6 Finetuning for Classification

This chapter covers

- Introducing different LLM finetuning approaches
- Preparing a dataset for text classification
- Modifying a pretrained LLM for finetuning
- Finetuning an LLM to identify spam messages
- Evaluating the accuracy of a finetuned LLM classifier
- Using a finetuned LLM to classify new data

In previous chapters, we coded the LLM architecture, pretrained it, and learned how to import pretrained weights from an external source, such as OpenAI, into our model. In this chapter, we are reaping the fruits of our labor by finetuning the LLM on a specific target task, such as classifying text, as illustrated in figure 6.1. The concrete example we will examine is classifying text messages as spam or not spam.

Figure 6.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset and finetuning it. This chapter focuses on finetuning a pretrained LLM as a classifier.

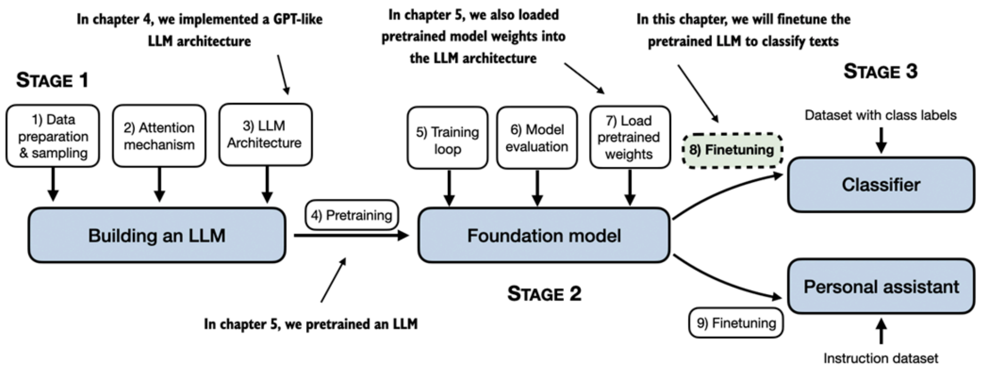
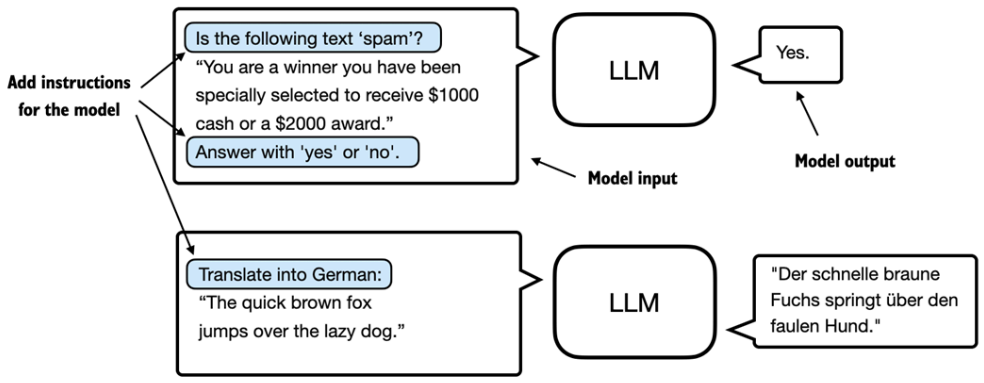

In chapter 4, we implemented a GPT-like LLM architecture
In chapter 5, we also loaded pretrained model weights into the LLM architecture
In this chapter, we will finetune the pretrained LLM to classify texts
STAGE 1
STAGE 3
1) Data preparation & sampling
2) Attention mechanism
3) LLM Architecture
5) Training loop
6) Model evaluation
7) Load pretrained weights
8) Finetuning
Dataset with class labels
Classifier
Building an LLM
4) Pretraining
Foundation model
Personal assistant
STAGE 2
In chapter 5, we pretrained an LLM
9) Finetuning
Instruction dataset

Figure 6.1 shows two main ways of finetuning an LLM: finetuning for classification (step 8) and finetuning an LLM to follow instructions (step 9). In the next section, we will discuss these two ways of finetuning in more detail.

6.1 Different categories of finetuning

The most common ways to finetune language models are *instruction-finetuning* and *classification-finetuning*. Instruction-finetuning involves training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts, as illustrated in figure 6.2.

Figure 6.2 Illustration of two different instruction-finetuning scenarios. At the top, the model is tasked with determining whether a given text is spam. At the bottom, the model is given an instruction on how to translate an English sentence into German.

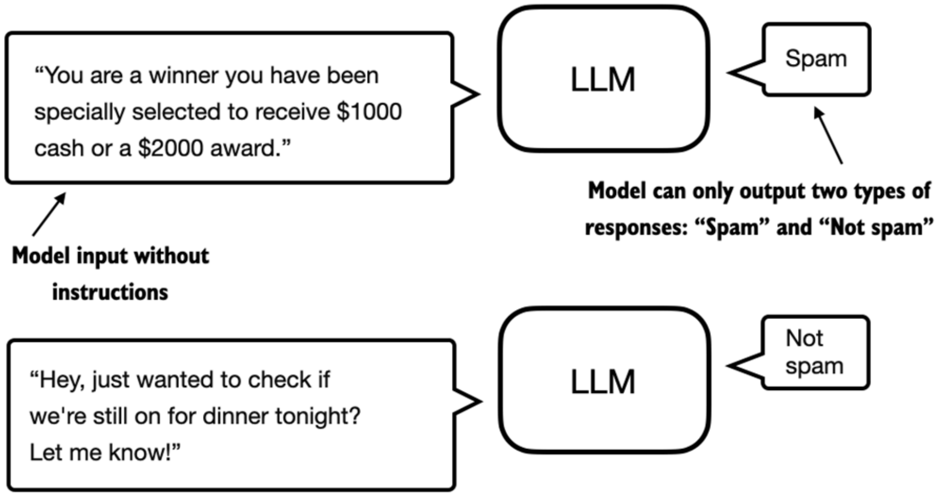

Add instructions for the model
Is the following text 'spam'?
"You are a winner you have been specially selected to receive $1000 cash or a $2000 award."
Answer with 'yes' or 'no'.
LLM
Yes.
Model input
Model output
Translate into German:
"The quick brown fox jumps over the lazy dog."
LLM
"Der schnelle braune Fuchs springt über den faulen Hund."

The next chapter will discuss instruction-finetuning, as illustrated in figure 6.2. Meanwhile, this chapter is centered on classification-finetuning, a concept you might already be acquainted with if you have a background in machine learning.

In classification-finetuning, the model is trained to recognize a specific set of class labels, such as "spam" and "not spam." Examples of classification tasks extend beyond large language models and email filtering; they include identifying different species of plants from images, categorizing news articles into topics like sports, politics, or technology, and distinguishing between benign and malignant tumors in medical imaging.

The key point is that a classification-finetuned model is restricted to predicting classes it has encountered during its training—for instance, it can determine whether something is "spam" or "not spam," as illustrated in figure 6.3, but it can't say anything else about the input text.

Figure 6.3 Illustration of a text classification scenario using an LLM. A model finetuned for spam classification does not require further instruction alongside the input. In contrast to an instruction-finetuned model, it can only respond with "spam" and "not spam."


"You are a winner you have been specially selected to receive $1000 cash or a $2000 award."
LLM
Spam
Model can only output two types of responses: "Spam" and "Not spam"
Model input without instructions
"Hey, just wanted to check if we're still on for dinner tonight? Let me know!"
LLM
Not spam

In contrast to the classification-finetuned model depicted in figure 6.3, an instruction-finetuned model typically has the capability to undertake a broader range of tasks. We can view a classification-finetuned model as highly specialized, and generally, it is easier to develop a specialized model than a generalist model that works well across various tasks.

Choosing the right approach

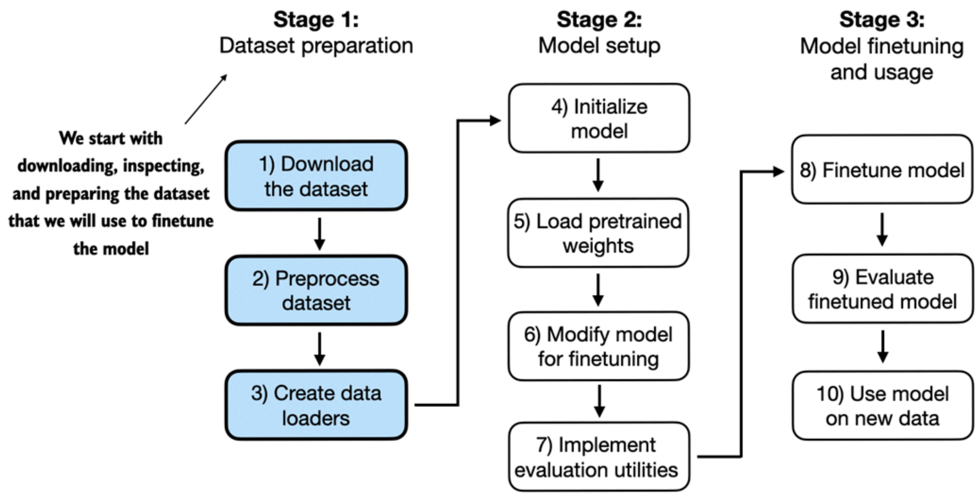
Instruction-finetuning improves a model's ability to understand and generate responses based on specific user instructions. Instruction-finetuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. Classification-finetuning, on the other hand, is ideal for projects requiring precise categorization of data into predefined classes, such as sentiment analysis or spam detection.

