. For instance, they hold multidimensional data, where each dimension represents a different feature. Tensor libraries like PyTorch can create, manipulate, and compute with these arrays efficiently. In this context, a tensor library functions as an array library.

PyTorch tensors are similar to NumPy arrays but have several additional features that are important for deep learning. For example, PyTorch adds an automatic differentiation engine, simplifying *computing gradients* (see section A.4). PyTorch tensors also support GPU computations to speed up deep neural network training (see section A.9).

> **PyTorch with a NumPy-like API**
>
> PyTorch adopts most of the NumPy array API and syntax for its tensor operations. If you are new to NumPy, you can get a brief overview of the most relevant concepts via my article "Scientific Computing in Python: Introduction to NumPy and Matplotlib" at https://sebastianraschka.com/blog/2020/numpy-intro.html.

A.2.1 Scalars, vectors, matrices, and tensors

As mentioned earlier, PyTorch tensors are data containers for array-like structures. A scalar is a zero-dimensional tensor (for instance, just a number), a vector is a one-dimensional tensor, and a matrix is a two-dimensional tensor. There is no specific term for higher-dimensional tensors, so we typically refer to a three-dimensional tensor as just a 3D tensor, and so forth. We can create objects of PyTorch's `Tensor` class using the `torch.tensor` function as shown in the following listing.

Listing A.1 Creating PyTorch tensors

```
import torch

tensor0d = torch.tensor(1)

tensor1d = torch.tensor([1, 2, 3])

tensor2d = torch.tensor([[1, 2],
                         [3, 4]])

tensor3d = torch.tensor([[[1, 2], [3, 4]],
                         [[5, 6], [7, 8]]])
```

Creates a zero-dimensional tensor (scalar) from a Python integer

Creates a one-dimensional tensor (vector) from a Python list

Creates a two-dimensional tensor from a nested Python list

Creates a three-dimensional tensor from a nested Python list

A.2.2 Tensor data types

PyTorch adopts the default 64-bit integer data type from Python. We can access the data type of a tensor via the `.dtype` attribute of a tensor:

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

This prints

```
torch.int64
```

If we create tensors from Python floats, PyTorch creates tensors with a 32-bit precision by default:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

The output is

```
torch.float32
```

This choice is primarily due to the balance between precision and computational efficiency. A 32-bit floating-point number offers sufficient precision for most deep learning tasks while consuming less memory and computational resources than a 64-bit floating-point number. Moreover, GPU architectures are optimized for 32-bit computations, and using this data type can significantly speed up model training and inference.

Moreover, it is possible to change the precision using a tensor's `.to` method. The following code demonstrates this by changing a 64-bit integer tensor into a 32-bit float tensor:

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

This returns

```
torch.float32
```

For more information about different tensor data types available in PyTorch, check the official documentation at https://pytorch.org/docs/stable/tensors.html.

A.2.3 Common PyTorch tensor operations

Comprehensive coverage of all the different PyTorch tensor operations and commands is outside the scope of this book. However, I will briefly describe relevant operations as we introduce them throughout the book.

We have already introduced the `torch.tensor()` function to create new tensors:

```
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])
print(tensor2d)
```

This prints

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

In addition, the `.shape` attribute allows us to access the shape of a tensor:

```
print(tensor2d.shape)
```

The output is

```
torch.Size([2, 3])
```

As you can see, `.shape` returns `[2, 3]`, meaning the tensor has two rows and three columns. To reshape the tensor into a 3×2 tensor, we can use the `.reshape` method:

```
print(tensor2d.reshape(3, 2))
```

This prints

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

However, note that the more common command for reshaping tensors in PyTorch is `.view()`:

```
print(tensor2d.view(3, 2))
```

The output is

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Similar to `.reshape` and `.view`, in several cases, PyTorch offers multiple syntax options for executing the same computation. PyTorch initially followed the original Lua

Torch syntax convention but then, by popular request, added syntax to make it similar to NumPy. (The subtle difference between `.view()` and `.reshape()` in PyTorch lies in their handling of memory layout: `.view()` requires the original data to be contiguous and will fail if it isn't, whereas `.reshape()` will work regardless, copying the data if necessary to ensure the desired shape.)

Next, we can use `.T` to transpose a tensor, which means flipping it across its diagonal. Note that this is not the same to reshaping a tensor, as you can see based on the following result:

```
print(tensor2d.T)
```

The output is

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

Lastly, the common way to multiply two matrices in PyTorch is the `.matmul` method:

```
print(tensor2d.matmul(tensor2d.T))
```

The output is

```
tensor([[14, 32],
        [32, 77]])
```

However, we can also adopt the `@` operator, which accomplishes the same thing more compactly:

```
print(tensor2d @ tensor2d.T)
```

This prints

```
tensor([[14, 32],
        [32, 77]])
```

As mentioned earlier, I introduce additional operations when needed. For readers who'd like to browse through all the different tensor operations available in PyTorch (we won't need most of these), I recommend checking out the official documentation at https://pytorch.org/docs/stable/tensors.html.

A.3 Seeing models as computation graphs

Now let's look at PyTorch's automatic differentiation engine, also known as autograd. PyTorch's autograd system provides functions to compute gradients in dynamic computational graphs automatically.

A computational graph is a directed graph that allows us to express and visualize mathematical expressions. In the context of deep learning, a computation graph lays

out the sequence of calculations needed to compute the output of a neural network—we will need this to compute the required gradients for backpropagation, the main training algorithm for neural networks.

Let's look at a concrete example to illustrate the concept of a computation graph. The code in the following listing implements the forward pass (prediction step) of a simple logistic regression classifier, which can be seen as a single-layer neural network. It returns a score between 0 and 1, which is compared to the true class label (0 or 1) when computing the loss.

Listing A.2 A logistic regression forward pass

```
import torch.nn.functional as F

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2])
b = torch.tensor([0.0])
z = x1 * w1 + b
a = torch.sigmoid(z)
loss = F.binary_cross_entropy(a, y)
```

This import statement is a common convention in PyTorch to prevent long lines of code.

True label

Input feature

Weight parameter

Bias unit

Net input

Activation and output

If not all components in the preceding code make sense to you, don't worry. The point of this example is not to implement a logistic regression classifier but rather to illustrate how we can think of a sequence of computations as a computation graph, as shown in figure A.7.

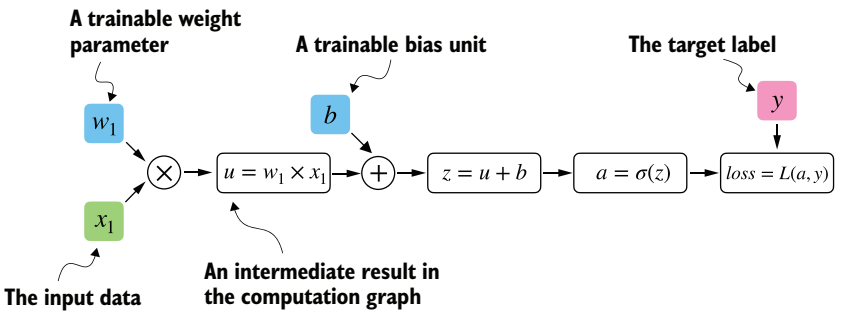


Figure A.7 A logistic regression forward pass as a computation graph. The input feature x_1 is multiplied by a model weight w_1 and passed through an activation function σ after adding the bias. The loss is computed by comparing the model output a with a given label y.

In fact, PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here w_1 and b) to train the model.

A.4 Automatic differentiation made easy

If we carry out computations in PyTorch, it will build a computational graph internally by default if one of its terminal nodes has the `requires_grad` attribute set to `True`. This is useful if we want to compute gradients. Gradients are required when training neural networks via the popular backpropagation algorithm, which can be considered an implementation of the *chain rule* from calculus for neural networks, illustrated in figure A.8.

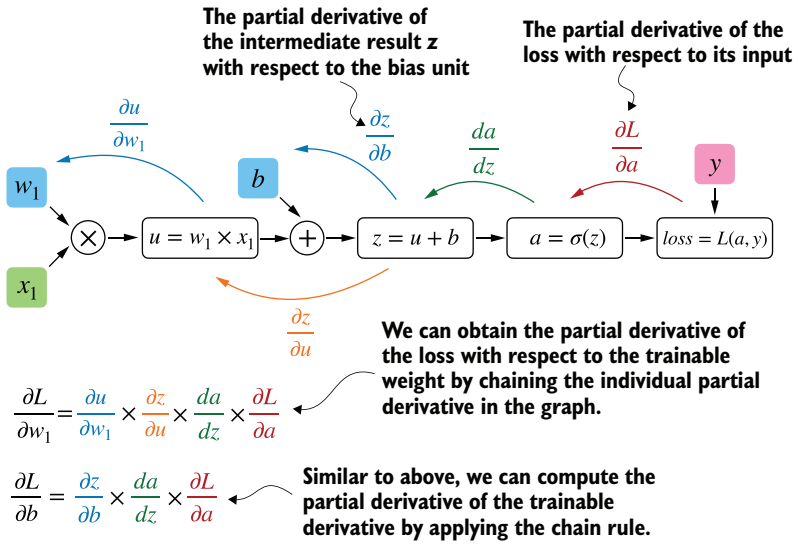


Figure A.8 The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, also called reverse-model automatic differentiation or backpropagation. We start from the output layer (or the loss itself) and work backward through the network to the input layer. We do this to compute the gradient of the loss with respect to each parameter (weights and biases) in the network, which informs how we update these parameters during training.

Partial derivatives and gradients

Figure A.8 shows partial derivatives, which measure the rate at which a function changes with respect to one of its variables. A *gradient* is a vector containing all of the partial derivatives of a multivariate function, a function with more than one variable as input.

If you are not familiar with or don't remember the partial derivatives, gradients, or chain rule from calculus, don't worry. On a high level, all you need to know for this book is that the chain rule is a way to compute gradients of a loss function given the model's parameters in a computation graph. This provides the information needed to update each parameter to minimize the loss function, which serves as a proxy for measuring the

model's performance using a method such as gradient descent. We will revisit the computational implementation of this training loop in PyTorch in section A.7.

How is this all related to the automatic differentiation (autograd) engine, the second component of the PyTorch library mentioned earlier? PyTorch's autograd engine constructs a computational graph in the background by tracking every operation performed on tensors. Then, calling the `grad` function, we can compute the gradient of the loss concerning the model parameter `w1`, as shown in the following listing.

Listing A.3 Computing gradients via autograd

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

By default, PyTorch destroys the computation graph after calculating the gradients to free memory. However, since we will reuse this computation graph shortly, we set retain_graph=True so that it stays in memory.

The resulting values of the loss gradients given the model's parameters are

```
print(grad_L_w1)
print(grad_L_b)
```

This prints

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Here, we have been using the grad function manually, which can be useful for experimentation, debugging, and demonstrating concepts. But, in practice, PyTorch provides even more high-level tools to automate this process. For instance, we can call `.backward` on the loss, and PyTorch will compute the gradients of all the leaf nodes in the graph, which will be stored via the tensors' `.grad` attributes:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

The outputs are

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

I've provided you with a lot of information, and you may be overwhelmed by the calculus concepts, but don't worry. While this calculus jargon is a means to explain PyTorch's autograd component, all you need to take away is that PyTorch takes care of the calculus for us via the `.backward` method—we won't need to compute any derivatives or gradients by hand.

A.5 Implementing multilayer neural networks

Next, we focus on PyTorch as a library for implementing deep neural networks. To provide a concrete example, let's look at a multilayer perceptron, a fully connected neural network, as illustrated in figure A.9.

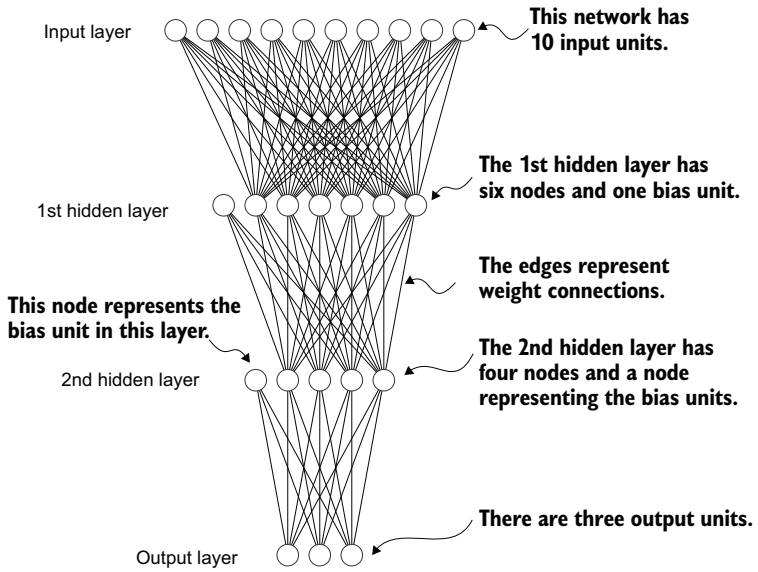


Figure A.9 A multilayer perceptron with two hidden layers. Each node represents a unit in the respective layer. For illustration purposes, each layer has a very small number of nodes.

When implementing a neural network in PyTorch, we can subclass the `torch.nn.Module` class to define our own custom network architecture. This `Module` base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.

Within this subclass, we define the network layers in the `__init__` constructor and specify how the layers interact in the forward method. The forward method describes how the input data passes through the network and comes together as a computation graph. In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function given the model parameters (see section A.7). The code in the following listing implements a

classic multilayer perceptron with two hidden layers to illustrate a typical usage of the `Module` class.

Listing A.4 A multilayer perceptron with two hidden layers

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),

            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

Coding the number of inputs and outputs as variables allows us to reuse the same code for datasets with different numbers of features and classes

The Linear layer takes the number of input and output nodes as arguments.

Nonlinear activation functions are placed between the hidden layers.

The number of output nodes of one hidden layer has to match the number of inputs of the next layer.

The outputs of the last layer are called logits.

We can then instantiate a new neural network object as follows:

```
model = NeuralNetwork(50, 3)
```

Before using this new `model` object, we can call `print` on the model to see a summary of its structure:

```
print(model)
```

This prints

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Note that we use the `Sequential` class when we implement the `NeuralNetwork` class. `Sequential` is not required, but it can make our life easier if we have a series of layers we want to execute in a specific order, as is the case here. This way, after instantiating `self.layers = Sequential(...)` in the `__init__` constructor, we just have to

call the `self.layers` instead of calling each layer individually in the `NeuralNetwork`'s `forward` method.