While instruction-finetuning is more versatile, it demands larger datasets and greater computational resources to develop models proficient in various tasks. In contrast, classification-finetuning requires less data and compute power, but its use is confined to the specific classes on which the model has been trained.

6.2 Preparing the dataset

In the remainder of this chapter, we will modify and classification-finetune the GPT model we implemented and pretrained in the previous chapters. We begin with downloading and preparing the dataset, as illustrated in figure 6.4.

Figure 6.4 Illustration of the three-stage process for classification-finetuning the LLM in this chapter. Stage 1 involves dataset preparation. Stage 2 focuses on model setup. Stage 3 covers the finetuning and evaluation of the model.



Stage 1:
Dataset preparation
Stage 2:
Model setup
Stage 3:
Model finetuning and usage
We start with downloading, inspecting, and preparing the dataset that we will use to finetune the model
1) Download the dataset
2) Preprocess dataset
3) Create data loaders
4) Initialize model
5) Load pretrained weights
6) Modify model for finetuning
7) Implement evaluation utilities
8) Finetune model
9) Evaluate finetuned model
10) Use model on new data

To provide an intuitive and useful example of classification-finetuning, we will work with a text message dataset that consists of spam and non-spam messages.

Note that these are text messages typically sent via phone, not email. However, the same steps also apply to email classification, and interested readers can find links to email spam classification datasets in the References section in appendix B.

The first step is to download the dataset via the following code:

Listing 6.1 Downloading and unzipping the dataset

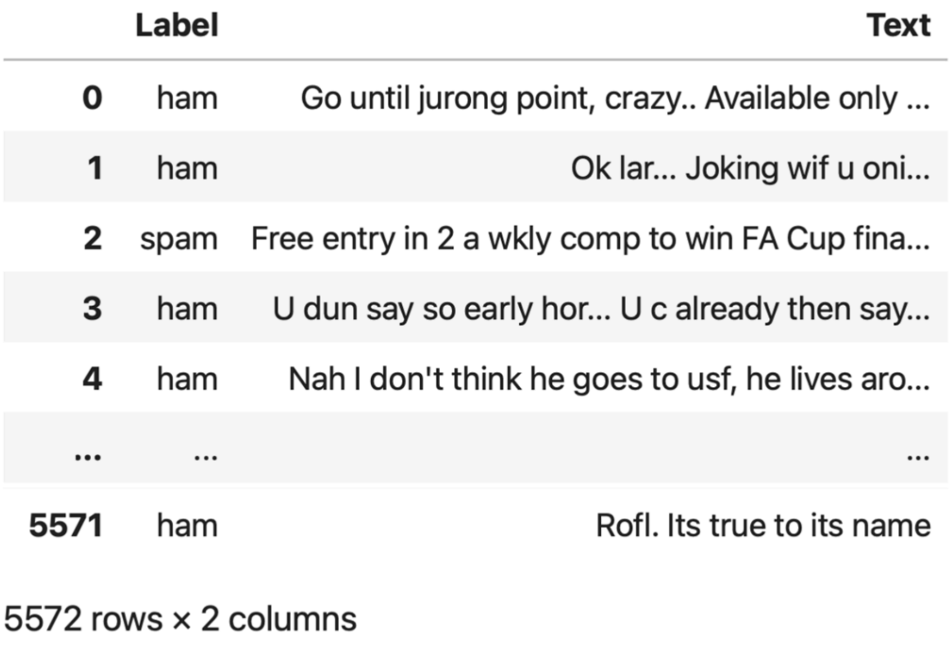
```python
import urllib.request
import zipfile
import os
from pathlib import Path

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"

def download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
        print(f"{data_file_path} already exists. Skipping download and extraction.")
        return

    with urllib.request.urlopen(url) as response:                                      A
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    with zipfile.ZipFile(zip_path, "r") as zip_ref:                                    B
        zip_ref.extractall(extracted_path)

    original_file_path = Path(extracted_path) / "SMSSpamCollection"                    C
    os.rename(original_file_path, data_file_path)
    print(f"File downloaded and saved as {data_file_path}")

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)
```

After executing the preceding code, the dataset is saved as a tab-separated text file, `SMSSpamCollection.tsv` , in the `sms_spam_collection` folder. We can load it into a pandas `DataFrame` as follows:

```python
import pandas as pd
df = pd.read_csv(data_file_path, sep="\t", header=None, names=["Label", "Text"])
df
```

The resulting data frame of the spam dataset is shown in figure 6.5.

Figure 6.5 Preview of the `SMSSpamCollection` dataset in a pandas `DataFrame`, showing class labels ("ham" or "spam") and corresponding text messages. The dataset consists of 5,572 rows (text messages and labels).



	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Let's examine the *class label distribution*:

```python
print(df["Label"].value_counts())
```

Executing the previous code, we find that the data contains "ham" (i.e., not spam) far more frequently than "spam":

```
Label
ham     4825
spam     747
Name: count, dtype: int64
```

For simplicity, and because we prefer a small dataset for educational purposes (which will facilitate faster fine-tuning of the large language model), we choose to undersample the dataset to include 747 instances from each class. While there are several other methods to handle class imbalances, these are beyond the scope of a book on large language models. Readers interested in exploring methods for dealing with imbalanced data can find additional information in the References section in appendix B.

We use the following code to undersample the dataset and create a balanced dataset:

Listing 6.2 Creating a balanced dataset

```python
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(num_spam, random_state=123)
    balanced_df = pd.concat([ham_subset, df[df["Label"] == "spam"]])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

After executing the previous code to balance the dataset, we can see that we now have equal amounts of spam and non-spam messages:

```
Label
ham     747
spam    747
Name: count, dtype: int64
```

Next, we convert the "string" class labels "ham" and "spam" into integer class labels 0 and 1, respectively:

```python
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

This process is similar to converting text into token IDs. However, instead of using the GPT vocabulary, which consists of more than 50,000 words, we are dealing with just two token IDs: 0 and 1.

We create a `random_split` function to split the dataset into three parts: 70% for training, 10% for validation, and 20% for testing. (These ratios are common in machine learning to train, adjust, and evaluate models.)

Listing 6.3 Splitting the dataset

```
def random_split(df, train_frac, validation_frac):          A
    df = df.sample(frac=1, random_state=123).reset_index(drop=True)
                                                               B
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)
                                                               C

    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df
                                                               D
train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
```

Additionally, we save the dataset as CSV (comma-separated value) files, which we can reuse later:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

In this section, we downloaded the dataset, balanced it, and split it into training and evaluation subsets. In the next section, we will set up the PyTorch data loaders that will be used to train the model.

6.3 Creating data loaders

In this section, we develop PyTorch data loaders that are conceptually similar to the ones we implemented in chapter 2.

Previously, in chapter 2, we utilized a sliding window technique to generate uniformly sized text chunks, which were then grouped into batches for more efficient model training. Each chunk functioned as an individual training instance.

However, in this chapter, we are working with a spam dataset that contains text messages of varying lengths. To batch these messages as we did with the text chunks in chapter 2, we have two primary options:

1. Truncate all messages to the length of the shortest message in the dataset or batch.
2. Pad all messages to the length of the longest message in the dataset or batch.

Option 1 is computationally cheaper, but it may result in significant information loss if shorter messages are much smaller than the average or longest messages, potentially reducing model performance. So, we opt for the second option, which preserves the entire content of all messages.

To implement option 2, where all messages are padded to the length of the longest message in the dataset, we add padding tokens to all shorter messages. For this purpose, we use `"<|endoftext|>"` as a padding token, as discussed in chapter 2.

However, instead of appending the string `"<|endoftext|>"` to each of the text messages directly, we can add the token ID corresponding to `"<|endoftext|>"` to the encoded text messages as illustrated in figure 6.6.

Figure 6.6 An illustration of the input text preparation process. First, each input text message is converted into a sequence of token IDs. Then, to ensure uniform sequence lengths, shorter sequences are padded with a padding token (in this case, token ID 50256) to match the length of the longest sequence.

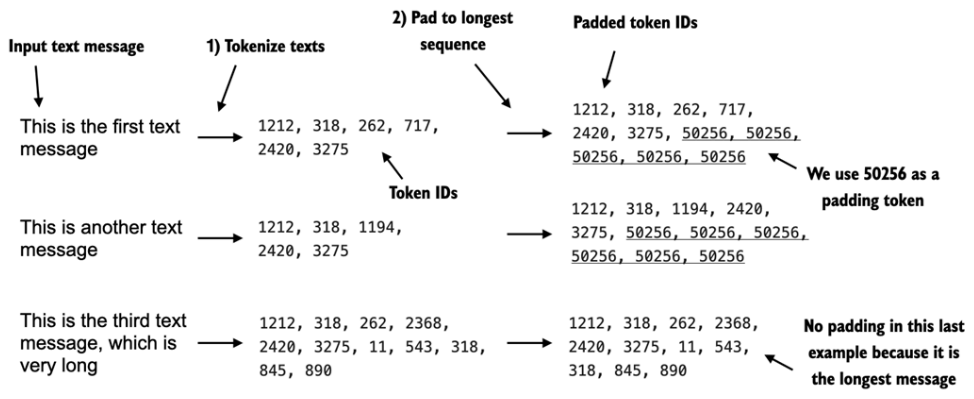


Figure 6.6 presumes that 50,256 is the token ID of the padding token `"<|endoftext|>"`. We can double-check that this is indeed the correct token ID by encoding the `"<|endoftext|>"` using the *GPT-2 tokenizer* from the `tiktoken` package that we used in previous chapters:

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

Executing the preceding code indeed returns `[50256]`.