Next, let's check the total number of trainable parameters of this model:

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

This prints

```
Total number of trainable model parameters: 2213
```

Each parameter for which `requires_grad=True` counts as a trainable parameter and will be updated during training (see section A.7).

In the case of our neural network model with the preceding two hidden layers, these trainable parameters are contained in the `torch.nn.Linear` layers. A `Linear` layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes referred to as a *feedforward* or *fully connected* layer.

Based on the `print(model)` call we executed here, we can see that the first `Linear` layer is at index position 0 in the layers attribute. We can access the corresponding weight parameter matrix as follows:

```
print(model.layers[0].weight)
```

This prints

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227,  ...,  0.0275, -0.0520, -0.0192],
        [-0.0169,  0.1265,  0.0255,  ..., -0.1247,  0.1191, -0.0698],
        [-0.0973, -0.0974, -0.0739,  ..., -0.0068, -0.0892,  0.1070],
        ...,
        [-0.0681,  0.1058, -0.0315,  ..., -0.1081, -0.0290, -0.1374],
        [-0.0159,  0.0587, -0.0916,  ..., -0.1153,  0.0700,  0.0770],
        [-0.1019,  0.1345, -0.0176,  ...,  0.0114, -0.0559, -0.0088]],
       requires_grad=True)
```

Since this large matrix is not shown in its entirety, let's use the `.shape` attribute to show its dimensions:

```
print(model.layers[0].weight.shape)
```

The result is

```
torch.Size([30, 50])
```

(Similarly, you could access the bias vector via `model.layers[0].bias`.)

The weight matrix here is a 30 × 50 matrix, and we can see that `requires_grad` is set to `True`, which means its entries are trainable—this is the default setting for weights and biases in `torch.nn.Linear`.

If you execute the preceding code on your computer, the numbers in the weight matrix will likely differ from those shown. The model weights are initialized with small random numbers, which differ each time we instantiate the network. In deep learning, initializing model weights with small random numbers is desired to break symmetry during training. Otherwise, the nodes would be performing the same operations and updates during backpropagation, which would not allow the network to learn complex mappings from inputs to outputs.

However, while we want to keep using small random numbers as initial values for our layer weights, we can make the random number initialization reproducible by seeding PyTorch's random number generator via `manual_seed`:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

The result is

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True)
```

Now that we have spent some time inspecting the `NeuralNetwork` instance, let's briefly see how it's used via the forward pass:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

The result is

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

In the preceding code, we generated a single random training example `X` as a toy input (note that our network expects 50-dimensional feature vectors) and fed it to the model, returning three scores. When we call `model(x)`, it will automatically execute the forward pass of the model.

The forward pass refers to calculating output tensors from input tensors. This involves passing the input data through all the neural network layers, starting from the input layer, through hidden layers, and finally to the output layer.

These three numbers returned here correspond to a score assigned to each of the three output nodes. Notice that the output tensor also includes a `grad_fn` value.

Here, `grad_fn=<AddmmBackward0>` represents the last-used function to compute a variable in the computational graph. In particular, `grad_fn=<AddmmBackward0>` means that the tensor we are inspecting was created via a matrix multiplication and addition operation. PyTorch will use this information when it computes gradients during backpropagation. The `<AddmmBackward0>` part of `grad_fn=<AddmmBackward0>` specifies the operation performed. In this case, it is an `Addmm` operation. `Addmm` stands for matrix multiplication (`mm`) followed by an addition (`Add`).

If we just want to use a network without training or backpropagation—for example, if we use it for prediction after training—constructing this computational graph for backpropagation can be wasteful as it performs unnecessary computations and consumes additional memory. So, when we use a model for inference (for instance, making predictions) rather than training, the best practice is to use the `torch.no_grad()` context manager. This tells PyTorch that it doesn't need to keep track of the gradients, which can result in significant savings in memory and computation:

```
with torch.no_grad():
    out = model(X)
print(out)
```

The result is

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

In PyTorch, it's common practice to code models such that they return the outputs of the last layer (logits) without passing them to a nonlinear activation function. That's because PyTorch's commonly used loss functions combine the `softmax` (or `sigmoid` for binary classification) operation with the negative log-likelihood loss in a single class. The reason for this is numerical efficiency and stability. So, if we want to compute class-membership probabilities for our predictions, we have to call the `softmax` function explicitly:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

This prints

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

The values can now be interpreted as class-membership probabilities that sum up to 1. The values are roughly equal for this random input, which is expected for a randomly initialized model without training.

A.6 Setting up efficient data loaders

Before we can train our model, we have to briefly discuss creating efficient data loaders in PyTorch, which we will iterate over during training. The overall idea behind data loading in PyTorch is illustrated in figure A.10.

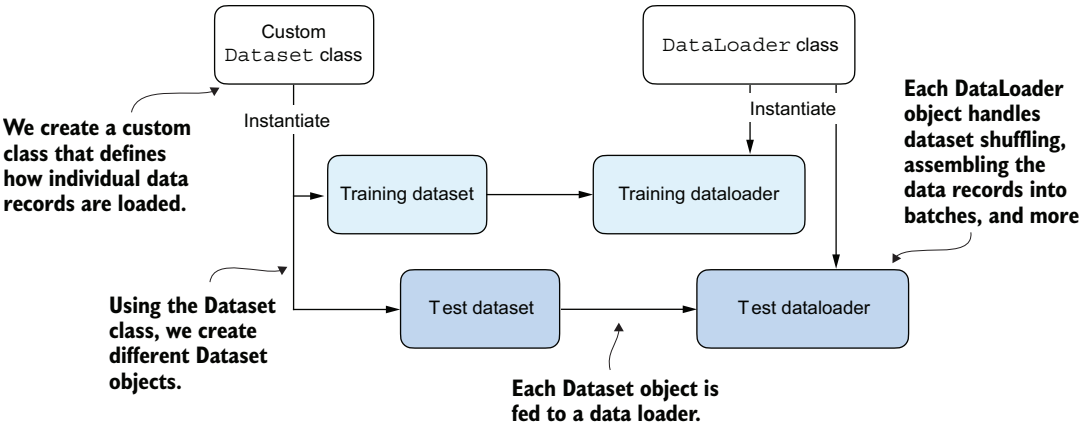


Figure A.10 PyTorch implements a `Dataset` and a `DataLoader` class. The `Dataset` class is used to instantiate objects that define how each data record is loaded. The `DataLoader` handles how the data is shuffled and assembled into batches.

Following figure A.10, we will implement a custom `Dataset` class, which we will use to create a training and a test dataset that we'll then use to create the data loaders. Let's start by creating a simple toy dataset of five training examples with two features each. Accompanying the training examples, we also create a tensor containing the corresponding class labels: three examples belong to class 0, and two examples belong to class 1. In addition, we make a test set consisting of two entries. The code to create this dataset is shown in the following listing.

Listing A.5 Creating a small toy dataset

```
X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
    [2.7, -1.5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])
y_test = torch.tensor([0, 1])
```

NOTE PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at zero). So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of five nodes.

Next, we create a custom dataset class, `ToyDataset`, by subclassing from PyTorch's `Dataset` parent class, as shown in the following listing.

Listing A.6 Defining a custom `Dataset` class

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

Instructions for retrieving exactly one data record and the corresponding label

Instructions for returning the total length of the dataset

The purpose of this custom `ToyDataset` class is to instantiate a PyTorch `DataLoader`. But before we get to this step, let's briefly go over the general structure of the `ToyDataset` code.

In PyTorch, the three main components of a custom `Dataset` class are the `__init__` constructor, the `__getitem__` method, and the `__len__` method (see listing A.6). In the `__init__` method, we set up attributes that we can access later in the `__getitem__` and `__len__` methods. These could be file paths, file objects, database connectors, and so on. Since we created a tensor dataset that sits in memory, we simply assign `X` and `y` to these attributes, which are placeholders for our tensor objects.

In the `__getitem__` method, we define instructions for returning exactly one item from the dataset via an `index`. This refers to the features and the class label corresponding to a single training example or test instance. (The data loader will provide this `index`, which we will cover shortly.)

Finally, the `__len__` method contains instructions for retrieving the length of the dataset. Here, we use the `.shape` attribute of a tensor to return the number of rows in the feature array. In the case of the training dataset, we have five rows, which we can double-check:

```
print(len(train_ds))
```

The result is

```
5
```

Now that we've defined a PyTorch `Dataset` class we can use for our toy dataset, we can use PyTorch's `DataLoader` class to sample from it, as shown in the following listing.

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader


torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

The ToyDataset instance created earlier serves as input to the data loader.

Whether or not to shuffle the data

The number of background processes

It is not necessary to shuffle a test dataset.

After instantiating the training data loader, we can iterate over it. The iteration over the `test_loader` works similarly but is omitted for brevity:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is

```
Batch 1: tensor([[-1.2000,  3.1000],
                 [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                 [-0.9000,  2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

As we can see based on the preceding output, the `train_loader` iterates over the training dataset, visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using `torch.manual_seed(123)` here, you should get the exact same shuffling order of training examples. However, if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks from getting caught in repetitive update cycles during training.

We specified a batch size of 2 here, but the third batch only contains a single example. That's because we have five training examples, and 5 is not evenly divisible by 2.

In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, set `drop_last=True`, which will drop the last batch in each epoch, as shown in the following listing.

Listing A.8 A training loader that drops the last batch

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

Now, iterating over the training loader, we can see that the last batch is omitted:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is

```
Batch 1: tensor([[-0.9000,  2.9000],
        [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
        [-0.5000,  2.6000]]) tensor([1, 0])
```

Lastly, let's discuss the setting `num_workers=0` in the `DataLoader`. This parameter in PyTorch's `DataLoader` function is crucial for parallelizing data loading and preprocessing. When `num_workers` is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. Instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when `num_workers` is set to a number greater than 0, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources (figure A.11).

However, if we are working with very small datasets, setting `num_workers` to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. So, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing `num_workers` may not provide any noticeable speedup. It may, in fact, lead to some problems. One potential problem is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

Furthermore, for Jupyter notebooks, setting `num_workers` to greater than 0 can sometimes lead to problems related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand

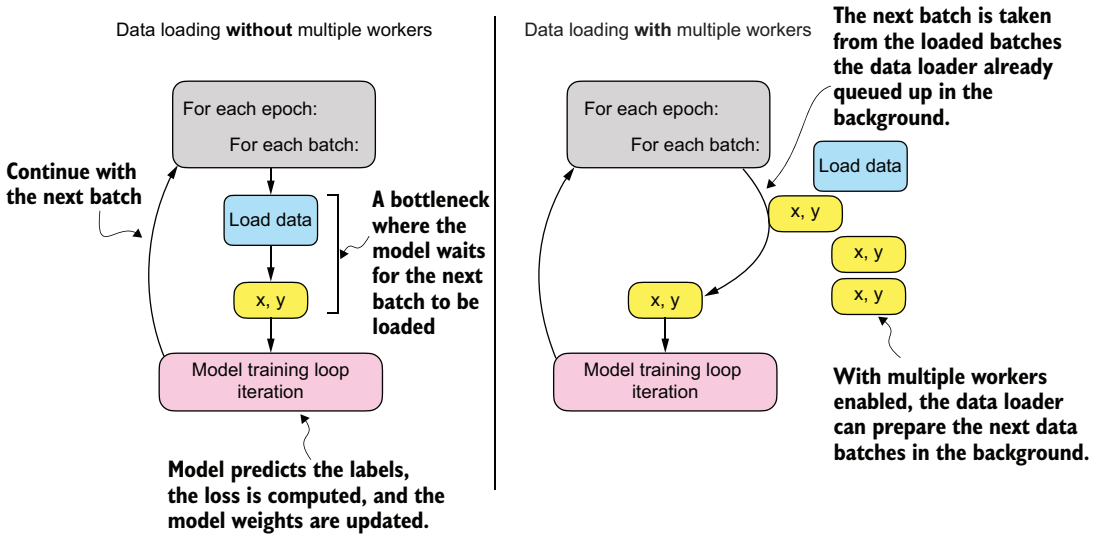


Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded (left). If multiple workers are enabled, the data loader can queue up the next batch in the background (right).

the tradeoff and make a calculated decision on setting the `num_workers` parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

In my experience, setting `num_workers=4` usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the `Dataset` class.

A.7 A typical training loop

Let's now train a neural network on the toy dataset. The following listing shows the training code.

Listing A.9 Neural network training in PyTorch

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(
    model.parameters(), lr=0.5
)

num_epochs = 3
for epoch in range(num_epochs):

    model.train()
```

The dataset has two features and two classes.

The optimizer needs to know which parameters to optimize.

```
for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
          f" | Train Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code
```

Sets the gradients from the previous round to 0 to prevent unintended gradient accumulation

Computes the gradients of the loss given the model parameters

The optimizer uses the gradients to update the model parameters.

Running this code yields the following outputs:

```
Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Trainl Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00
```

As we can see, the loss reaches 0 after three epochs, a sign that the model converged on the training set. Here, we initialize a model with two inputs and two outputs because our toy dataset has two input features and two class labels to predict. We used a stochastic gradient descent (`SGD`) optimizer with a learning rate (`lr`) of 0.5. The learning rate is a hyperparameter, meaning it's a tunable setting that we must experiment with based on observing the loss. Ideally, we want to choose a learning rate such that the loss converges after a certain number of epochs—the number of epochs is another hyperparameter to choose.

Exercise A.3

How many parameters does the neural network introduced in listing A.9 have?

In practice, we often use a third dataset, a so-called validation dataset, to find the optimal hyperparameter settings. A validation dataset is similar to a test set. However, while we only want to use a test set precisely once to avoid biasing the evaluation, we usually use the validation set multiple times to tweak the model settings.

We also introduced new settings called `model.train()` and `model.eval()`. As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training and inference, such as *dropout* or *batch normalization* layers. Since we don't have dropout

or other components in our `NeuralNetwork` class that are affected by these settings, using `model.train()` and `model.eval()` is redundant in our preceding code. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

As discussed earlier, we pass the logits directly into the `cross_entropy` loss function, which will apply the `softmax` function internally for efficiency and numerical stability reasons. Then, calling `loss.backward()` will calculate the gradients in the computation graph that PyTorch constructed in the background. The `optimizer.step()` method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

NOTE To prevent undesired gradient accumulation, it is important to include an `optimizer.zero_grad()` call in each update round to reset the gradients to 0. Otherwise, the gradients will accumulate, which may be undesired.

After we have trained the model, we can use it to make predictions:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

The results are

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

To obtain the class membership probabilities, we can then use PyTorch's `softmax` function:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

This outputs

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
```

Let's consider the first row in the preceding code output. Here, the first value (column) means that the training example has a 99.91% probability of belonging to class

0 and a 0.09% probability of belonging to class 1. (The `set_printoptions` call is used here to make the outputs more legible.)

We can convert these values into class label predictions using PyTorch's `argmax` function, which returns the index position of the highest value in each row if we set `dim=1` (setting `dim=0` would return the highest value in each column instead):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

This prints

```
tensor([0, 0, 0, 1, 1])
```

Note that it is unnecessary to compute `softmax` probabilities to obtain the class labels. We could also apply the `argmax` function to the logits (outputs) directly:

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

The output is

```
tensor([0, 0, 0, 1, 1])
```

Here, we computed the predicted labels for the training dataset. Since the training dataset is relatively small, we could compare it to the true training labels by eye and see that the model is 100% correct. We can double-check this using the `==` comparison operator:

```
predictions == y_train
```

The results are

```
tensor([True, True, True, True, True])
```

Using `torch.sum`, we can count the number of correct predictions:

```
torch.sum(predictions == y_train)
```

The output is

```
5
```

Since the dataset consists of five training examples, we have five out of five predictions that are correct, which has $5/5 \times 100\% = 100\%$ prediction accuracy.

To generalize the computation of the prediction accuracy, let's implement a `compute_accuracy` function, as shown in the following listing.

Listing A.10 A function to compute the prediction accuracy

```
def compute_accuracy(model, dataloader):

    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):

        with torch.no_grad():
            logits = model(features)

        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions
        correct += torch.sum(compare)
        total_examples += len(compare)

    return (correct / total_examples).item()
```

Returns a tensor of True/False values depending on whether the labels match

The sum operation counts the number of True values.

The fraction of correct prediction, a value between 0 and 1. .item() returns the value of the tensor as a Python float.

The code iterates over a data loader to compute the number and fraction of the correct predictions. When we work with large datasets, we typically can only call the model on a small part of the dataset due to memory limitations. The `compute_accuracy` function here is a general method that scales to datasets of arbitrary size since, in each iteration, the dataset chunk that the model receives is the same size as the batch size seen during training. The internals of the `compute_accuracy` function are similar to what we used before when we converted the logits to the class labels.

We can then apply the function to the training:

```
print(compute_accuracy(model, train_loader))
```

The result is

```
1.0
```

Similarly, we can apply the function to the test set:

```
print(compute_accuracy(model, test_loader))
```

This prints

```
1.0
```

A.8 Saving and loading models

Now that we've trained our model, let's see how to save it so we can reuse it later. Here's the recommended way of saving and loading models in PyTorch:

```
torch.save(model.state_dict(), "model.pth")
```

The model's `state_dict` is a Python dictionary object that maps each layer in the model to its trainable parameters (weights and biases). `"model.pth"` is an arbitrary filename for the model file saved to disk. We can give it any name and file ending we like; however, `.pth` and `.pt` are the most common conventions.

Once we saved the model, we can restore it from disk:

```
model = NeuralNetwork(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

The `torch.load("model.pth")` function reads the file `"model.pth"` and reconstructs the Python dictionary object containing the model's parameters while `model.load_state_dict()` applies these parameters to the model, effectively restoring its learned state from when we saved it.

The line `model = NeuralNetwork(2, 2)` is not strictly necessary if you execute this code in the same session where you saved a model. However, I included it here to illustrate that we need an instance of the model in memory to apply the saved parameters. Here, the `NeuralNetwork(2, 2)` architecture needs to match the original saved model exactly.

A.9 Optimizing training performance with GPUs

Next, let's examine how to utilize GPUs, which accelerate deep neural network training compared to regular CPUs. First, we'll look at the main concepts behind GPU computing in PyTorch. Then we will train a model on a single GPU. Finally, we'll look at distributed training using multiple GPUs.

A.9.1 PyTorch computations on GPU devices

Modifying the training loop to run optionally on a GPU is relatively simple and only requires changing three lines of code (see section A.7). Before we make the modifications, it's crucial to understand the main concept behind GPU computations within PyTorch. In PyTorch, a device is where computations occur and data resides. The CPU and the GPU are examples of devices. A PyTorch tensor resides in a device, and its operations are executed on the same device.

Let's see how this works in action. Assuming that you installed a GPU-compatible version of PyTorch (see section A.1.3), we can double-check that our runtime indeed supports GPU computing via the following code:

```
print(torch.cuda.is_available())
```

The result is

```
True
```

Now, suppose we have two tensors that we can add; this computation will be carried out on the CPU by default:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

This outputs

```
tensor([5., 7., 9.])
```

We can now use the `.to()` method. This method is the same as the one we use to change a tensor's datatype (see 2.2.2) to transfer these tensors onto a GPU and perform the addition there:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

The output is

```
tensor([5., 7., 9.], device='cuda:0')
```

The resulting tensor now includes the device information, `device='cuda:0'`, which means that the tensors reside on the first GPU. If your machine hosts multiple GPUs, you can specify which GPU you'd like to transfer the tensors to. You do so by indicating the device ID in the transfer command. For instance, you can use `.to("cuda:0")`, `.to("cuda:1")`, and so on.

However, all tensors must be on the same device. Otherwise, the computation will fail, where one tensor resides on the CPU and the other on the GPU:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

The results are

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at
least two devices, cuda:0 and cpu!
```

In sum, we only need to transfer the tensors onto the same GPU device, and PyTorch will handle the rest.

A.9.2 Single-GPU training

Now that we are familiar with transferring tensors to the GPU, we can modify the training loop to run on a GPU. This step requires only changing three lines of code, as shown in the following listing.

Listing A.11 A training loop on a GPU

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")
model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Insert optional model evaluation code
```

Defines a device variable that defaults to a GPU

Transfers the model onto the GPU

Transfers the data onto the GPU

Running the preceding code will output the following, similar to the results obtained on the CPU (section A.7):

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00
```

We can use `.to("cuda")` instead of `device = torch.device("cuda")`. Transferring a tensor to `"cuda"` instead of `torch.device("cuda")` works as well and is shorter (see section A.9.1). We can also modify the statement, which will make the same code executable on a CPU if a GPU is not available. This is considered best practice when sharing PyTorch code:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In the case of the modified training loop here, we probably won't see a speedup due to the memory transfer cost from CPU to GPU. However, we can expect a significant speedup when training deep neural networks, especially LLMs.

PyTorch on macOS

On an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models) instead of a computer with an Nvidia GPU, you can change

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

to

```
device = torch.device(
    "mps" if torch.backends.mps.is_available() else "cpu"
)
```

to take advantage of this chip.

Exercise A.4

Compare the run time of matrix multiplication on a CPU to a GPU. At what matrix size do you begin to see the matrix multiplication on the GPU being faster than on the CPU? Hint: use the `%timeit` command in Jupyter to compare the run time. For example, given matrices `a` and `b`, run the command `%timeit a @ b` in a new notebook cell.

A.9.3 Training with multiple GPUs

Distributed training is the concept of dividing the model training across multiple GPUs and machines. Why do we need this? Even when it is possible to train a model on a single GPU or machine, the process could be exceedingly time-consuming. The training time can be significantly reduced by distributing the training process across multiple machines, each with potentially multiple GPUs. This is particularly crucial in the experimental stages of model development, where numerous training iterations might be necessary to fine-tune the model parameters and architecture.

> **NOTE** For this book, access to or use of multiple GPUs is not required. This section is included for those interested in how multi-GPU computing works in PyTorch.

Let's begin with the most basic case of distributed training: PyTorch's `DistributedDataParallel` (DDP) strategy. DDP enables parallelism by splitting the input data across the available devices and processing these data subsets simultaneously.

How does this work? PyTorch launches a separate process on each GPU, and each process receives and keeps a copy of the model; these copies will be synchronized during training. To illustrate this, suppose we have two GPUs that we want to use to train a neural network, as shown in figure A.12.

Each of the two GPUs will receive a copy of the model. Then, in every training iteration, each model will receive a minibatch (or just "batch") from the data loader. We

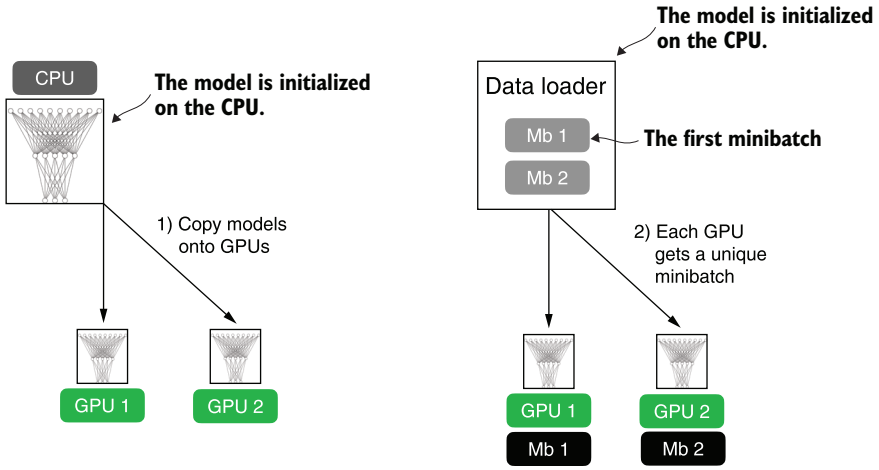


Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.

can use a `DistributedSampler` to ensure that each GPU will receive a different, non-overlapping batch when using DDP.

Since each model copy will see a different sample of the training data, the model copies will return different logits as outputs and compute different gradients during the backward pass. These gradients are then averaged and synchronized during training to update the models. This way, we ensure that the models don't diverge, as illustrated in figure A.13.

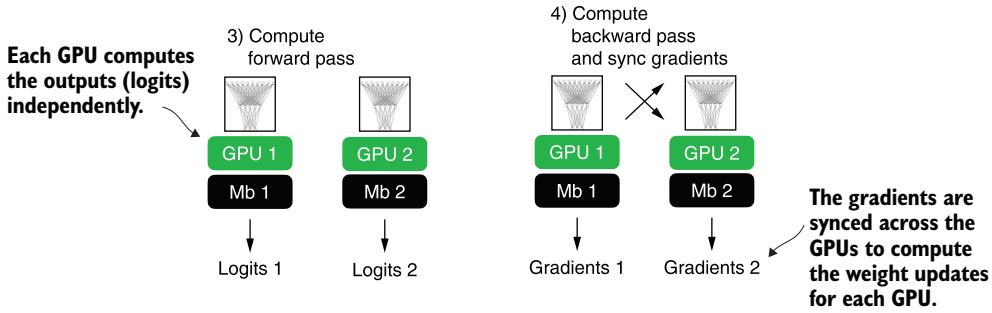


Figure A.13 The forward and backward passes in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has the same updated weights.

The benefit of using DDP is the enhanced speed it offers for processing the dataset compared to a single GPU. Barring a minor communication overhead between devices that

comes with DDP use, it can theoretically process a training epoch in half the time with two GPUs compared to just one. The time efficiency scales up with the number of GPUs, allowing us to process an epoch eight times faster if we have eight GPUs, and so on.

> **NOTE** DDP does not function properly within interactive Python environments like Jupyter notebooks, which don't handle multiprocessing in the same way a standalone Python script does. Therefore, the following code should be executed as a script, not within a notebook interface like Jupyter. DDP needs to spawn multiple processes, and each process should have its own Python interpreter instance.

Let's now see how this works in practice. For brevity, I focus on the core parts of the code that need to be adjusted for DDP training. However, readers who want to run the code on their own multi-GPU machine or a cloud instance of their choice should use the standalone script provided in this book's GitHub repository at https://github.com/rasbt/LLMs-from-scratch.

First, we import a few additional submodules, classes, and functions for distributed training PyTorch, as shown in the following listing.

Listing A.12 PyTorch utilities for distributed training

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

Before we dive deeper into the changes to make the training compatible with DDP, let's briefly go over the rationale and usage for these newly imported utilities that we need alongside the `DistributedDataParallel` class.

PyTorch's `multiprocessing` submodule contains functions such as `multiprocessing.spawn`, which we will use to spawn multiple processes and apply a function to multiple inputs in parallel. We will use it to spawn one training process per GPU. If we spawn multiple processes for training, we will need a way to divide the dataset among these different processes. For this, we will use the `DistributedSampler`.

`init_process_group` and `destroy_process_group` are used to initialize and quit the distributed training mods. The `init_process_group` function should be called at the beginning of the training script to initialize a process group for each process in the distributed setup, and `destroy_process_group` should be called at the end of the training script to destroy a given process group and release its resources. The code in the following listing illustrates how these new components are used to implement DDP training for the `NeuralNetwork` model we implemented earlier.

Listing A.13 Model training with the `DistributedDataParallel` strategy

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"
```

Address of the main node

```
    os.environ["MASTER_PORT"] = "12345"
    init_process_group(
        backend="nccl",
        rank=rank,
        world_size=world_size
    )
    torch.cuda.set_device(rank)

def prepare_dataset():
    # insert dataset preparation code
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,
        pin_memory=True,
        drop_last=True,
        sampler=DistributedSampler(train_ds)
    )
    return train_loader, test_loader

def main(rank, world_size, num_epochs):
    ddp_setup(rank, world_size)
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
        train_loader.sampler.set_epoch(epoch)
        model.train()
        for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            # insert model prediction and backpropagation code
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank)
    print(f"[GPU{rank}] Training accuracy", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Test accuracy", test_acc)
    destroy_process_group()

if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)
```

world_size is the number of GPUs to use.

Any free port on the machine

nccl stands for NVIDIA Collective Communication Library.

rank refers to the index of the GPU we want to use.

Sets the current GPU device on which tensors will be allocated and operations will be performed

DistributedSampler takes care of the shuffling now.

Enables faster memory transfer when training on GPU

Splits the dataset into distinct, non-overlapping subsets for each process (GPU)

The main function running the model training

rank is the GPU ID

Cleans up resource allocation

Launches the main function using multiple processes, where nprocs=world_size means one process per GPU.