As we have seen in chapter 2, we first need to implement a PyTorch `Dataset`, which specifies how the data is loaded and processed, before we can instantiate the data loaders.

For this purpose, we define the `SpamDataset` class, which implements the concepts illustrated in figure 6.6. This `SpamDataset` class handles several key tasks: it identifies the longest sequence in the training dataset, encodes the text messages, and ensures that all other sequences are padded with a *padding token* to match the length of the longest sequence.

Listing 6.4 Setting up a Pytorch Dataset class

```
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None, pad_token_id=50256):
        self.data = pd.read_csv(csv_file)
                                                                          A
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
                                                                          B
            self.encoded_texts = [
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]

                                                                          C

        self.encoded_texts = [
            encoded_text + [pad_token_id] * (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

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

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length
```

The `SpamDataset` class loads data from the CSV files we created earlier, tokenizes the text using the GPT-2 tokenizer from `tiktoken` and allows us to *pad* or *truncate* the sequences to a uniform length determined by either the longest sequence or a predefined maximum length. This ensures each input tensor is of the same size, which is necessary to create the batches in the training data loader we implement next:

```
train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)
```

Note that the longest sequence length is stored in the dataset's `max_length` attribute. If you are curious to see the number of tokens in the longest sequence, you can use the following code:

```
print(train_dataset.max_length)
```

The code outputs 120, showing that the longest sequence contains no more than 120 tokens, a common length for text messages. It's worth noting that the model can handle sequences of up to 1,024 tokens, given its context length limit. If your dataset includes longer texts, you can pass `max_length=1024` when creating the training dataset in the preceding code to ensure that the data does not exceed the model's supported input (context) length.

Next, we pad the validation and test sets to match the length of the longest training sequence. It's important to note that any validation and test set samples exceeding the length of the longest training example are truncated using `encoded_text[:self.max_length]` in the `SpamDataset` code we defined earlier. This truncation is optional; you could also set `max_length=None` for both validation and test sets, provided there are no sequences exceeding 1,024 tokens in these sets.

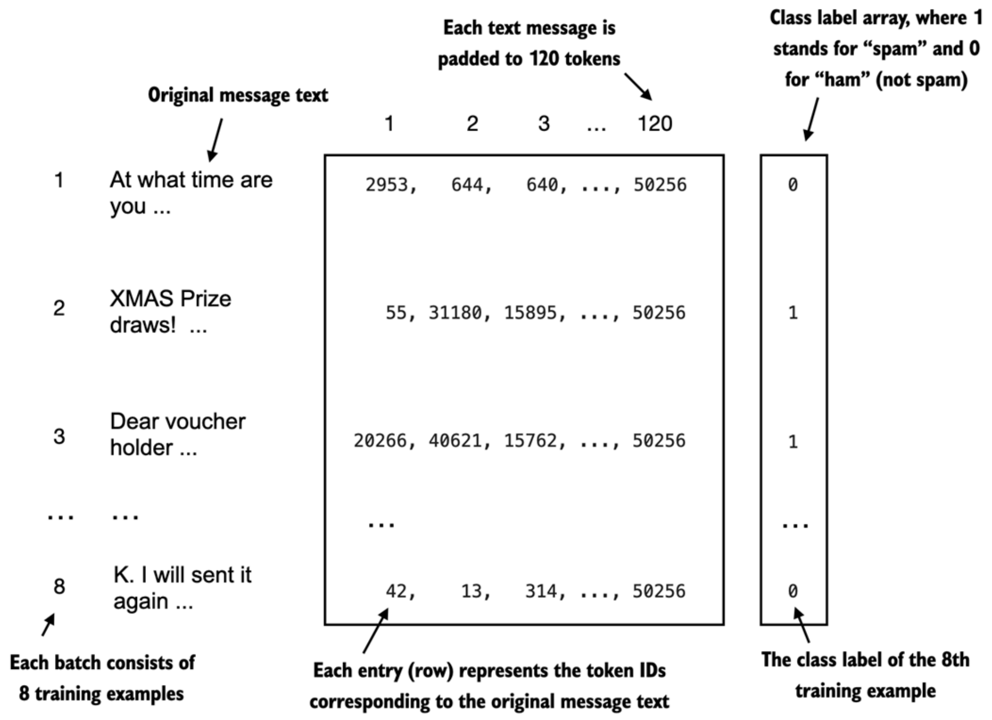
```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
```

Exercise 6.1 Increasing the context length

Pad the inputs to the maximum number of tokens the model supports and observe how it impacts the predictive performance.

Using the datasets as inputs, we can instantiate the data loaders similarly to what we did in chapter 2. However, in this case, the targets represent class labels rather than the next tokens in the text. For instance, choosing a batch size of 8, each batch will consist of 8 training examples of length 120 and the corresponding class label of each example, as illustrated in figure 6.7.

Figure 6.7 An illustration of a single training batch consisting of 8 text messages represented as token IDs. Each text message consists of 120 token IDs. In addition, a class label array stores the 8 class labels corresponding to the text messages, which can be either 0 (not spam) or 1 (spam).


Each text message is padded to 120 tokens
Class label array, where 1 stands for "spam" and 0 for "ham" (not spam)
Original message text
1 2 3 ... 120
1 At what time are you ...
2953, 644, 640, ..., 50256
0
2 XMAS Prize draws! ...
55, 31180, 15895, ..., 50256
1
3 Dear voucher holder ...
20266, 40621, 15762, ..., 50256
1
8 K. I will sent it again ...
42, 13, 314, ..., 50256
0
Each batch consists of 8 training examples
Each entry (row) represents the token IDs corresponding to the original message text
The class label of the 8th training example

The following code creates the training, validation, and test set data loaders that load the text messages and labels in batches of size 8, as illustrated in figure 6.7:

Listing 6.5 Creating PyTorch data loaders

```python
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,
)
```

To ensure that the data loaders are working and are indeed returning batches of the expected size, we iterate over the training loader and then print the tensor dimensions of the last batch:

```python
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 as follows:

```
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

As we can see, the input batches consist of 8 training examples with 120 tokens each, as expected. The label tensor stores the class labels corresponding to the 8 training examples.

Lastly, to get an idea of the dataset size, let's 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")
```

The number of batches in each dataset are as follows:

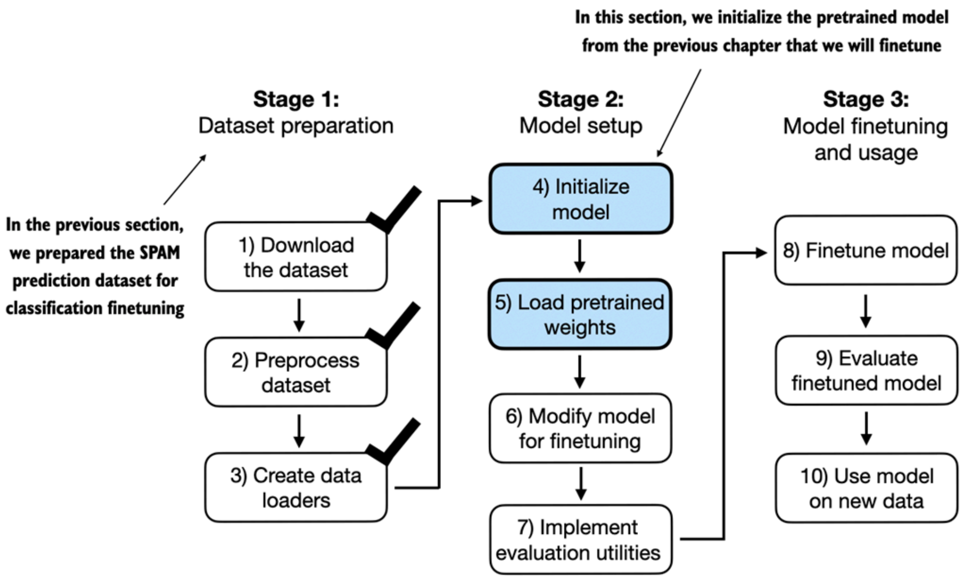
```
130 training batches
19 validation batches
38 test batches
```

This concludes the data preparation in this chapter. Next, we will prepare the model for finetuning.

6.4 Initializing a model with pretrained weights

In this section, we prepare the model we will use for the classification-finetuning to identify spam messages. We start with initializing the pretrained model we worked with in the previous chapter, as illustrated in figure 6.8.

Figure 6.8 Illustration of the three-stage process for classification-finetuning the LLM in this chapter. After completing stage 1, preparing the dataset, this section focuses on initializing the LLM we will finetune to classify spam messages.


In this section, we initialize the pretrained model from the previous chapter that we will finetune
Stage 1: Dataset preparation
Stage 2: Model setup
Stage 3: Model finetuning and usage
In the previous section, we prepared the SPAM prediction dataset for classification finetuning
1) Download the dataset
2) Preprocess dataset
3) Create data loaders
4) Initialize model
5) Load pretrained weights
6) Modify model for finetuning
7) Implement evaluation utilities
8) Finetune model
9) Evaluate finetuned model
10) Use model on new data

We start the model preparation process by reusing the configurations from chapter 5:

```python
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
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},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

assert train_dataset.max_length <= BASE_CONFIG["context_length"], (
    f"Dataset length {train_dataset.max_length} exceeds model's context "
    f"length {BASE_CONFIG['context_length']}. Reinitialize data sets with "
    f"`max_length={BASE_CONFIG['context_length']}`"
)
```

Next, we import the `download_and_load_gpt2` function from the `gpt_download.py` file we downloaded in chapter 5. Furthermore, we also reuse the `GPTModel` class and `load_weights_into_gpt` function from chapter 5 to load the downloaded weights into the GPT model:

Listing 6.6 Loading a pretrained GPT model

```python
from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

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 loading the model weights into the `GPTModel`, we use the text generation utility function from the previous chapters to ensure that the model generates coherent text:

```python
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))
```

As we can see based on the following output, the model generates coherent text, which is an indicator that the model weights have been loaded correctly:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Now, before we start finetuning the model as a spam classifier, let's see if the model can perhaps already classify spam messages by by prompting it with instructions:

```python
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

The model output is as follows:

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner you have
been specially selected to receive $1000 cash or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
```

Based on the output, it's apparent that the model struggles with following instructions.

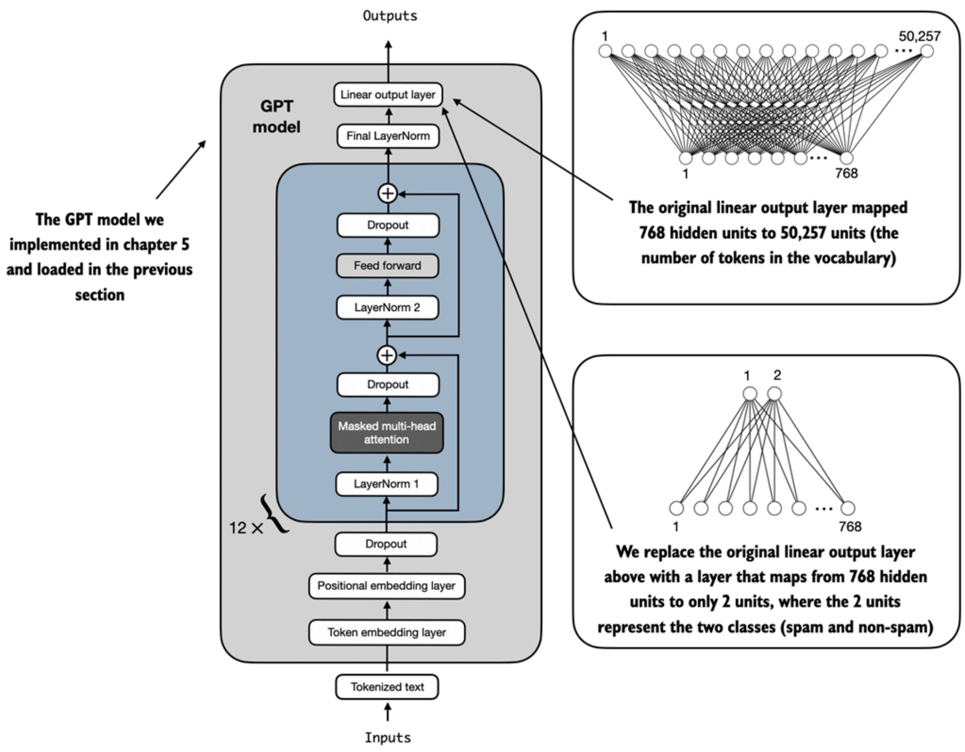
This is anticipated, as it has undergone only pretraining and lacks instruction-finetuning, which we will explore in the upcoming chapter.

The next section prepares the model for classification-finetuning.

6.5 Adding a classification head

In this section, we modify the pretrained large language model to prepare it for classification-finetuning. To do this, we replace the original output layer, which maps the hidden representation to a vocabulary of 50,257, with a smaller output layer that maps to two classes: 0 ("not spam") and 1 ("spam"), as shown in figure 6.9.

Figure 6.9 This figure illustrates adapting a GPT model for spam classification by altering its architecture. Initially, the model's linear output layer mapped 768 hidden units to a vocabulary of 50,257 tokens. For spam detection, this layer is replaced with a new output layer that maps the same 768 hidden units to just two classes, representing "spam" and "not spam."


Outputs
GPT model
Linear output layer
Final LayerNorm
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
12 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Inputs
The GPT model we implemented in chapter 5 and loaded in the previous section
1
50,257
1
768
The original linear output layer mapped 768 hidden units to 50,257 units (the number of tokens in the vocabulary)
1
2
1
768
We replace the original linear output layer above with a layer that maps from 768 hidden units to only 2 units, where the 2 units represent the two classes (spam and non-spam)

As shown in figure 6.9, we use the same model as in previous chapters except for replacing the output layer.

Output layer nodes

We could technically use a single output node since we are dealing with a binary classification task. However, this would require modifying the loss function, as discussed in an article in the Reference section in appendix B. Therefore, we choose a more general approach where the number of output nodes matches the number of classes. For example, for a 3-class problem, such as classifying news articles as "Technology", "Sports", or "Politics", we would use three output nodes, and so forth.

Before we attempt the modification illustrated in figure 6.9, let's print the model architecture via `print(model`), which prints the following:

```
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): Linear(in_features=768, out_features=768, bias=True)
        (W_key): Linear(in_features=768, out_features=768, bias=True)
        (W_value): Linear(in_features=768, out_features=768, bias=True)
        (out_proj): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU()
          (2): Linear(in_features=3072, out_features=768, bias=True)
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
  )
  (final_norm): LayerNorm()
  (out_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

Above, we can see the architecture we implemented in chapter 4 neatly laid out. As discussed in chapter 4, the `GPTModel` consists of embedding layers followed by 12 identical *transformer blocks* (only the last block is shown for brevity), followed by a final `LayerNorm` and the output layer, `out_head` .

Next, we replace the `out_head` with a new output layer, as illustrated in figure 6.9, that we will finetune.

Finetuning selected layers versus all layers

Since we start with a pretrained model, it's not necessary to finetune all model layers. This is because, in neural network-based language models, the lower layers generally capture basic language structures and semantics that are applicable across a wide range of tasks and datasets. So, finetuning only the last layers (layers near the output), which are more specific to nuanced linguistic patterns and task-specific features, can often be sufficient to adapt the model to new tasks. A nice side effect is that it is computationally more efficient to finetune only a small number of layers. Interested readers can find more information, including experiments, on which layers to finetune in the References section for this chapter in appendix B.

To get the model ready for classification-finetuning, we first *freeze* the model, meaning that we make all layers non-trainable:

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

Then, as shown in figure 6.9, we replace the output layer (`model.out_head`), which originally maps the layer inputs to 50,257 dimensions (the size of the vocabulary):

Listing 6.7 Adding a classification layer