Before we run this code, let's summarize how it works in addition to the preceding annotations. We have a `__name__ == "__main__"` clause at the bottom containing code executed when we run the code as a Python script instead of importing it as a module. This code first prints the number of available GPUs using `torch.cuda.device_count()`, sets a random seed for reproducibility, and then spawns new processes using PyTorch's `multiprocessesing.spawn` function. Here, the `spawn` function launches one process per GPU setting `nproces=world_size`, where the world size is the number of available GPUs. This `spawn` function launches the code in the `main` function we define in the same script with some additional arguments provided via `args`. Note that the `main` function has a `rank` argument that we don't include in the `mp.spawn()` call. That's because the `rank`, which refers to the process ID we use as the GPU ID, is already passed automatically.

The `main` function sets up the distributed environment via `ddp_setup`—another function we defined—loads the training and test sets, sets up the model, and carries out the training. Compared to the single-GPU training (section A.9.2), we now transfer the model and data to the target device via `.to(rank)`, which we use to refer to the GPU device ID. Also, we wrap the model via `DDP`, which enables the synchronization of the gradients between the different GPUs during training. After the training finishes and we evaluate the models, we use `destroy_process_group()` to cleanly exit the distributed training and free up the allocated resources.

Earlier I mentioned that each GPU will receive a different subsample of the training data. To ensure this, we set `sampler=DistributedSampler(train_ds)` in the training loader.

The last function to discuss is `ddp_setup`. It sets the main node's address and port to allow for communication between the different processes, initializes the process group with the NCCL backend (designed for GPU-to-GPU communication), and sets the `rank` (process identifier) and world size (total number of processes). Finally, it specifies the GPU device corresponding to the current model training process rank.

SELECTING AVAILABLE GPUS ON A MULTI-GPU MACHINE

If you wish to restrict the number of GPUs used for training on a multi-GPU machine, the simplest way is to use the `CUDA_VISIBLE_DEVICES` environment variable. To illustrate this, suppose your machine has multiple GPUs, and you only want to use one GPU—for example, the GPU with index 0. Instead of `python some_script.py`, you can run the following code from the terminal:

```
CUDA_VISIBLE_DEVICES=0 python some_script.py
```

Or, if your machine has four GPUs and you only want to use the first and third GPU, you can use

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

Setting `CUDA_VISIBLE_DEVICES` in this way is a simple and effective way to manage GPU allocation without modifying your PyTorch scripts.

Let's now run this code and see how it works in practice by launching the code as a script from the terminal:

```
python ch02-DDP-script.py
```

Note that it should work on both single and multi-GPU machines. If we run this code on a single GPU, we should see the following output:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

The code output looks similar to that using a single GPU (section A.9.2), which is a good sanity check.

Now, if we run the same command and code on a machine with two GPUs, we should see the following:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

As expected, we can see that some batches are processed on the first GPU (`GPU0`) and others on the second (`GPU1`). However, we see duplicated output lines when printing the training and test accuracies. Each process (in other words, each GPU) prints the test accuracy independently. Since DDP replicates the model onto each GPU and each process runs independently, if you have a print statement inside your testing loop, each process will execute it, leading to repeated output lines. If this bothers you, you can fix it using the rank of each process to control your print statements:

```
if rank == 0:                                  <-- Only print in the first process
    print("Test accuracy: ", accuracy)
```

This is, in a nutshell, how distributed training via DDP works. If you are interested in additional details, I recommend checking the official API documentation at https://mng.bz/9dPr.

> **Alternative PyTorch APIs for multi-GPU training**
> If you prefer a more straightforward way to use multiple GPUs in PyTorch, you can consider add-on APIs like the open-source Fabric library. I wrote about it in "Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism" (https://mng.bz/jXle).

Summary

- PyTorch is an open source library with three core components: a tensor library, automatic differentiation functions, and deep learning utilities.
- PyTorch's tensor library is similar to array libraries like NumPy.
- In the context of PyTorch, tensors are array-like data structures representing scalars, vectors, matrices, and higher-dimensional arrays.
- PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch's tensor format is its GPU support to accelerate computations.
- The automatic differentiation (autograd) capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes `Dataset` and `DataLoader` classes to set up efficient data-loading pipelines.
- It's easiest to train models on a CPU or single GPU.
- Using `DistributedDataParallel` is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.

appendix B References and further reading

Chapter 1

Custom-built LLMs are able to outperform general-purpose LLMs as a team at Bloomberg showed via a version of GPT pretrained on finance data from scratch. The custom LLM outperformed ChatGPT on financial tasks while maintaining good performance on general LLM benchmarks:

- "BloombergGPT: A Large Language Model for Finance" (2023) by Wu et al., https://arxiv.org/abs/2303.17564

Existing LLMs can be adapted and fine-tuned to outperform general LLMs as well, which teams from Google Research and Google DeepMind showed in a medical context:

- "Towards Expert-Level Medical Question Answering with Large Language Models" (2023) by Singhal et al., https://arxiv.org/abs/2305.09617

The following paper proposed the original transformer architecture:

- "Attention Is All You Need" (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762

On the original encoder-style transformer, called BERT, see

- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al., https://arxiv.org/abs/1810.04805

The paper describing the decoder-style GPT-3 model, which inspired modern LLMs and will be used as a template for implementing an LLM from scratch in this book, is

- "Language Models are Few-Shot Learners" (2020) by Brown et al., https://arxiv.org/abs/2005.14165

The following covers the original vision transformer for classifying images, which illustrates that transformer architectures are not only restricted to text inputs:

- "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (2020) by Dosovitskiy et al., https://arxiv.org/abs/2010.11929

The following experimental (but less popular) LLM architectures serve as examples that not all LLMs need to be based on the transformer architecture:

- "RWKV: Reinventing RNNs for the Transformer Era" (2023) by Peng et al., https://arxiv.org/abs/2305.13048
- "Hyena Hierarchy: Towards Larger Convolutional Language Models" (2023) by Poli et al., https://arxiv.org/abs/2302.10866
- "Mamba: Linear-Time Sequence Modeling with Selective State Spaces" (2023) by Gu and Dao, https://arxiv.org/abs/2312.00752

Meta AI's model is a popular implementation of a GPT-like model that is openly available in contrast to GPT-3 and ChatGPT:

- "Llama 2: Open Foundation and Fine-Tuned Chat Models" (2023) by Touvron et al., https://arxiv.org/abs/2307.092881

For readers interested in additional details about the dataset references in section 1.5, this paper describes the publicly available *The Pile* dataset curated by Eleuther AI:

- "The Pile: An 800GB Dataset of Diverse Text for Language Modeling" (2020) by Gao et al., https://arxiv.org/abs/2101.00027

The following paper provides the reference for InstructGPT for fine-tuning GPT-3, which was mentioned in section 1.6 and will be discussed in more detail in chapter 7:

- "Training Language Models to Follow Instructions with Human Feedback" (2022) by Ouyang et al., https://arxiv.org/abs/2203.02155

Chapter 2

Readers who are interested in discussion and comparison of embedding spaces with latent spaces and the general notion of vector representations can find more information in the first chapter of my book:

- *Machine Learning Q and AI* (2023) by Sebastian Raschka, https://leanpub.com/machine-learning-q-and-ai

The following paper provides more in-depth discussions of how byte pair encoding is used as a tokenization method:

- "Neural Machine Translation of Rare Words with Subword Units" (2015) by Sennrich et al., https://arxiv.org/abs/1508.07909

The code for the byte pair encoding tokenizer used to train GPT-2 was open-sourced by OpenAI:

- https://github.com/openai/gpt-2/blob/master/src/encoder.py

OpenAI provides an interactive web UI to illustrate how the byte pair tokenizer in GPT models works:

- https://platform.openai.com/tokenizer

For readers interested in coding and training a BPE tokenizer from the ground up, Andrej Karpathy's GitHub repository `minbpe` offers a minimal and readable implementation:

- "A Minimal Implementation of a BPE Tokenizer," https://github.com/karpathy/minbpe

Readers who are interested in studying alternative tokenization schemes that are used by some other popular LLMs can find more information in the SentencePiece and WordPiece papers:

- "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing" (2018) by Kudo and Richardson, https://aclanthology.org/D18-2012/
- "Fast WordPiece Tokenization" (2020) by Song et al., https://arxiv.org/abs/2012.15524

Chapter 3

Readers interested in learning more about Bahdanau attention for RNN and language translation can find detailed insights in the following paper:

- "Neural Machine Translation by Jointly Learning to Align and Translate" (2014) by Bahdanau, Cho, and Bengio, https://arxiv.org/abs/1409.0473

The concept of self-attention as scaled dot-product attention was introduced in the original transformer paper:

- "Attention Is All You Need" (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762

FlashAttention is a highly efficient implementation of a self-attention mechanism, which accelerates the computation process by optimizing memory access patterns. FlashAttention is mathematically the same as the standard self-attention mechanism but optimizes the computational process for efficiency:

- "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (2022) by Dao et al., https://arxiv.org/abs/2205.14135
- "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning" (2023) by Dao, https://arxiv.org/abs/2307.08691

PyTorch implements a function for self-attention and causal attention that supports FlashAttention for efficiency. This function is beta and subject to change:

- `scaled_dot_product_attention` documentation: https://mng.bz/NRJd

PyTorch also implements an efficient `MultiHeadAttention` class based on the `scaled_dot_product` function:

- `MultiHeadAttention` documentation: https://mng.bz/DdJV

Dropout is a regularization technique used in neural networks to prevent overfitting by randomly dropping units (along with their connections) from the neural network during training:

- "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" (2014) by Srivastava et al., https://jmlr.org/papers/v15/srivastava14a.html

While using the multi-head attention based on scaled-dot product attention remains the most common variant of self-attention in practice, authors have found that it's possible to also achieve good performance without the value weight matrix and projection layer:

- "Simplifying Transformer Blocks" (2023) by He and Hofmann, https://arxiv.org/abs/2311.01906

Chapter 4

The following paper introduces a technique that stabilizes the hidden state dynamics neural networks by normalizing the summed inputs to the neurons within a hidden layer, significantly reducing training time compared to previously published methods:

- "Layer Normalization" (2016) by Ba, Kiros, and Hinton, https://arxiv.org/abs/1607.06450

Post-LayerNorm, used in the original transformer model, applies layer normalization after the self-attention and feed forward networks. In contrast, Pre-LayerNorm, as adopted in models like GPT-2 and newer LLMs, applies layer normalization before these components, which can lead to more stable training dynamics and has been shown to improve performance in some cases, as discussed in the following papers:

- "On Layer Normalization in the Transformer Architecture" (2020) by Xiong et al., https://arxiv.org/abs/2002.04745
- "ResiDual: Transformer with Dual Residual Connections" (2023) by Tie et al., https://arxiv.org/abs/2304.14802

A popular variant of LayerNorm used in modern LLMs is RMSNorm due to its improved computing efficiency. This variant simplifies the normalization process by normalizing the inputs using only the root mean square of the inputs, without subtracting the mean before squaring. This means it does not center the data before computing the scale. RMSNorm is described in more detail in

- "Root Mean Square Layer Normalization" (2019) by Zhang and Sennrich, https://arxiv.org/abs/1910.07467

The Gaussian Error Linear Unit (GELU) activation function combines the properties of both the classic ReLU activation function and the normal distribution's cumulative distribution function to model layer outputs, allowing for stochastic regularization and nonlinearities in deep learning models:

- "Gaussian Error Linear Units (GELUs)" (2016) by Hendricks and Gimpel, https://arxiv.org/abs/1606.08415

The GPT-2 paper introduced a series of transformer-based LLMs with varying sizes—124 million, 355 million, 774 million, and 1.5 billion parameters:

- "Language Models Are Unsupervised Multitask Learners" (2019) by Radford et al., https://mng.bz/DMv0

OpenAI's GPT-3 uses fundamentally the same architecture as GPT-2, except that the largest version (175 billion) is 100x larger than the largest GPT-2 model and has been trained on much more data. Interested readers can refer to the official GPT-3 paper by OpenAI and the technical overview by Lambda Labs, which calculates that training GPT-3 on a single RTX 8000 consumer GPU would take 665 years:

- "Language Models are Few-Shot Learners" (2023) by Brown et al., https://arxiv.org/abs/2005.14165
- "OpenAI's GPT-3 Language Model: A Technical Overview," https://lambdalabs.com/blog/demystifying-gpt-3

NanoGPT is a code repository with a minimalist yet efficient implementation of a GPT-2 model, similar to the model implemented in this book. While the code in this book is different from nanoGPT, this repository inspired the reorganization of a large GPT Python parent class implementation into smaller submodules:

- "NanoGPT, a Repository for Training Medium-Sized GPTs, https://github.com/karpathy/nanoGPT

An informative blog post showing that most of the computation in LLMs is spent in the feed forward layers rather than attention layers when the context size is smaller than 32,000 tokens is:

- "In the Long (Context) Run" by Harm de Vries, https://www.harmdevries.com/post/context-length/

Chapter 5

For information on detailing the loss function and applying a log transformation to make it easier to handle for mathematical optimization, see my lecture video:

- L8.2 Logistic Regression Loss Function, https://www.youtube.com/watch?v=GxJe0DZvydM

The following lecture and code example by the author explain how PyTorch's cross-entropy functions works under the hood:

- L8.7.1 OneHot Encoding and Multi-category Cross Entropy, https://www.youtube.com/watch?v=4n71-tZ94yk
- Understanding Onehot Encoding and Cross Entropy in PyTorch, https://mng.bz/o05v

The following two papers detail the dataset, hyperparameter, and architecture details used for pretraining LLMs:

- "Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling" (2023) by Biderman et al., https://arxiv.org/abs/2304.01373
- "OLMo: Accelerating the Science of Language Models" (2024) by Groeneveld et al., https://arxiv.org/abs/2402.00838

The following supplementary code available for this book contains instructions for preparing 60,000 public domain books from Project Gutenberg for LLM training:

- Pretraining GPT on the Project Gutenberg Dataset, https://mng.bz/Bdw2

Chapter 5 discusses the pretraining of LLMs, and appendix D covers more advanced training functions, such as linear warmup and cosine annealing. The following paper finds that similar techniques can be successfully applied to continue pretraining already pretrained LLMs, along with additional tips and insights:

- "Simple and Scalable Strategies to Continually Pre-train Large Language Models" (2024) by Ibrahim et al., https://arxiv.org/abs/2403.08763

BloombergGPT is an example of a domain-specific LLM created by training on both general and domain-specific text corpora, specifically in the field of finance:

- "BloombergGPT: A Large Language Model for Finance" (2023) by Wu et al., https://arxiv.org/abs/2303.17564

GaLore is a recent research project that aims to make LLM pretraining more efficient. The required code change boils down to just replacing PyTorch's `AdamW` optimizer in the training function with the `GaLoreAdamW` optimizer provided by the `galore-torch` Python package:

- "GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection" (2024) by Zhao et al., https://arxiv.org/abs/2403.03507
- GaLore code repository, https://github.com/jiaweizzhao/GaLore

The following papers and resources share openly available, large-scale pretraining datasets for LLMs that consist of hundreds of gigabytes to terabytes of text data:

- "Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research" (2024) by Soldaini et al., https://arxiv.org/abs/2402.00159

- "The Pile: An 800GB Dataset of Diverse Text for Language Modeling" (2020) by Gao et al., https://arxiv.org/abs/2101.00027
- "The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only," (2023) by Penedo et al., https://arxiv.org/abs/2306.01116
- "RedPajama," by Together AI, https://mng.bz/d6nw
- The FineWeb Dataset, which includes more than 15 trillion tokens of cleaned and deduplicated English web data sourced from CommonCrawl, https://mng.bz/rVzy

The paper that originally introduced top-k sampling is

- "Hierarchical Neural Story Generation" (2018) by Fan et al., https://arxiv.org/abs/1805.04833

An alternative to top-k sampling is top-p sampling (not covered in chapter 5), which selects from the smallest set of top tokens whose cumulative probability exceeds a threshold p, while top-k sampling picks from the top k tokens by probability:

- Top-p sampling, https://en.wikipedia.org/wiki/Top-p_sampling

Beam search (not covered in chapter 5) is an alternative decoding algorithm that generates output sequences by keeping only the top-scoring partial sequences at each step to balance efficiency and quality:

- "Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models" (2016) by Vijayakumar et al., https://arxiv.org/abs/1610.02424

Chapter 6

Additional resources that discuss the different types of fine-tuning are

- "Using and Finetuning Pretrained Transformers," https://mng.bz/VxJG
- "Finetuning Large Language Models," https://mng.bz/x28X

Additional experiments, including a comparison of fine-tuning the first output token versus the last output token, can be found in the supplementary code material on GitHub:

- Additional spam classification experiments, https://mng.bz/AdJx

For a binary classification task, such as spam classification, it is technically possible to use only a single output node instead of two output nodes, as I discuss in the following article:

- "Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch," https://mng.bz/ZEJA

You can find additional experiments on fine-tuning different layers of an LLM in the following article, which shows that fine-tuning the last transformer block, in addition to the output layer, improves the predictive performance substantially:

- "Finetuning Large Language Models," https://mng.bz/RZJv

Readers can find additional resources and information for dealing with imbalanced classification datasets in the imbalanced-learn documentation:

- "Imbalanced-Learn User Guide," https://mng.bz/2KNa

For readers interested in classifying spam emails rather than spam text messages, the following resource provides a large email spam classification dataset in a convenient CSV format similar to the dataset format used in chapter 6:

- Email Spam Classification Dataset, https://mng.bz/1GEq

GPT-2 is a model based on the decoder module of the transformer architecture, and its primary purpose is to generate new text. As an alternative, encoder-based models such as BERT and RoBERTa can be effective for classification tasks:

- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al., https://arxiv.org/abs/1810.04805
- "RoBERTa: A Robustly Optimized BERT Pretraining Approach" (2019) by Liu et al., https://arxiv.org/abs/1907.11692
- "Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews," https://mng.bz/PZJR

Recent papers are showing that the classification performance can be further improved by removing the causal mask during classification fine-tuning alongside other modifications:

- "Label Supervised LLaMA Finetuning" (2023) by Li et al., https://arxiv.org/abs/2310.01208
- "LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders" (2024) by BehnamGhader et al., https://arxiv.org/abs/2404.05961

Chapter 7

The Alpaca dataset for instruction fine-tuning contains 52,000 instruction–response pairs and is one of the first and most popular publicly available datasets for instruction fine-tuning:

- "Stanford Alpaca: An Instruction-Following Llama Model," https://github.com/tatsu-lab/stanford_alpaca

Additional publicly accessible datasets suitable for instruction fine-tuning include

- LIMA, https://huggingface.co/datasets/GAIR/lima
  - For more information, see "LIMA: Less Is More for Alignment," Zhou et al., https://arxiv.org/abs/2305.11206

- UltraChat, https://huggingface.co/datasets/openchat/ultrachat-sharegpt
  - A large-scale dataset consisting of 805,000 instruction–response pairs; for more information, see "Enhancing Chat Language Models by Scaling High-quality Instructional Conversations," by Ding et al., https://arxiv.org/abs/2305.14233
- Alpaca GPT4, https://mng.bz/Aa0p
  - An Alpaca-like dataset with 52,000 instruction–response pairs generated with GPT-4 instead of GPT-3.5

Phi-3 is a 3.8-billion-parameter model with an instruction-fine-tuned variant that is reported to be comparable to much larger proprietary models, such as GPT-3.5:

- "Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone" (2024) by Abdin et al., https://arxiv.org/abs/2404.14219

Researchers propose a synthetic instruction data generation method that generates 300,000 high-quality instruction-response pairs from an instruction fine-tuned Llama-3 model. A pretrained Llama 3 base model fine-tuned on these instruction examples performs comparably to the original instruction fine-tuned Llama-3 model:

- "Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing" (2024) by Xu et al., https://arxiv.org/abs/2406.08464

Research has shown that not masking the instructions and inputs in instruction fine-tuning effectively improves performance on various NLP tasks and open-ended generation benchmarks, particularly when trained on datasets with lengthy instructions and brief outputs or when using a small number of training examples:

- "Instruction Tuning with Loss Over Instructions" (2024) by Shi, https://arxiv.org/abs/2405.14394

Prometheus and PHUDGE are openly available LLMs that match GPT-4 in evaluating long-form responses with customizable criteria. We don't use these because at the time of this writing, they are not supported by Ollama and thus cannot be executed efficiently on a laptop:

- "Prometheus: Inducing Finegrained Evaluation Capability in Language Models" (2023) by Kim et al., https://arxiv.org/abs/2310.08491
- "PHUDGE: Phi-3 as Scalable Judge" (2024) by Deshwal and Chawla, "https://arxiv.org/abs/2405.08029
- "Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models" (2024), by Kim et al., https://arxiv.org/abs/2405.01535

The results in the following report support the view that large language models primarily acquire factual knowledge during pretraining and that fine-tuning mainly enhances their efficiency in using this knowledge. Furthermore, this study explores

how fine-tuning large language models with new factual information affects their ability to use preexisting knowledge, revealing that models learn new facts more slowly and their introduction during fine-tuning increases the model's tendency to generate incorrect information:

- "Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?" (2024) by Gekhman, https://arxiv.org/abs/2405.05904

Preference fine-tuning is an optional step after instruction fine-tuning to align the LLM more closely with human preferences. The following articles by the author provide more information about this process:

- "LLM Training: RLHF and Its Alternatives," https://mng.bz/ZVPm
- "Tips for LLM Pretraining and Evaluating Reward Models," https://mng.bz/RNXj

Appendix A

While appendix A should be sufficient to get you up to speed, if you are looking for more comprehensive introductions to deep learning, I recommend the following books:

- *Machine Learning with PyTorch and Scikit-Learn* (2022) by Sebastian Raschka, Hayden Liu, and Vahid Mirjalili. ISBN 978-1801819312
- *Deep Learning with PyTorch* (2021) by Eli Stevens, Luca Antiga, and Thomas Viehmann. ISBN 978-1617295263

For a more thorough introduction to the concepts of tensors, readers can find a 15-minute video tutorial that I recorded:

- "Lecture 4.1: Tensors in Deep Learning," https://www.youtube.com/watch?v=JXfDlgrfOBY

If you want to learn more about model evaluation in machine learning, I recommend my article

- "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning" (2018) by Sebastian Raschka, https://arxiv.org/abs/1811.12808

For readers who are interested in a refresher or gentle introduction to calculus, I've written a chapter on calculus that is freely available on my website:

- "Introduction to Calculus," by Sebastian Raschka, https://mng.bz/WEyW

Why does PyTorch not call `optimizer.zero_grad()` automatically for us in the background? In some instances, it may be desirable to accumulate the gradients, and PyTorch will leave this as an option for us. If you want to learn more about gradient accumulation, please see the following article:

- "Finetuning Large Language Models on a Single GPU Using Gradient Accumulation" by Sebastian Raschka, https://mng.bz/8wPD

This appendix covers DDP, which is a popular approach for training deep learning models across multiple GPUs. For more advanced use cases where a single model doesn't fit onto the GPU, you may also consider PyTorch's Fully Sharded Data Parallel (FSDP) method, which performs distributed data parallelism and distributes large layers across different GPUs. For more information, see this overview with further links to the API documentation:

- "Introducing PyTorch Fully Sharded Data Parallel (FSDP) API," https://mng.bz/EZJR

appendix C
Exercise solutions

The complete code examples for the exercises' answers can be found in the supplementary GitHub repository at https://github.com/rasbt/LLMs-from-scratch.

Chapter 2

Exercise 2.1

You can obtain the individual token IDs by prompting the encoder with one string at a time:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

This prints

```
[33901]
[86]
# ...
```

You can then use the following code to assemble the original string:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

This returns

```
'Akwirw ier'
```

Exercise 2.2

The code for the data loader with `max_length=2` and `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=2, stride=2
)
```

It produces batches of the following format:

```
tensor([[  40,  367],
        [2885, 1464],
        [1807, 3619],
        [ 402,  271]])
```

The code of the second data loader with `max_length=8` and `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=8, stride=2
)
```

An example batch looks like

```
tensor([[   40,   367,  2885,  1464,  1807,  3619,   402,   271],
        [ 2885,  1464,  1807,  3619,   402,   271, 10899,  2138],
        [ 1807,  3619,   402,   271, 10899,  2138,   257,  7026],
        [  402,   271, 10899,  2138,   257,  7026, 15632,   438]])
```

Chapter 3

Exercise 3.1

The correct weight assignment is

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Exercise 3.2

To achieve an output dimension of 2, similar to what we had in single-head attention, we need to change the projection dimension `d_out` to 1.

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num_heads=2)
```

Exercise 3.3

The initialization for the smallest GPT-2 model is

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

Chapter 4

Exercise 4.1

We can calculate the number of parameters in the feed forward and attention modules as follows:

```
block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Total number of parameters in feed forward module: {total_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_params:,}")
```

As we can see, the feed forward module contains approximately twice as many parameters as the attention module:

```
Total number of parameters in feed forward module: 4,722,432
Total number of parameters in attention module: 2,360,064
```

Exercise 4.2

To instantiate the other GPT model sizes, we can modify the configuration dictionary as follows (here shown for GPT-2 XL):

```
GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)
```

Then, reusing the code from section 4.6 to calculate the number of parameters and RAM requirements, we find

```
gpt2-xl:
Total number of parameters: 1,637,792,000
Number of trainable parameters considering weight tying: 1,557,380,800
Total size of the model: 6247.68 MB
```

Exercise 4.3

There are three distinct places in chapter 4 where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module. We can control the dropout rates for each of the layers by coding them separately in the config file and then modifying the code implementation accordingly.

The modified configuration is as follows:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,
```

```
    "context_length": 1024,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,
    "drop_rate_shortcut": 0.1,
    "drop_rate_emb": 0.1,
    "qkv_bias": False
}
```

Dropout for multi-head attention

Dropout for shortcut connections

Dropout for embedding layer

The modified `TransformerBlock` and `GPTModel` look like

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate_shortcut"]
        )

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x


class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"]
        )
```

Dropout for multi-head attention

Dropout for shortcut connections

```
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"]
        )
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logitss
```

Dropout for embedding layer

Chapter 5

Exercise 5.1

We can print the number of times the token (or word) "pizza" is sampled using the `print_sampled_tokens` function we defined in this section. Let's start with the code we defined in section 5.3.1.

The "pizza" token is sampled 0x if the temperature is 0 or 0.1, and it is sampled 32× if the temperature is scaled up to 5. The estimated probability is $32/1000 \times 100\% = 3.2\%$.

The actual probability is 4.3% and is contained in the rescaled softmax probability tensor (`scaled_probas[2][6]`).

Exercise 5.2

Top-k sampling and temperature scaling are settings that have to be adjusted based on the LLM and the desired degree of diversity and randomness in the output.

When using relatively small top-k values (e.g., smaller than 10) and when the temperature is set below 1, the model's output becomes less random and more deterministic. This setting is useful when we need the generated text to be more predictable, coherent, and closer to the most likely outcomes based on the training data.

Applications for such low k and temperature settings include generating formal documents or reports where clarity and accuracy are most important. Other examples of applications include technical analysis or code-generation tasks, where precision is

crucial. Also, question answering and educational content require accurate answers where a temperature below 1 is helpful.

On the other hand, larger top-k values (e.g., values in the range of 20 to 40) and temperature values above 1 are useful when using LLMs for brainstorming or generating creative content, such as fiction.

Exercise 5.3

There are multiple ways to force deterministic behavior with the `generate` function:

1. Setting to `top_k=None` and applying no temperature scaling
2. Setting `top_k=1`

Exercise 5.4

In essence, we have to load the model and optimizer that we saved in the main chapter:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Then, call the `train_simple_function` with `num_epochs=1` to train the model for another epoch.

Exercise 5.5

We can use the following code to calculate the training and validation set losses of the GPT model:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

The resulting losses for the 124-million parameter are as follows:

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

The main observation is that the training and validation set performances are in the same ballpark. This can have multiple explanations:

1. "The Verdict" was not part of the pretraining dataset when OpenAI trained GPT-2. Hence, the model is not explicitly overfitting to the training set and performs similarly well on the training and validation set portions of "The Verdict." (The validation set loss is slightly lower than the training set loss, which is unusual in deep learning. However, it's likely due to random noise since the dataset is relatively small. In practice, if there is no overfitting, the training and validation set performances are expected to be roughly identical).

2 "The Verdict" was part of GPT-2's training dataset. In this case, we can't tell whether the model is overfitting the training data because the validation set would have been used for training as well. To evaluate the degree of overfitting, we'd need a new dataset generated after OpenAI finished training GPT-2 to make sure that it couldn't have been part of the pretraining.

Exercise 5.6

In the main chapter, we experimented with the smallest GPT-2 model, which has only 124-million parameters. The reason was to keep the resource requirements as low as possible. However, you can easily experiment with larger models with minimal code changes. For example, instead of loading the 1,558 million instead of 124 million model weights in chapter 5, the only two lines of code that we have to change are the following:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

The updated code is

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

Chapter 6

Exercise 6.1

We can pad the inputs to the maximum number of tokens the model supports by setting the max length to `max_length = 1024` when initializing the datasets:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

However, the additional padding results in a substantially worse test accuracy of 78.33% (vs. the 95.67% in the main chapter).

Exercise 6.2

Instead of fine-tuning just the final transformer block, we can fine-tune the entire model by removing the following lines from the code:

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

This modification results in a 1% improved test accuracy of 96.67% (vs. the 95.67% in the main chapter).

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.

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.

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
```

The result was

```
13.8 µs ± 425 ns per loop
```

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

appendix D
Adding bells and whistles to the training loop

In this appendix, we enhance the training function for the pretraining and fine-tuning processes covered in chapters 5 to 7. In particular, it covers *learning rate warmup*, *cosine decay*, and *gradient clipping*. We then incorporate these techniques into the training function and pretrain an LLM.