```
torch.manual_seed(123)
num_classes = 2
model.out_head
= torch.nn.Linear(

in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

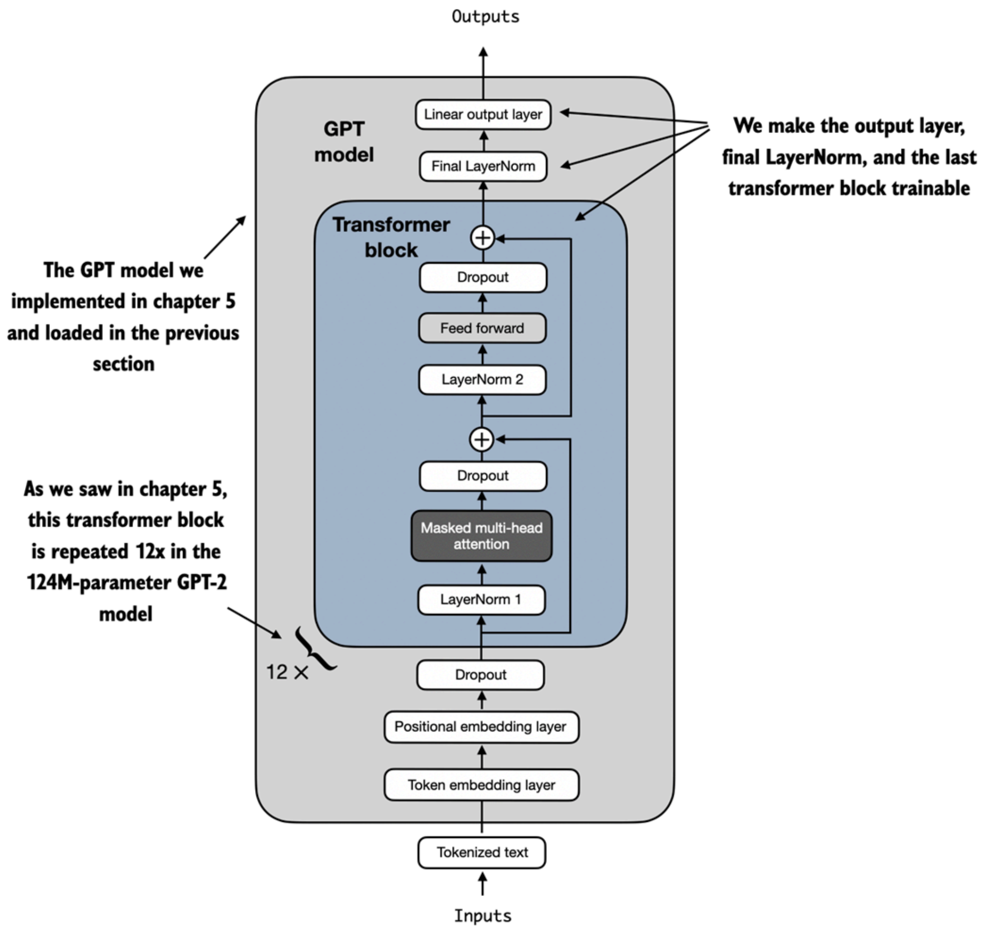
Note that in the preceding code, we use `BASE_CONFIG["emb_dim"]` , which is equal to 768 in the `"gpt2-small (124M)"` model, to keep the code below more general. This means we can also use the same code to work with the larger GPT-2 model variants.

This new `model.out_head` output layer has its `requires_grad` attribute set to `True` by default, which means that it's the only layer in the model that will be updated during training.

Technically, training the output layer we just added is sufficient. However, as I found in experiments, finetuning additional layers can noticeably improve the predictive performance of the finetuned model. (For more details, refer to the References in appendix C.)

Additionally, we configure the last transformer block and the final `LayerNorm` module, which connects this block to the output layer, to be trainable, as depicted in figure 6.10.

Figure 6.10 The GPT model we developed in earlier chapters, which we loaded previously, includes 12 repeated transformer blocks. Alongside the output layer, we set the final `LayerNorm` and the last transformer block as trainable, while the remaining 11 transformer blocks and the embedding layers are kept non-trainable.



Outputs
GPT model
Linear output layer
Final LayerNorm
We make the output layer, final LayerNorm, and the last transformer block trainable
Transformer block
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
The GPT model we implemented in chapter 5 and loaded in the previous section
As we saw in chapter 5, this transformer block is repeated 12x in the 124M-parameter GPT-2 model
12 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Inputs

To make the final `LayerNorm` and last transformer block trainable, as illustrated in figure 6.10, we set their respective `requires_grad` to `True` :

```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Exercise 6.2 Finetuning the whole model

Instead of finetuning just the final transformer block, finetune the entire model and assess the impact on predictive performance.

Even though we added a new output layer and marked certain layers as trainable or non-trainable, we can still use this model in a similar way to previous chapters. For instance, we can feed it an example text identical to how we have done it in earlier chapters. For example, consider the following example text:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape) # shape: (batch_size, num_tokens)
```

As the print output shows, the preceding code encodes the inputs into a tensor consisting of 4 input tokens:

```
Inputs: tensor([[5211,  345,  423,  640]])
Inputs dimensions: torch.Size([1, 4])
```

Then, we can pass the encoded token IDs to the model as usual:

```
with torch.no_grad():
    outputs = model(inputs)
print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape) # shape: (batch_size, num_tokens,
num_classes)
```

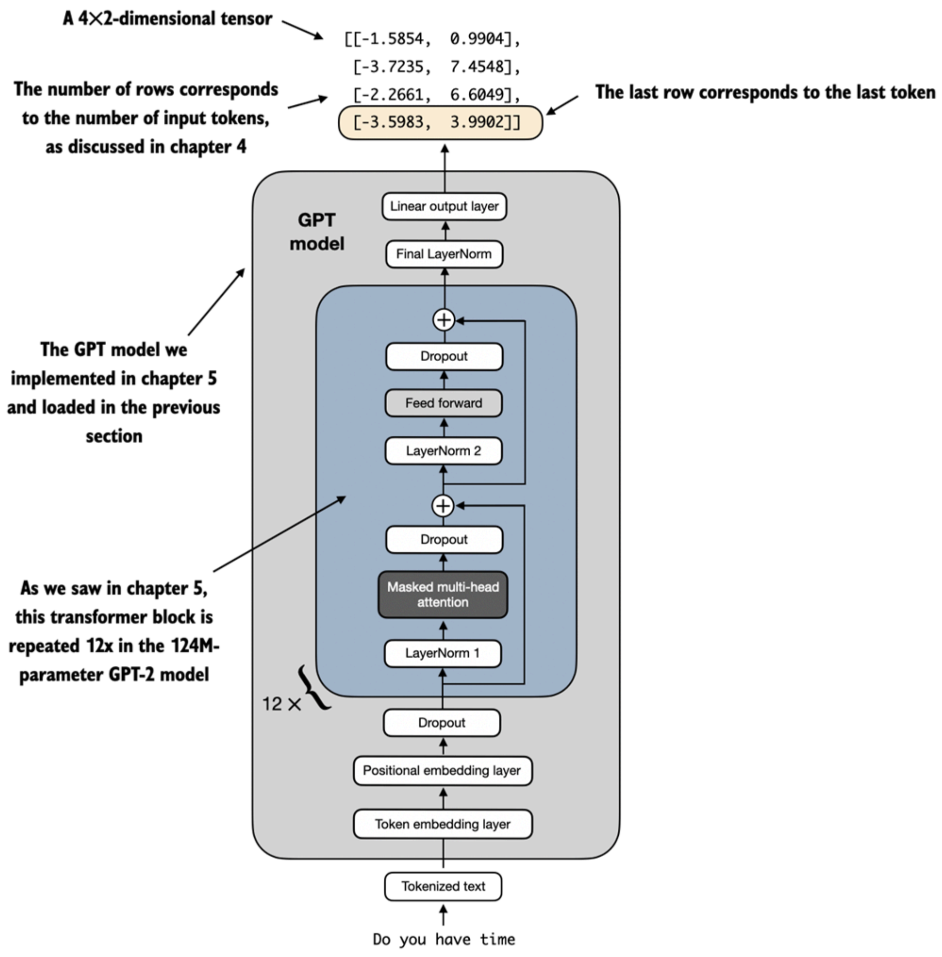
The output tensor looks like as follows:

```
Outputs:
 tensor([[[-1.5854,  0.9904],
          [-3.7235,  7.4548],
          [-2.2661,  6.6049],
          [-3.5983,  3.9902]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

In chapters 4 and 5, a similar input would have produced an output tensor of [1, 4, 50257], where 50,257 represents the vocabulary size. As in previous chapters, the number of output rows corresponds to the number of input tokens (in this case, 4). However, each output's embedding dimension (the number of columns) is now reduced to 2 instead of 50,257 since we replaced the output layer of the model.

Remember that we are interested in finetuning this model so that it returns a class label that indicates whether a model input is spam or not spam. To achieve this, we don't need to finetune all 4 output rows but can focus on a single output token. In particular, we will focus on the last row corresponding to the last output token, as illustrated in figure 6.11.

Figure 6.11 An illustration of the GPT model with a 4-token example input and output. The output tensor consists of 2 columns due to the modified output layer. We are only focusing on the last row corresponding to the last token when finetuning the model for spam classification.


A 4×2-dimensional tensor
[[-1.5854, 0.9904],
[-3.7235, 7.4548],
[-2.2661, 6.6049],
[-3.5983, 3.9902]]
The number of rows corresponds to the number of input tokens, as discussed in chapter 4
The last row corresponds to the last token
GPT model
Linear output layer
Final LayerNorm
The GPT model we implemented in chapter 5 and loaded in the previous section
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
As we saw in chapter 5, this transformer block is repeated 12x in the 124M-parameter GPT-2 model
12 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Do you have time