To make the code self-contained, we reinitialize the model we trained in chapter 5:

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,

    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

Vocabulary size
Shortened context length (orig: 1024)
Embedding dimension
Number of attention heads
Number of layers
Dropout rate
Query-key-value bias

After initializing the model, we need to initialize the data loaders. First, we load the "The Verdict" short story:

```
import os
import urllib.request

file_path = "the-verdict.txt"

url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
    "main/ch02/01_main-chapter-code/the-verdict.txt"
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()
```

Next, we load the `text_data` into the data loaders:

```
from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

D.1 Learning rate warmup

Implementing a learning rate warmup can stabilize the training of complex models such as LLMs. This process involves gradually increasing the learning rate from a very low initial value (`initial_lr`) to a maximum value specified by the user (`peak_lr`). Starting the training with smaller weight updates decreases the risk of the model encountering large, destabilizing updates during its training phase.

Suppose we plan to train an LLM for 15 epochs, starting with an initial learning rate of 0.0001 and increasing it to a maximum learning rate of 0.01:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
```

The number of warmup steps is usually set between 0.1% and 20% of the total number of steps, which we can calculate as follows:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps)        <-- 20% warmup
print(warmup_steps)
```

This prints `27`, meaning that we have 20 warmup steps to increase the initial learning rate from 0.0001 to 0.01 in the first 27 training steps.

Next, we implement a simple training loop template to illustrate this warmup process:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps

global_step = -1
track_lrs = []

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
```

This increment is determined by how much we increase the inital_lr in each of the 20 warmup steps.

Executes a typical training loop iterating over the batches in the training loader in each epoch

Updates the learning rate if we are still in the warmup phase

Applies the calculated learning rate to the optimizer

In a complete training loop, the loss and the model updates would be calculated, which are omitted here for simplicity.

After running the preceding code, we visualize how the learning rate was changed by the training loop to verify that the learning rate warmup works as intended:

```
import matplotlib.pyplot as plt

plt.ylabel("Learning rate")
plt.xlabel("Step")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

The resulting plot shows that the learning rate starts with a low value and increases for 20 steps until it reaches the maximum value after 20 steps (figure D.1).

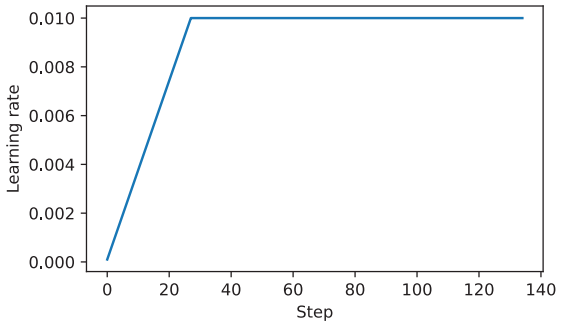


Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

Next, we will modify the learning rate further so that it decreases after reaching the maximum learning rate, which further helps improve the model training.

D.2 Cosine decay

Another widely adopted technique for training complex deep neural networks and LLMs is *cosine decay*. This method modulates the learning rate throughout the training epochs, making it follow a cosine curve after the warmup stage.

In its popular variant, cosine decay reduces (or decays) the learning rate to nearly zero, mimicking the trajectory of a half-cosine cycle. The gradual learning decrease in cosine decay aims to decelerate the pace at which the model updates its weights. This is particularly important because it helps minimize the risk of overshooting the loss minima during the training process, which is essential for ensuring the stability of the training during its later phases.

We can modify the training loop template by adding cosine decay:

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
```

Applies linear warmup

Uses cosine annealing after warmup

```
        lr = min_lr + (peak_lr - min_lr) * 0.5 * (
            1 + math.cos(math.pi * progress)
        )

    for param_group in optimizer.param_groups:
        param_group["lr"] = lr
    track_lrs.append(optimizer.param_groups[0]["lr"])
```

Again, to verify that the learning rate has changed as intended, we plot the learning rate:

```
plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()
```

The resulting learning rate plot shows that the learning rate starts with a linear warmup phase, which increases for 20 steps until it reaches the maximum value after 20 steps. After the 20 steps of linear warmup, cosine decay kicks in, reducing the learning rate gradually until it reaches its minimum (figure D.2).

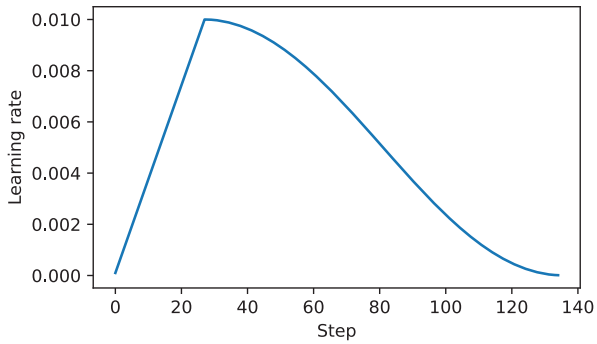


Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.

D.3 Gradient clipping

Gradient clipping is another important technique for enhancing stability during LLM training. This method involves setting a threshold above which gradients are downscaled to a predetermined maximum magnitude. This process ensures that the updates to the model's parameters during backpropagation stay within a manageable range.

For example, applying the `max_norm=1.0` setting within PyTorch's `clip_grad_norm_` function ensures that the norm of the gradients does not surpass 1.0. Here, the term "norm" signifies the measure of the gradient vector's length, or magnitude, within the model's parameter space, specifically referring to the L2 norm, also known as the Euclidean norm.

In mathematical terms, for a vector $\boldsymbol{v}$ composed of components $\boldsymbol{v} = [v_1, v_2, ..., v_n]$, the L2 norm is

$$|\boldsymbol{v}|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

This calculation method is also applied to matrices. For instance, consider a gradient matrix given by

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

If we want to clip these gradients to a `max_norm` of 1, we first compute the L2 norm of these gradients, which is

$$|\mathbf{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Given that $|\mathbf{G}|_2 = 5$ exceeds our `max_norm` of 1, we scale down the gradients to ensure their norm equals exactly 1. This is achieved through a scaling factor, calculated as `max_norm`$/|\mathbf{G}|_2 = 1/5$. Consequently, the adjusted gradient matrix **G'** becomes

$$\mathbf{G}' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{4}{5} \end{bmatrix}$$

To illustrate this gradient clipping process, we begin by initializing a new model and calculating the loss for a training batch, similar to the procedure in a standard training loop:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

Upon calling the `.backward()` method, PyTorch calculates the loss gradients and stores them in a `.grad` attribute for each model weight (parameter) tensor.

To clarify the point, we can define the following `find_highest_gradient` utility function to identify the highest gradient value by scanning all the `.grad` attributes of the model's weight tensors after calling `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

The largest gradient value identified by the preceding code is

```
tensor(0.0411)
```

Let's now apply gradient clipping and see how this affects the largest gradient value:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

The largest gradient value after applying the gradient clipping with the max norm of 1 is substantially smaller than before:

```
tensor(0.0185)
```

D.4 The modified training function

Finally, we improve the `train_model_simple` training function (see chapter 5) by adding the three concepts introduced herein: linear warmup, cosine decay, and gradient clipping. Together, these methods help stabilize LLM training.

The code, with the changes compared to the `train_model_simple` annotated, is as follows:

```
from chapter05 import evaluate_model, generate_and_print_sample


def train_model(model, train_loader, val_loader, optimizer, device,
                n_epochs, eval_freq, eval_iter, start_context, tokenizer,
                warmup_steps, initial_lr=3e-05, min_lr=1e-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1

    peak_lr = optimizer.param_groups[0]["lr"]
    total_training_steps = len(train_loader) * n_epochs
    lr_increment = (peak_lr - initial_lr) / warmup_steps

    for epoch in range(n_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            global_step += 1

            if global_step < warmup_steps:
                lr = initial_lr + global_step * lr_increment
            else:
                progress = ((global_step - warmup_steps) /
                            (total_training_steps - warmup_steps))
                lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                    1 + math.cos(math.pi * progress))
```

Retrieves the initial learning rate from the optimizer, assuming we use it as the peak learning rate

Calculates the total number of iterations in the training process

Calculates the learning rate increment during the warmup phase

Adjusts the learning rate based on the current phase (warmup or cosine annealing)

```
            for param_group in optimizer.param_groups:
                param_group["lr"] = lr
            track_lrs.append(lr)
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward()

            if global_step >= warmup_steps:
                torch.nn.utils.clip_grad_norm_(
                    model.parameters(), max_norm=1.0
                )

            optimizer.step()
            tokens_seen += input_batch.numel()

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader,
                    device, eval_iter
                )
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Iter {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )


        generate_and_print_sample(
            model, tokenizer, device, start_context
        )

    return train_losses, val_losses, track_tokens_seen, track_lrs
```

Applies the calculated learning rate to the optimizer

Applies gradient clipping after the warmup phase to avoid exploding gradients

Everything below here remains unchanged compared to the train_model_simple function used in chapter 5.

After defining the `train_model` function, we can use it in a similar fashion to train the model compared to the `train_model_simple` method we used for pretraining:

```
import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 0.001
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding("gpt2")

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)
```

The training will take about 5 minutes to complete on a MacBook Air or similar laptop and prints the following outputs:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 7.949, Val loss 8.184
Ep 2 (Iter 000015): Train loss 6.362, Val loss 6.876
Every effort moves you,,,,,,,,,,,,,,,,,,,, the,,,,,,,,, the,,,,,,,,,,
the,,,,,,,,
...
Ep 15 (Iter 000130): Train loss 0.035, Val loss 6.938
Every effort moves you?"  "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!"  He laughed again, and threw back his head to look up
at the sketch of the donkey. "There were days when I
```

Like pretraining, the model begins to overfit after a few epochs since it is a very small dataset, and we iterate over it multiple times. Nonetheless, we can see that the function is working since it minimizes the training set loss.

Readers are encouraged to train the model on a larger text dataset and compare the results obtained with this more sophisticated training function to the results that can be obtained with the `train_model_simple` function.

appendix E
Parameter-efficient fine-tuning with LoRA

Low-rank adaptation (*LoRA*) is one of the most widely used techniques for *parameter-efficient fine-tuning*. The following discussion is based on the spam classification fine-tuning example given in chapter 6. However, LoRA fine-tuning is also applicable to the supervised *instruction fine-tuning* discussed in chapter 7.

E.1 Introduction to LoRA

LoRA is a technique that adapts a pretrained model to better suit a specific, often smaller dataset by adjusting only a small subset of the model's weight parameters. The "low-rank" aspect refers to the mathematical concept of limiting model adjustments to a smaller dimensional subspace of the total weight parameter space. This effectively captures the most influential directions of the weight parameter changes during training. The LoRA method is useful and popular because it enables efficient fine-tuning of large models on task-specific data, significantly cutting down on the computational costs and resources usually required for fine-tuning.

Suppose a large weight matrix W is associated with a specific layer. LoRA can be applied to all linear layers in an LLM. However, we focus on a single layer for illustration purposes.

When training deep neural networks, during backpropagation, we learn a ΔW matrix, which contains information on how much we want to update the original weight parameters to minimize the loss function during training. Hereafter, I use the term "weight" as shorthand for the model's weight parameters.

In regular training and fine-tuning, the weight update is defined as follows:

$$W_{updated} = W + \Delta W$$

The LoRA method, proposed by Hu et al. (https://arxiv.org/abs/2106.09685), offers a more efficient alternative to computing the weight updates ΔW by learning an approximation of it:

$$\Delta W \approx AB$$

where A and B are two matrices much smaller than W, and AB represents the matrix multiplication product between A and B.

Using LoRA, we can then reformulate the weight update we defined earlier:

$$W_{updated} = W + AB$$

Figure E.1 illustrates the weight update formulas for full fine-tuning and LoRA side by side.

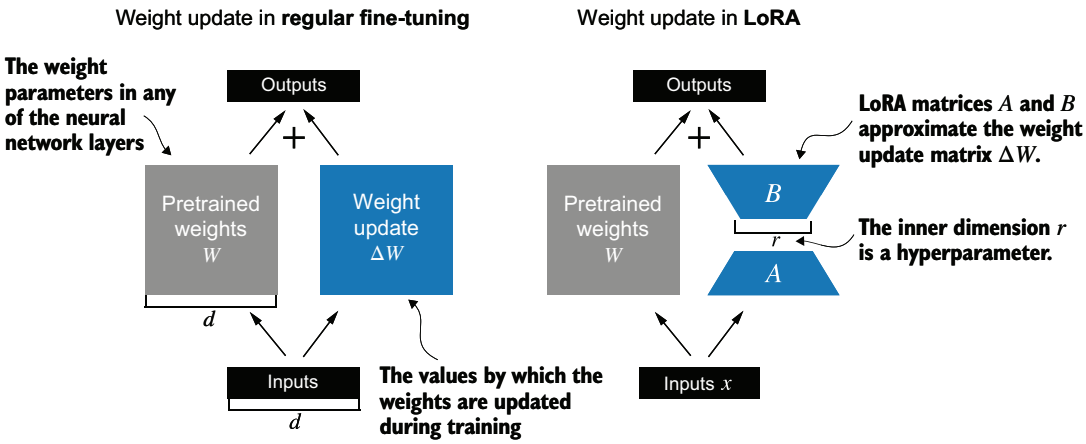


Figure E.1 A comparison between weight update methods: regular fine-tuning and LoRA. Regular fine-tuning involves updating the pretrained weight matrix *W* directly with ΔW (left). LoRA uses two smaller matrices, *A* and *B*, to approximate ΔW, where the product *AB* is added to *W*, and *r* denotes the inner dimension, a tunable hyperparameter (right).

If you paid close attention, you might have noticed that the visual representations of full fine-tuning and LoRA in figure E.1 differ slightly from the earlier presented formulas. This variation is attributed to the distributive law of matrix multiplication, which allows us to separate the original and updated weights rather than combine them. For example, in the case of regular fine-tuning with x as the input data, we can express the computation as

$$x(W + \Delta W) = xW + x\Delta W$$

Similarly, we can write the following for LoRA:

$$x(W + AB) = xW + xAB$$

Besides reducing the number of weights to update during training, the ability to keep the LoRA weight matrices separate from the original model weights makes LoRA even more useful in practice. Practically, this allows for the pretrained model weights to remain unchanged, with the LoRA matrices being applied dynamically after training when using the model.

Keeping the LoRA weights separate is very useful in practice because it enables model customization without needing to store multiple complete versions of an LLM. This reduces storage requirements and improves scalability, as only the smaller LoRA matrices need to be adjusted and saved when we customize LLMs for each specific customer or application.

Next, let's see how LoRA can be used to fine-tune an LLM for spam classification, similar to the fine-tuning example in chapter 6.

E.2 Preparing the dataset

Before applying LoRA to the spam classification example, we must load the dataset and pretrained model we will work with. The code here repeats the data preparation from chapter 6. (Instead of repeating the code, we could open and run the chapter 6 notebook and insert the LoRA code from section E.4 there.)

First, we download the dataset and save it as CSV files.

Listing E.1 Downloading and preparing the dataset

```
from pathlib import Path
import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
    create_balanced_dataset,
    random_split
)

url = \
"https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
balanced_df = create_balanced_dataset(df)
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)
```

```
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Next, we create the `SpamDataset` instances.

Listing E.2 Instantiating PyTorch datasets

```
import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
    tokenizer=tokenizer
)
val_dataset = SpamDataset("validation.csv",
    max_length=train_dataset.max_length, tokenizer=tokenizer
)
test_dataset = SpamDataset(
    "test.csv", max_length=train_dataset.max_length, tokenizer=tokenizer
)
```

After creating the PyTorch dataset objects, we instantiate the data loaders.

Listing E.3 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
```

As a verification step, we iterate through the data loaders and check that the batches contain eight training examples each, where each training example consists of 120 tokens:

```
print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

The output is

```
Train loader:
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

Lastly, we print the total number of batches in each dataset:

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

In this case, we have the following number of batches per dataset:

```
130 training batches
19 validation batches
38 test batches
```

E.3 Initializing the model

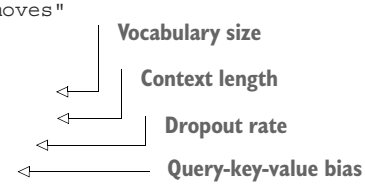
We repeat the code from chapter 6 to load and prepare the pretrained GPT model. We begin by downloading the model weights and loading them into the `GPTModel` class.

Listing E.4 Loading a pretrained GPT model

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt


CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257,
    "context_length": 1024,
    "drop_rate": 0.0,
    "qkv_bias": True
}
```



```
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},
}

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()
```

To ensure that the model was loaded corrected, let's double-check that it generates coherent text:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))
```

The following output shows that the model generates coherent text, which is an indicator that the model weights are loaded correctly:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Next, we prepare the model for classification fine-tuning, similar to chapter 6, where we replace the output layer:

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Lastly, we calculate the initial classification accuracy of the not-fine-tuned model (we expect this to be around 50%, which means that the model is not able to distinguish between spam and nonspam messages yet reliably):

```
from chapter06 import calc_accuracy_loader

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)


print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The initial prediction accuracies are

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

E.4 Parameter-efficient fine-tuning with LoRA

Next, we modify and fine-tune the LLM using LoRA. We begin by initializing a LoRALayer that creates the matrices A and B, along with the `alpha` scaling factor and the `rank` (r) setting. This layer can accept an input and compute the corresponding output, as illustrated in figure E.2.

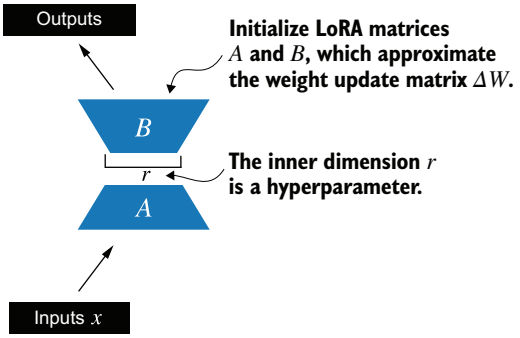


Figure E.2 The LoRA matrices *A* and *B* are applied to the layer inputs and are involved in computing the model outputs. The inner dimension *r* of these matrices serves as a setting that adjusts the number of trainable parameters by varying the sizes of *A* and *B*.

In code, this LoRA layer can be implemented as follows.

Listing E.5 Implementing a LoRA layer

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

The same initialization used for Linear layers in PyTorch

The `rank` governs the inner dimension of matrices A and B. Essentially, this setting determines the number of extra parameters introduced by LoRA, which creates balance between the adaptability of the model and its efficiency via the number of parameters used.

The other important setting, `alpha`, functions as a scaling factor for the output from the low-rank adaptation. It primarily dictates the degree to which the output from the adapted layer can affect the original layer's output. This can be seen as a way to regulate the effect of the low-rank adaptation on the layer's output. The `LoRALayer` class we have implemented so far enables us to transform the inputs of a layer.

In LoRA, the typical goal is to substitute existing `Linear` layers, allowing weight updates to be applied directly to the pre-existing pretrained weights, as illustrated in figure E.3.

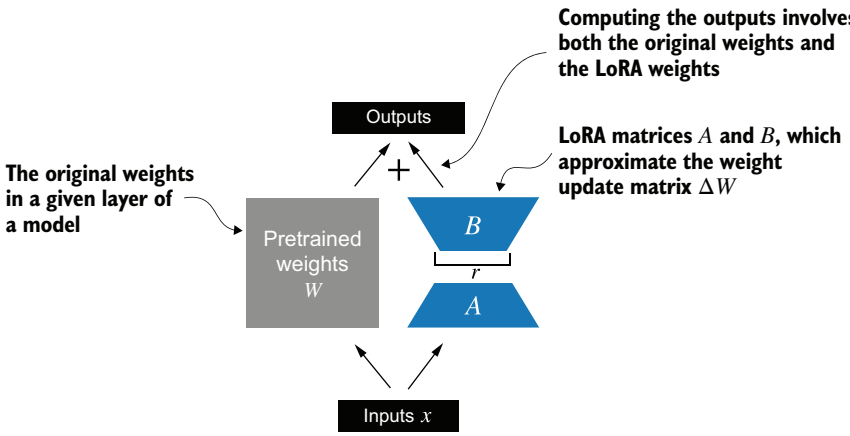


Figure E.3 The integration of LoRA into a model layer. The original pretrained weights (*W*) of a layer are combined with the outputs from LoRA matrices (*A* and *B*), which approximate the weight update matrix (Δ*W*). The final output is calculated by adding the output of the adapted layer (using LoRA weights) to the original output.

To integrate the original `Linear` layer weights, we now create a `LinearWithLoRA` layer. This layer utilizes the previously implemented `LoRALayer` and is designed to replace existing `Linear` layers within a neural network, such as the self-attention modules or feed-forward modules in the `GPTModel`.

Listing E.6 Replacing a `Linear` layers with `LinearWithLora` layers

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

This code combines a standard `Linear` layer with the `LoRALayer`. The `forward` method computes the output by adding the results from the original linear layer and the LoRA layer.

Since the weight matrix B (`self.B` in `LoRALayer`) is initialized with zero values, the product of matrices A and B results in a zero matrix. This ensures that the multiplication does not alter the original weights, as adding zero does not change them.

To apply LoRA to the earlier defined `GPTModel`, we introduce a `replace_linear_with_lora` function. This function will swap all existing `Linear` layers in the model with the newly created `LinearWithLoRA` layers:

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            replace_linear_with_lora(module, rank, alpha)
```

Replaces the Linear layer with LinearWithLoRA

Recursively applies the same function to child modules

We have now implemented all the necessary code to replace the `Linear` layers in the `GPTModel` with the newly developed `LinearWithLoRA` layers for parameter-efficient fine-tuning. Next, we will apply the `LinearWithLoRA` upgrade to all `Linear` layers found in the multihead attention, feed-forward modules, and the output layer of the `GPTModel`, as shown in figure E.4.

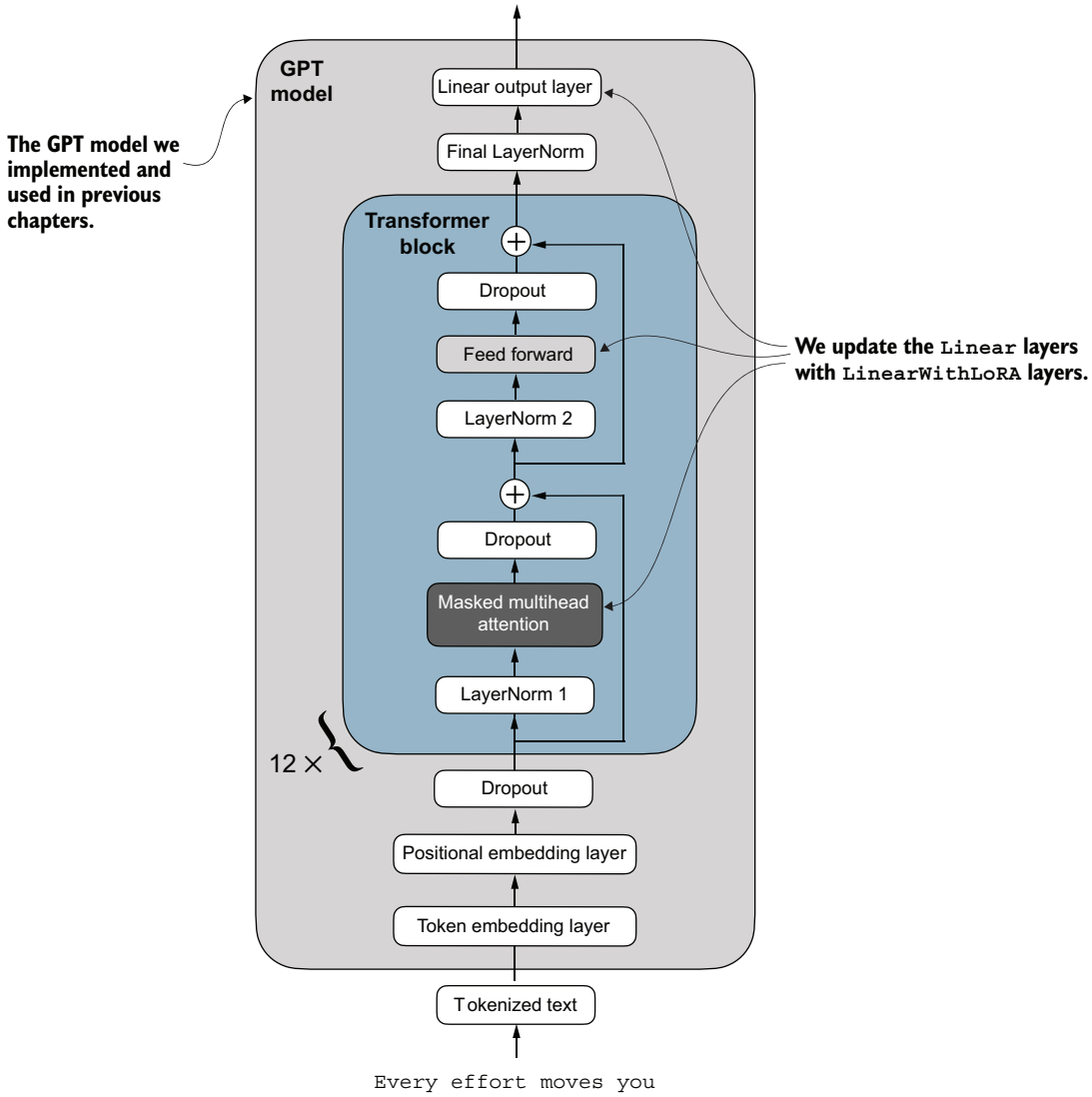


Figure E.4 The architecture of the GPT model. It highlights the parts of the model where `Linear` layers are upgraded to `LinearWithLoRA` layers for parameter-efficient fine-tuning.

Before we apply the `LinearWithLoRA` layer upgrades, we first freeze the original model parameters:

```
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:,}")
```

Now, we can see that none of the 124 million model parameters are trainable:

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

Next, we use the `replace_linear_with_lora` to replace the `Linear` layers:

```
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:,}")
```

After adding the LoRA layers, the number of trainable parameters is as follows:

```
Total trainable LoRA parameters: 2,666,528
```

As we can see, we reduced the number of trainable parameters by almost 50× when using LoRA. A `rank` and `alpha` of 16 are good default choices, but it is also common to increase the rank parameter, which in turn increases the number of trainable parameters. Alpha is usually chosen to be half, double, or equal to the rank.

Let's verify that the layers have been modified as intended by printing the model architecture:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

The output is

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    ...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
```

```
        (out_proj): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): LinearWithLoRA(
            (linear): Linear(in_features=768, out_features=3072, bias=True)
            (lora): LoRALayer()
          )
          (1): GELU()
          (2): LinearWithLoRA(
            (linear): Linear(in_features=3072, out_features=768, bias=True)
            (lora): LoRALayer()
          )
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
  )
  (final_norm): LayerNorm()
  (out_head): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=2, bias=True)
    (lora): LoRALayer()
  )
)
```

The model now includes the new `LinearWithLoRA` layers, which themselves consist of the original `Linear` layers, set to nontrainable, and the new LoRA layers, which we will fine-tune.