```
To extract the last output token, illustrated in figure 6.11, from the output tensor, we
use the following code:
print("Last output token:", outputs[:, -1, :])
```

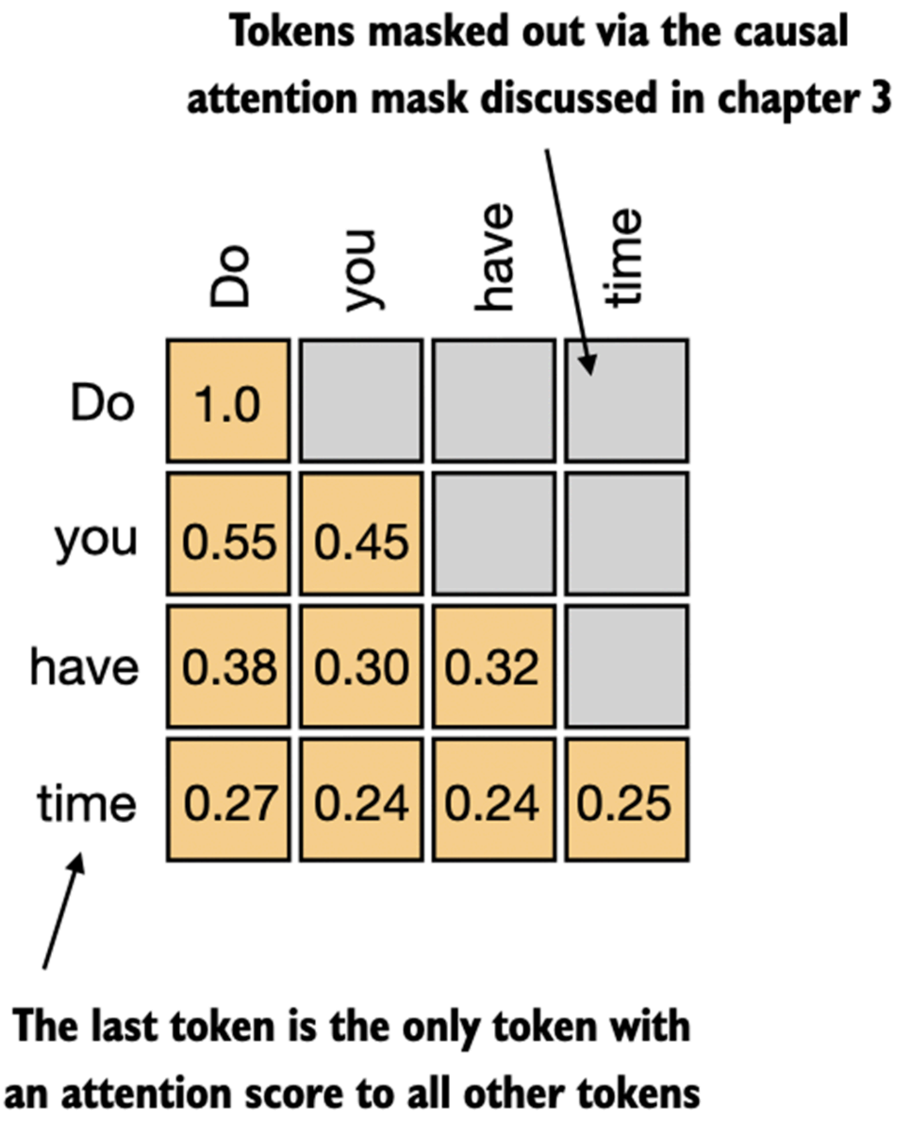
This prints the following:

```
Last output token: tensor([[-3.5983,  3.9902]])
```

Before we proceed to the next section, let's recap our discussion. We will focus on converting the values into a class-label prediction. But first, let's understand why we are particularly interested in the last output token, and not the 1st, 2nd, or 3rd output token.

In chapter 3, we explored the attention mechanism, which establishes a relationship between each input token and every other input token. Subsequently, we introduced the concept of a *causal attention mask*, commonly used in GPT-like models. This mask restricts a token's focus to only its current position and those before it, ensuring that each token can only be influenced by itself and preceding tokens, as illustrated in figure 6.12.

Figure 6.12 Illustration of the causal attention mechanism as discussed in chapter 3, where the attention scores between input tokens are displayed in a matrix format. The empty cells indicate masked positions due to the causal attention mask, preventing tokens from attending to future tokens. The values in the cells represent attention scores, with the last token, "time," being the only one that computes attention scores for all preceding tokens.


Tokens masked out via the causal attention mask discussed in chapter 3
Do
you
have
time
Do
you
have
time
1.0
0.55
0.45
0.38
0.30
0.32
0.27
0.24
0.24
0.25
The last token is the only token with an attention score to all other tokens

Given the causal attention mask setup shown in figure 6.12, the last token in a sequence accumulates the most information since it is the only token with access to data from all the previous tokens. Therefore, in our spam classification task, we focus on this last token during the finetuning process.

Having modified the model, the next section will detail the process of transforming the last token into class label predictions and calculate the model's initial prediction accuracy. Following this, we will finetune the model for the spam classification task in the subsequent section.

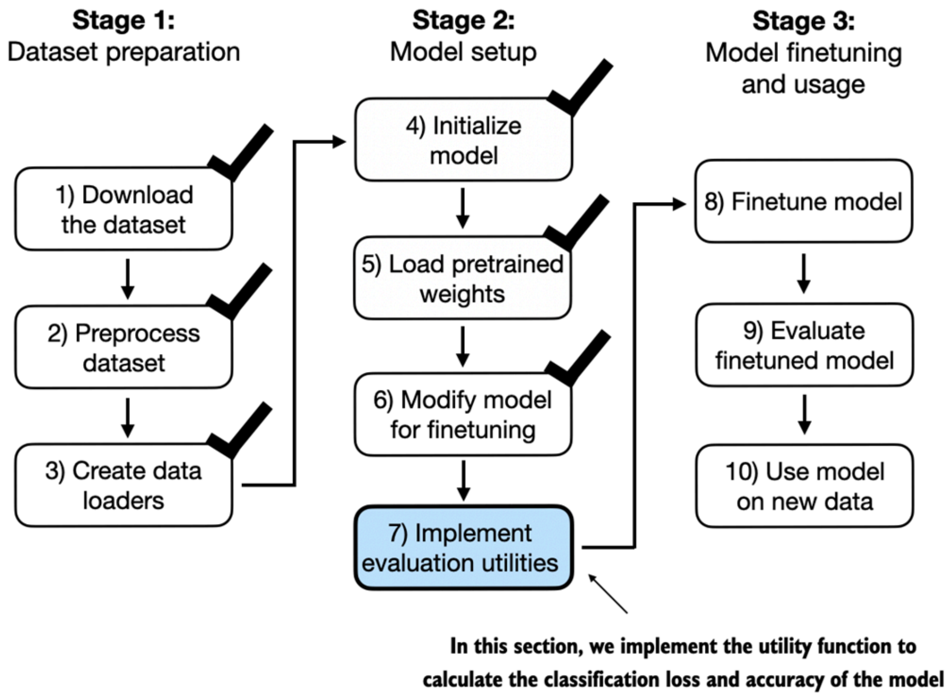
Exercise 6.3 Finetuning the first versus last token

Rather than finetuning the last output token, try finetuning the first output token and observe the changes in predictive performance when finetuning the model in later sections.

6.6 Calculating the classification loss and accuracy

So far in this chapter, we have prepared the dataset, loaded a pretrained model, and modified it for classification-finetuning. Before we proceed with the finetuning itself, only one small part remains: implementing the model evaluation functions used during finetuning, as illustrated in figure 6.13. We will tackle this in this section.

Figure 6.13 Illustration of the three-stage process for classification-finetuning the LLM in this chapter. This section implements the last step of stage 2, implementing the functions to evaluate the model's performance to classify spam messages before, during, and after the finetuning.

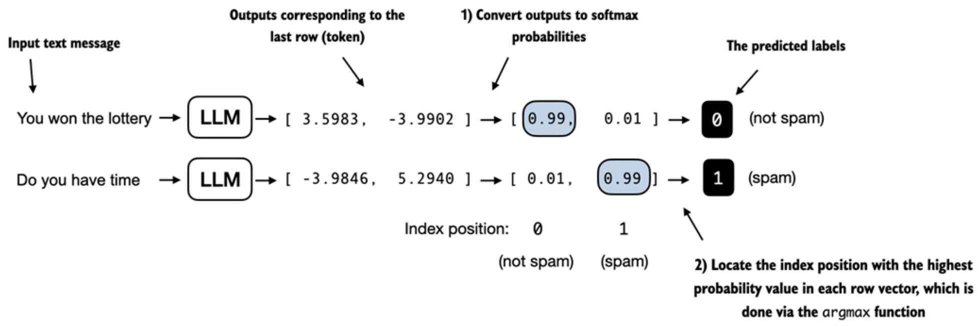


Stage 1:
Dataset preparation
Stage 2:
Model setup
Stage 3:
Model finetuning and usage
1) Download the dataset
2) Preprocess dataset
3) Create data loaders
4) Initialize model
5) Load pretrained weights
6) Modify model for finetuning
7) Implement evaluation utilities
8) Finetune model
9) Evaluate finetuned model
10) Use model on new data
In this section, we implement the utility function to calculate the classification loss and accuracy of the model

Before implementing the evaluation utilities, let's briefly discuss how we convert the model outputs into class label predictions.

In the previous chapter, we computed the token ID of the next token generated by the LLM by converting the 50,257 outputs into probabilities via the *softmax* function and then returning the position of the highest probability via the *argmax* function. In this chapter, we take the same approach to calculate whether the model outputs a "spam" or "not spam" prediction for a given input, as shown in figure 6.14, with the only difference being that we work with 2-dimensional instead of 50,257-dimensional outputs.

Figure 6.14. The model outputs corresponding to the last token are converted into probability scores for each input text. Then, the class labels are obtained by looking up the index position of the highest probability score. Note that the model predicts the spam labels incorrectly because it has not yet been trained.



Input text message
Outputs corresponding to the last row (token)
1) Convert outputs to softmax probabilities
The predicted labels
You won the lottery
LLM
[3.5983, -3.9902]
[0.99, 0.01]
0 (not spam)
Do you have time
LLM
[-3.9846, 5.2940]
[0.01, 0.99]
1 (spam)
Index position: 0 1
(not spam) (spam)
2) Locate the index position with the highest probability value in each row vector, which is done via the argmax function

To illustrate figure 6.14 with a concrete example, let's consider the last token output from the previous section:

```
print("Last output token:", outputs[:, -1, :])
```

The values of the tensor corresponding to the last token are as follows:

```
Last output token: tensor([[-3.5983,  3.9902]])
```

We can obtain the class label via the following code:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

In this case, the code returns 1, meaning the model predicts that the input text is "spam." Using the softmax function here is optional because the largest outputs directly correspond to the highest probability scores, as mentioned in chapter 5. Hence, we can simplify the code as follows, without using `softmax` :

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

This concept can be used to compute the so-called classification accuracy, which measures the percentage of correct predictions across a dataset.

To determine the classification accuracy, we apply the argmax-based prediction code to all examples in the dataset and calculate the proportion of correct predictions by defining a `calc_accuracy_loader` function:

Listing 6.8 Calculating the classification accuracy

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch, target_batch = input_batch.to(device), target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]  #A
            predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (predicted_labels == target_batch).sum().item()
        else:
            break
    return correct_predictions / num_examples
```

Let's use the function to determine the classification accuracies across various datasets estimated from 10 batches for efficiency:

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

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}%")
```

Via the `device` setting, the model automatically runs on a GPU if a GPU with Nvidia CUDA support is available and otherwise runs on a CPU. The output is as follows:

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

As we can see, the prediction accuracies are near a random prediction, which would be 50% in this case. To improve the prediction accuracies, we need to finetune the model.

However, before we begin finetuning the model, we need to define the loss function that we will optimize during the training process. Our objective is to maximize the spam classification accuracy of the model, which means that the preceding code should output the correct class labels: `0` for non-spam and `1` for spam texts.

However, classification accuracy is not a differentiable function, so we use cross entropy loss as a proxy to maximize accuracy. This is the same cross entropy loss discussed in chapter 5.

Accordingly, the `calc_loss_batch` function remains the same as in chapter 5, with one adjustment: we focus on optimizing only the last token, `model(input_batch)[:, -1, :]`, rather than all tokens, `model(input_batch)`:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch, target_batch = input_batch.to(device), target_batch.to(device)
    logits = model(input_batch)[:, -1, :]  # Logits of last output token
    loss = torch.nn.functional.cross_entropy(logits, target_batch)
    return loss
```

We use the `calc_loss_batch` function to compute the loss for a single batch obtained from the previously defined data loaders. To calculate the loss for all batches in a data loader, we define the `calc_loss_loader` function, which is identical to the one described in chapter 5:

Listing 6.9 Calculating the classification loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else: #A
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
Similar to calculating the training accuracy, we now compute the initial loss for each
data set:
with torch.no_grad(): #B
    train_loss = calc_loss_loader(train_loader, model, device, num_batches=5)
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
```

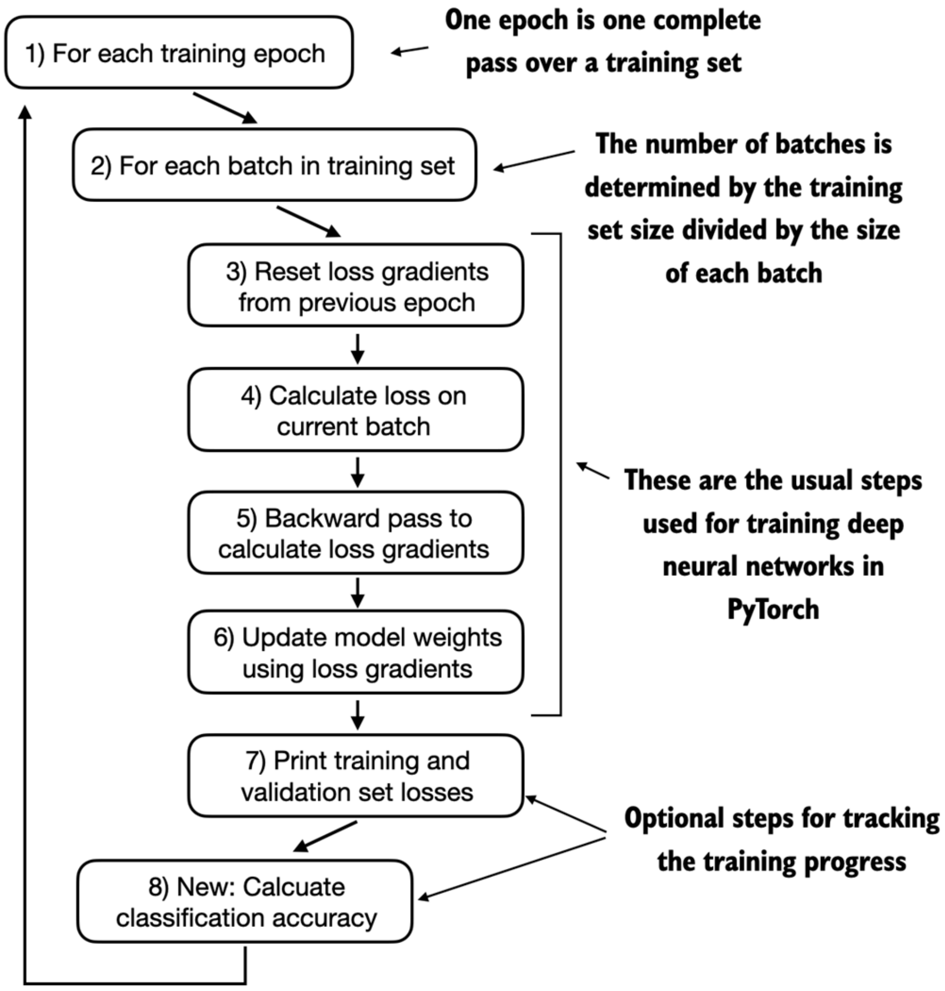
```
print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
The initial loss values are as follows:
Training loss: 3.095
Validation loss: 2.583
Test loss: 2.322
```

In the next section, we will implement a training function to finetune the model, which means adjusting the model to minimize the training set loss. Minimizing the training set loss will help increase the classification accuracy, our overall goal.

6.7 Finetuning the model on supervised data

In this section, we define and use the training function to finetune the pretrained LLM and improve its spam classification accuracy. The training loop, illustrated in figure 6.15, is the same overall training loop we used in chapter 5, with the only difference being that we calculate the classification accuracy instead of generating a sample text for evaluating the model.

Figure 6.15 A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights to minimize the training set loss.



1) For each training epoch
One epoch is one complete pass over a training set
2) For each batch in training set
The number of batches is determined by the training set size divided by the size of each batch
3) Reset loss gradients from previous epoch
4) Calculate loss on current batch
5) Backward pass to calculate loss gradients
6) Update model weights using loss gradients
These are the usual steps used for training deep neural networks in PyTorch
7) Print training and validation set losses
8) New: Calcuate classification accuracy
Optional steps for tracking the training progress

The training function implementing the concepts shown in figure 6.15 also closely mirrors the `train_model_simple` function used for pretraining the model in chapter 5.

The only two distinctions are that we now track the number of training examples seen (`examples_seen`) instead of the number of tokens, and we calculate the accuracy after each epoch instead of printing a sample text:

Listing 6.10 Finetuning the model to classify spam

```
def train_classifier_simple(model, train_loader, val_loader, optimizer, device,
num_epochs, eval_freq, eval_iter, tokenizer):
    # Initialize lists to track losses and examples seen
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1

    # Main training loop
    for epoch in range(num_epochs):
        model.train()  #A

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() #B
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward() #C
            optimizer.step() #D
            examples_seen += input_batch.shape[0] #E
            global_step += 1

            #F
            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)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")

        #G
        train_accuracy = calc_accuracy_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_accuracy = calc_accuracy_loader(
            val_loader, model, device, num_batches=eval_iter
        )

        print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
        print(f"Validation accuracy: {val_accuracy*100:.2f}%")
        train_accs.append(train_accuracy)
        val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen
```

The `evaluate_model` function used in the preceding `train_classifier_simple` is identical the one we used in chapter 5:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device,
num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device, num_batches=eval_iter)
    model.train()
    return train_loss, val_loss
```

Next, we initialize the optimizer, set the number of training epochs, and initiate the training using the `train_classifier_simple` function. We will discuss the choice of the the number of training epochs after we evaluated the results. The training takes about 6 minutes on an M3 MacBook Air laptop computer and less than half a minute on a V100 or A100 GPU:

```
import time

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 as follows:

```
Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 5.65 minutes.
```

Similar to chapter 5, we then use matplotlib to plot the loss function for the training and validation set:

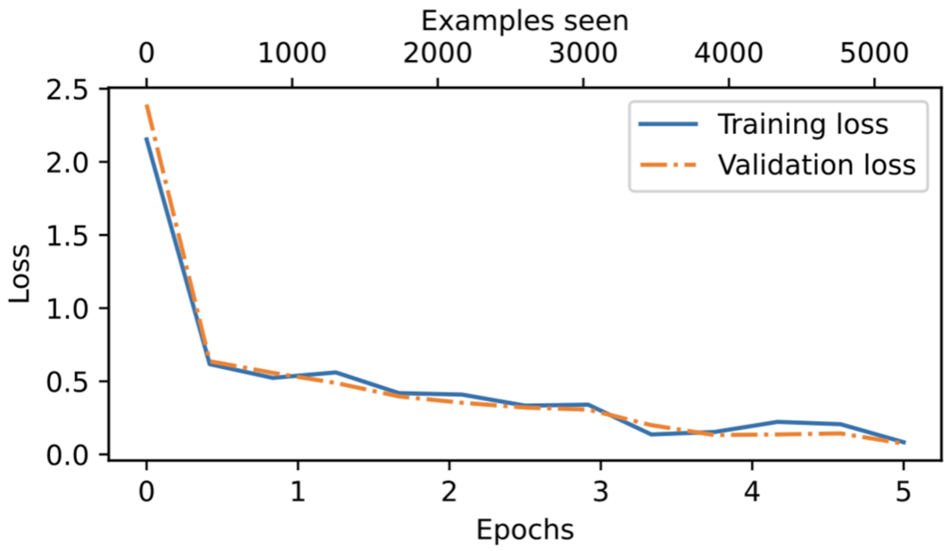
Listing 6.11 Plotting the classification loss

```
import matplotlib.pyplot as plt

def plot_values(epochs_seen, examples_seen, train_values, val_values, label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    #A
    ax1.plot(epochs_seen, train_values, label=f"Training {label}")
    ax1.plot(epochs_seen, val_values, linestyle="-.", label=f"Validation {label}")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    #B
    ax2 = ax1.twiny()
    ax2.plot(examples_seen, train_values, alpha=0)  # Invisible plot for aligning ticks
    ax2.set_xlabel("Examples seen")

    fig.tight_layout()  #C
    plt.savefig(f"{label}-plot.pdf")
    plt.show()
```

```
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)
```

The resulting loss curves are shown in the plot in figure 6.16.

Figure 6.16 This graph shows the model's training and validation loss over the five training epochs. The training loss, represented by the solid line, and the validation loss, represented by the dashed line, both sharply decline in the first epoch and gradually stabilize towards the fifth epoch. This pattern indicates good learning progress and suggests that the model learned from the training data while generalizing well to the unseen validation data.



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).

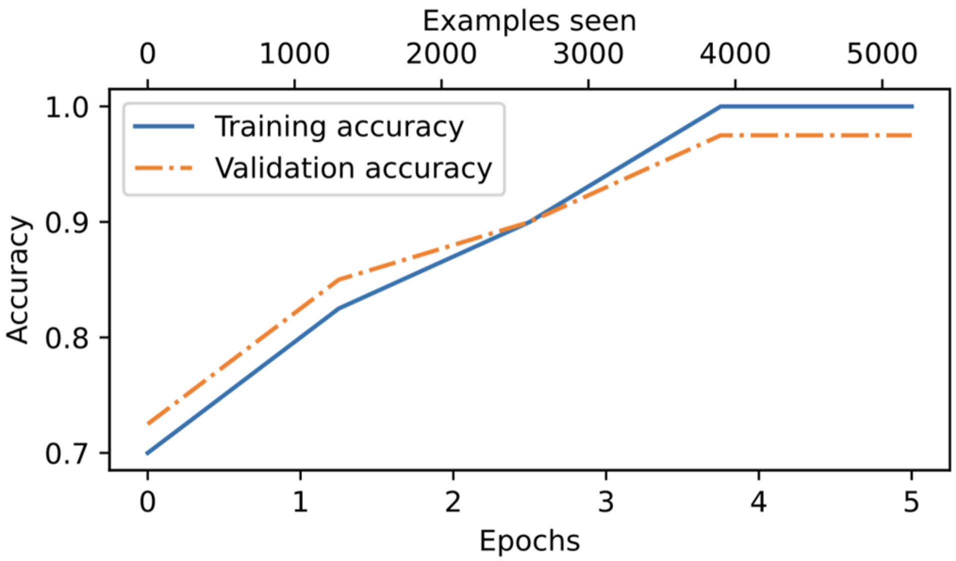
Choosing the number of epochs

Earlier, when we initiated the training, we set the number of epochs to 5. The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation. An epoch number of 5 is usually a good starting point. If the model overfits after the first few epochs, as a loss plot as shown in Figure 6.16 could indicate, we may need to reduce the number of epochs. Conversely, if the trendline suggests that the validation loss could improve with further training, we should increase the number of epochs. In this concrete case 5 epochs was a reasonable number as there is no sign of early overfitting, and the validation loss is close to 0.

Using the same `plot_values` function, let's now also 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")
The resulting accuracy graphs are shown in figure 6.17.
```

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 much.



Based on the accuracy plot in figure 6.17, the model achieves a relatively high training and validation accuracy after epochs 4 and 5.

However, it's important to note that we previously set `eval_iter=5` when using the `train_classifier_simple` function, which means our estimations of training and validation performance were based on only 5 batches for efficiency during training.

Now, we will 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 as follows:

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

The training and test set performances are almost identical.

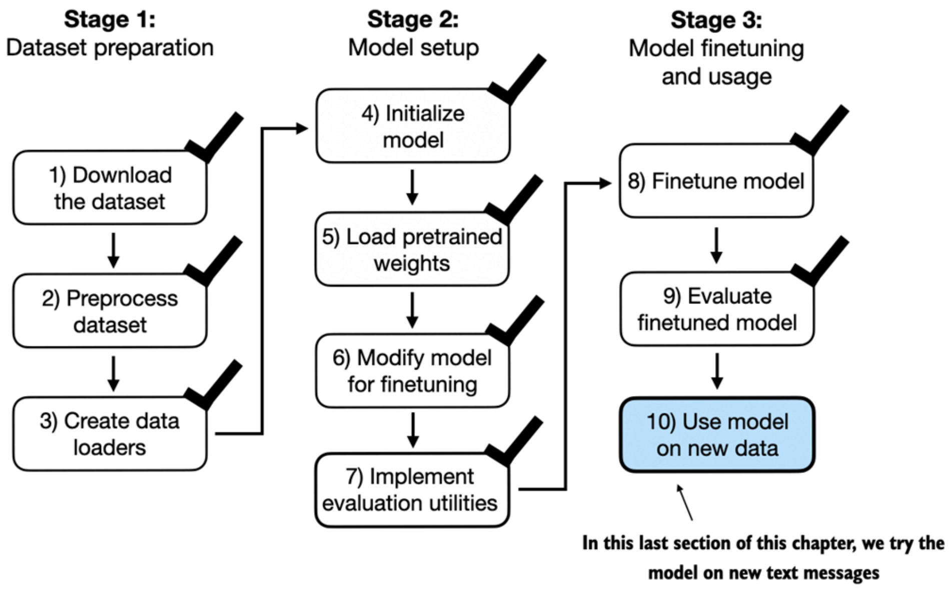
A slight discrepancy between the training and test set accuracies suggests minimal overfitting of the training data. Typically, the validation set accuracy is somewhat higher than the test set accuracy because the model development often involves tuning hyperparameters to perform well on the validation 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

After finetuning and evaluating the model in the previous sections, we are now in the final stage of this chapter, as illustrated in figure 6.18: using the model to classify spam messages.

Figure 6.18 Illustration of the three-stage process for classification-finetuning the LLM in this chapter. This section implements the final step of stage 3, using the finetuned model to classify new spam messages.


Stage 1:
Dataset preparation
Stage 2:
Model setup
Stage 3:
Model finetuning and usage
1) Download the dataset
2) Preprocess dataset
3) Create data loaders
4) Initialize model
5) Load pretrained weights
6) Modify model for finetuning
7) Implement evaluation utilities
8) Finetune model
9) Evaluate finetuned model
10) Use model on new data
In this last section of this chapter, we try the model on new text messages

Finally, let's use the finetuned GPT-based spam classification model. The following `classify_review` function follows data preprocessing steps similar to those we used in the `SpamDataset` implemented earlier in this chapter. And then, after processing text into token IDs, the function uses the model to predict an integer class label, similar to what we have implemented in section 6.6, and then returns the corresponding class name:

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) #A
    supported_context_length = model.pos_emb.weight.shape[1]

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

    input_ids += [pad_token_id] * (max_length - len(input_ids)) #C
    input_tensor = torch.tensor(input_ids, device=device).unsqueeze(0) #D


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

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

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"` . Next, 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
))
```

Also, here, the model 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 using the `torch.save` method we introduced in the previous chapter.

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

Once saved, the model can be loaded as follows:

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

6.9 Summary

- There are different strategies for finetuning LLMs, including classification-finetuning (this chapter) and instruction-finetuning (next chapter)
- Classification-finetuning 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 2 output nodes; in previous chapters, the number of output nodes was equal to the number of unique tokens in the vocabulary, namely, 50,256
- Instead of predicting the next token in the text as in pretraining, classification-finetuning trains the model to output a correct class label, for example, "spam" or "not spam."
- The model input for finetuning is text converted into token IDs, similar to pretraining.
- Before finetuning 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).
- Finetuning a classification model uses the same cross entropy loss function that is used for pretraining the LLM.