Before we begin fine-tuning the model, let's calculate the initial classification accuracy:

```
torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

These accuracy values are identical to the values from chapter 6. This result occurs because we initialized the LoRA matrix B with zeros. Consequently, the product of matrices AB results in a zero matrix. This ensures that the multiplication does not alter the original weights since adding zero does not change them.

Now let's move on to the exciting part—fine-tuning the model using the training function from chapter 6. The training takes about 15 minutes on an M3 MacBook Air laptop and less than half a minute on a V100 or A100 GPU.

Listing E.7 Fine-tuning a model with LoRA layers

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50, eval_iter=5,
        tokenizer=tokenizer
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

The output we see during the training is

```
Ep 1 (Step 000000): Train loss 3.820, Val loss 3.462
Ep 1 (Step 000050): Train loss 0.396, Val loss 0.364
Ep 1 (Step 000100): Train loss 0.111, Val loss 0.229
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 2 (Step 000150): Train loss 0.135, Val loss 0.073
Ep 2 (Step 000200): Train loss 0.008, Val loss 0.052
Ep 2 (Step 000250): Train loss 0.021, Val loss 0.179
Training accuracy: 97.50% | Validation accuracy: 97.50%
Ep 3 (Step 000300): Train loss 0.096, Val loss 0.080
Ep 3 (Step 000350): Train loss 0.010, Val loss 0.116
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 4 (Step 000400): Train loss 0.003, Val loss 0.151
Ep 4 (Step 000450): Train loss 0.008, Val loss 0.077
Ep 4 (Step 000500): Train loss 0.001, Val loss 0.147
Training accuracy: 100.00% | Validation accuracy: 97.50%
```

```
Ep 5 (Step 000550): Train loss 0.007, Val loss 0.094
Ep 5 (Step 000600): Train loss 0.000, Val loss 0.056
Training accuracy: 100.00% | Validation accuracy: 97.50%

Training completed in 12.10 minutes.
```

Training the model with LoRA took longer than training it without LoRA (see chapter 6) because the LoRA layers introduce an additional computation during the forward pass. However, for larger models, where backpropagation becomes more costly, models typically train faster with LoRA than without it.

As we can see, the model received perfect training and very high validation accuracy. Let's also visualize the loss curves to better see whether the training has converged:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(
    epochs_tensor, examples_seen_tensor,
    train_losses, val_losses, label="loss"
)
```

Figure E.5 plots the results.

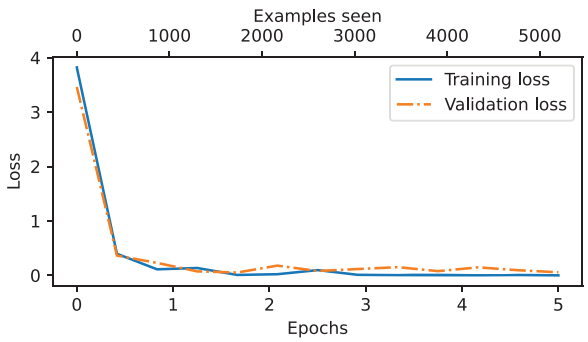


Figure E.5 The training and validation loss curves over six epochs for a machine learning model. Initially, both training and validation loss decrease sharply and then they level off, indicating the model is converging, which means that it is not expected to improve noticeably with further training.

In addition to evaluating the model based on the loss curves, let's also calculate the accuracies on the full training, validation, and test set (during the training, we approximated the training and validation set accuracies from five batches via the `eval_iter=5` setting):

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 100.00%
Validation accuracy: 96.64%
Test accuracy: 98.00%
```

These results show that the model performs well across training, validation, and test datasets. With a training accuracy of 100%, the model has perfectly learned the training data. However, the slightly lower validation and test accuracies (96.64% and 97.33%, respectively) suggest a small degree of overfitting, as the model does not generalize quite as well on unseen data compared to the training set. Overall, the results are very impressive, considering we fine-tuned only a relatively small number of model weights (2.7 million LoRA weights instead of the original 124 million model weights).

index

Symbols

Numerics

A

B

C

D

E

F

G

I

K

L

M

N

S

T

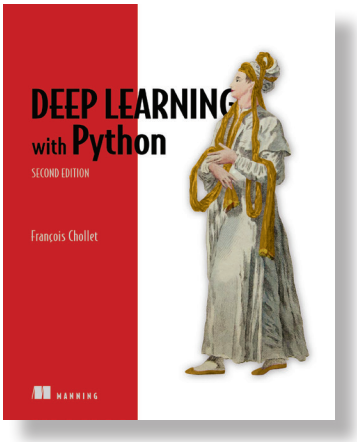
U

V

W

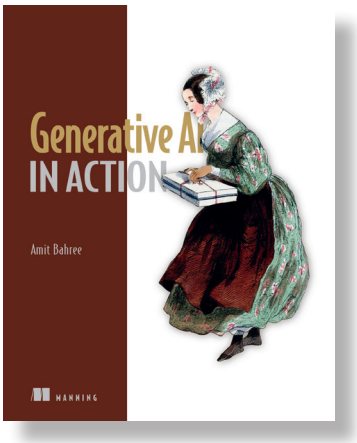
X

Z



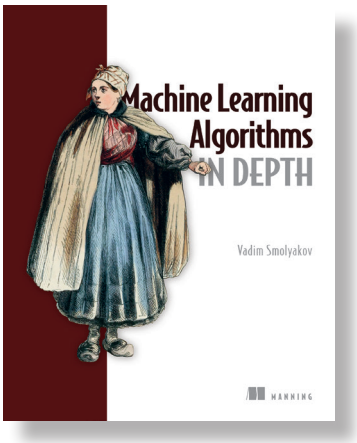
Deep Learning with Python, Second Edition
by Francois Chollet

ISBN 9781617296864
504 pages, $59.99
October 2021



Generative AI in Action
by Amit Bahree

ISBN 9781633436947
469 pages (estimated), $59.99
October 2024 (estimated)

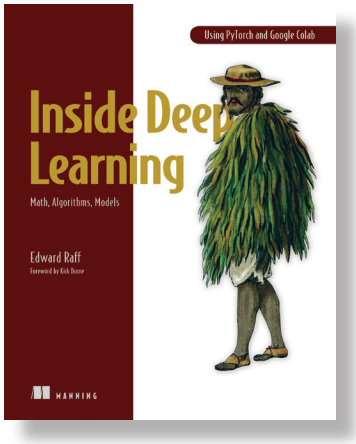


Machine Learning Algorithms in Depth
by Vadim Smolyakov

ISBN 9781633439214
328 pages, $79.99
July 2024

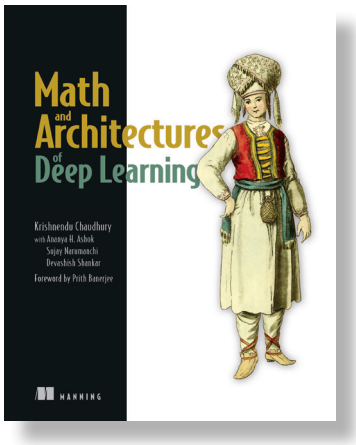
For ordering information, go to www.manning.com

RELATED MANNING TITLES



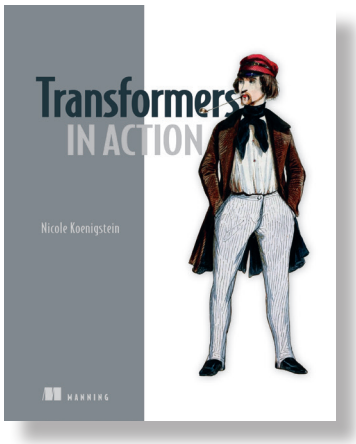
Inside Deep Learning
by Edward Raff
Foreword by Kirk Borne

ISBN 9781617298639
600 pages, $59.99
April 2022



Math and Architectures of Deep Learning
by Krishnendu Chaudhury
with Ananya H. Ashok, Sujay Narumanchi, Devashish Shankar
Foreword by Prith Banerjee

ISBN 9781617296482
552 pages, $69.99
April 2024



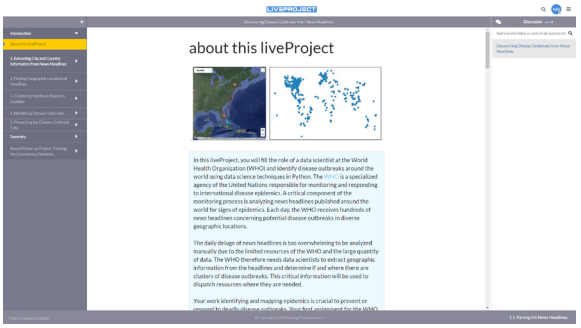
Transformers in Action
by Nicole Koenigstein

ISBN 9781633437883
393 pages (estimated), $59.99
February 2025 (estimated)

For ordering information, go to www.manning.com

MANNING

LIVEPROJECT



Hands-on projects for learning your way

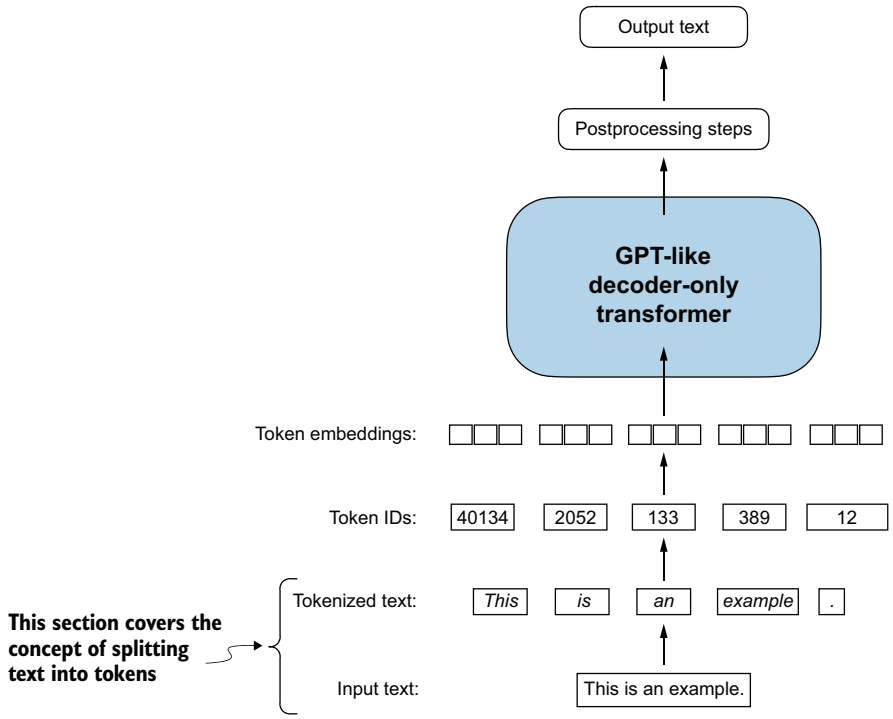
liveProjects are an exciting way to develop your skills that's just like learning on the job.

In a Manning liveProject, you tackle a real-world IT challenge and work out your own solutions. To make sure you succeed, you'll get 90 days of full and unlimited access to a hand-picked list of Manning book and video resources.

Here's how liveProject works:

- **Achievable milestones.** Each project is broken down into steps and sections so you can keep track of your progress.
- **Collaboration and advice.** Work with other liveProject participants through chat, working groups, and peer project reviews.
- **Compare your results.** See how your work shapes up against an expert implementation by the liveProject's creator.
- **Everything you need to succeed.** Datasets and carefully selected learning resources come bundled with every liveProject.
- **Build your portfolio.** All liveProjects teach skills that are in demand from industry. When you're finished, you'll have the satisfaction that comes with success and a real project to add to your portfolio.

Explore dozens of data, development, and cloud engineering liveProjects at www.manning.com!



A view of the text processing steps in the context of an LLM. The process starts with input text, which is broken down into tokens and then converted into numerical token IDs. These IDs are linked to token embeddings that serve as the input for the GPT model. The model processes these embeddings and generates output text. Finally, the output undergoes postprocessing steps to produce the final text. This flow illustrates the basic operations of tokenization, embedding, transformation, and postprocessing in a GPT model that is implemented from the ground up in this book.

AI

BUILD A Large Language Model (FROM SCRATCH)

Sebastian Raschka

Physicist Richard P. Feynman reportedly said, "I don't understand anything I can't build." Based on this same powerful principle, bestselling author Sebastian Raschka guides you step by step as you build a GPT-style LLM that you can run on your laptop. This is an engaging book that covers each stage of the process, from planning and coding to training and fine-tuning.

Build a Large Language Model (From Scratch) is a practical and eminently-satisfying hands-on journey into the foundations of generative AI. Without relying on any existing LLM libraries, you'll code a base model, evolve it into a text classifier, and ultimately create a chatbot that can follow your conversational instructions. And you'll really understand it because you built it yourself!

What's Inside

- Plan and code an LLM comparable to GPT-2
- Load pretrained weights
- Construct a complete training pipeline
- Fine-tune your LLM for text classification
- Develop LLMs that follow human instructions

Readers need intermediate Python skills and some knowledge of machine learning. The LLM you create will run on any modern laptop and can optionally utilize GPUs.

Sebastian Raschka is a Staff Research Engineer at Lightning AI, where he works on LLM research and develops open-source software.

The technical editor on this book was David Caswell.

For print book owners, all ebook formats are free:
https://www.manning.com/freebook

MANNING

"Truly inspirational! It motivates you to put your new skills into action."

—Benjamin Muskalla
Senior Engineer, GitHub

"The most understandable and comprehensive explanation of language models yet! Its unique and practical teaching style achieves a level of understanding you can't get any other way."

—Cameron Wolfe
Senior Scientist, Netflix

"Sebastian combines deep knowledge with practical engineering skills and a knack for making complex ideas simple. This is the guide you need!"

—Chip Huyen, author of *Designing Machine Learning Systems* and *AI Engineering*

"Definitive, up-to-date coverage. Highly recommended!"

—Dr. Vahid Mirjalili, Senior Data Scientist, FM Global

ISBN-13: 978-1-63343-716-6