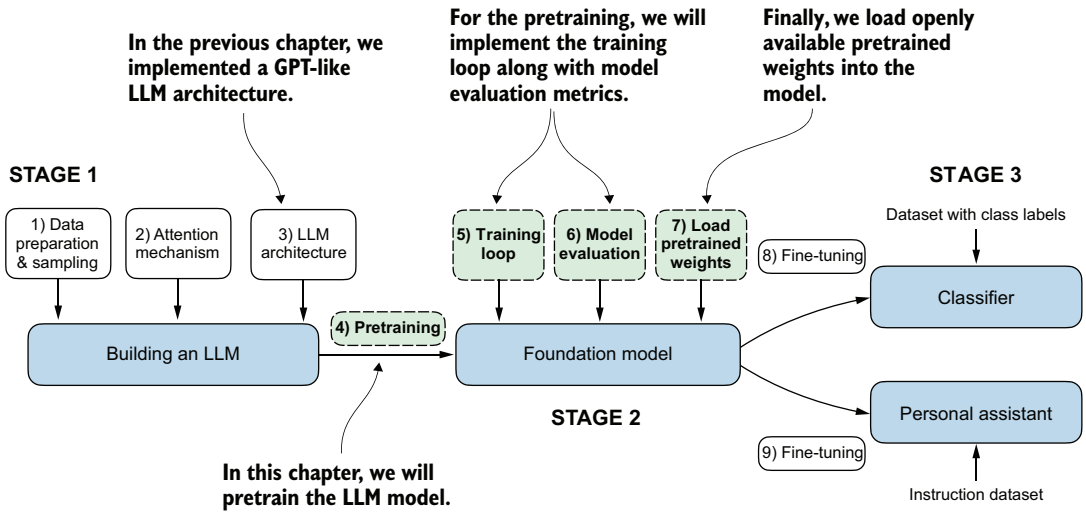

Figure 5.1 The three main stages of coding an LLM. This chapter focuses on stage 2: pretraining the LLM (step 4), which includes implementing the training code (step 5), evaluating the performance (step 6), and saving and loading model weights (step 7).

Weight parameters

In the context of LLMs and other deep learning models, *weights* refer to the trainable parameters that the learning process adjusts. These weights are also known as *weight parameters* or simply *parameters*. In frameworks like PyTorch, these weights are stored in linear layers; we used these to implement the multi-head attention module in chapter 3 and the `GPTModel` in chapter 4. After initializing a layer (`new_layer = torch.nn.Linear(...)`), we can access its weights through the `.weight` attribute, `new_layer.weight`. Additionally, for convenience, PyTorch allows direct access to all a model's trainable parameters, including weights and biases, through the method `model.parameters()`, which we will use later when implementing the model training.

5.1 Evaluating generative text models

After briefly recapping the text generation from chapter 4, we will set up our LLM for text generation and then discuss basic ways to evaluate the quality of the generated text. We will then calculate the training and validation losses. Figure 5.2 shows the topics covered in this chapter, with these first three steps highlighted.

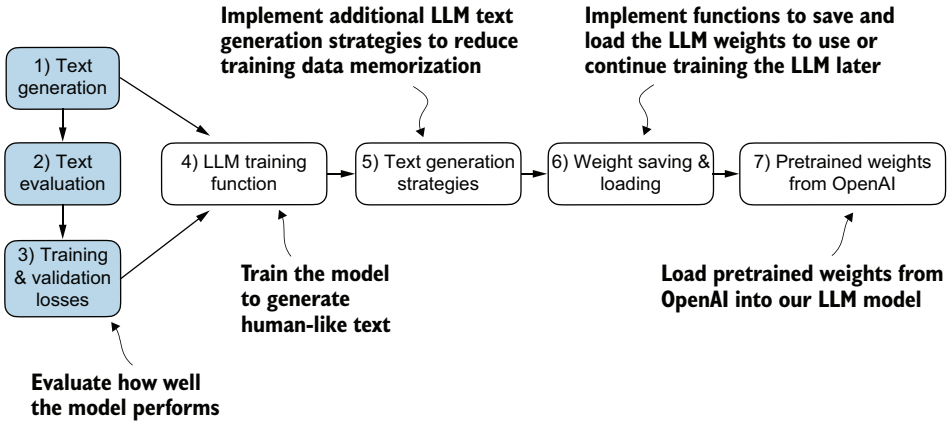

Figure 5.2 An overview of the topics covered in this chapter. We begin by recapping text generation (step 1) before moving on to discuss basic model evaluation techniques (step 2) and training and validation losses (step 3).

5.1.1 Using GPT to generate text

Let's set up the LLM and briefly recap the text generation process we implemented in chapter 4. We begin by initializing the GPT model that we will later evaluate and train using the `GPTModel` class and `GPT_CONFIG_124M` dictionary (see chapter 4):

```
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
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

We shorten the context length from 1,024 to 256 tokens.

It's possible and common to set dropout to 0.

Considering the `GPT_CONFIG_124M` dictionary, the only adjustment we have made compared to the previous chapter is that we have reduced the context length (`context_length`) to 256 tokens. This modification reduces the computational demands of training the model, making it possible to carry out the training on a standard laptop computer.

Originally, the GPT-2 model with 124 million parameters was configured to handle up to 1,024 tokens. After the training process, we will update the context size setting

and load pretrained weights to work with a model configured for a 1,024-token context length.

Using the `GPTModel` instance, we adopt the `generate_text_simple` function from chapter 4 and introduce two handy functions: `text_to_token_ ids` and `token_ids_ to_text`. These functions facilitate the conversion between text and token representations, a technique we will utilize throughout this chapter.

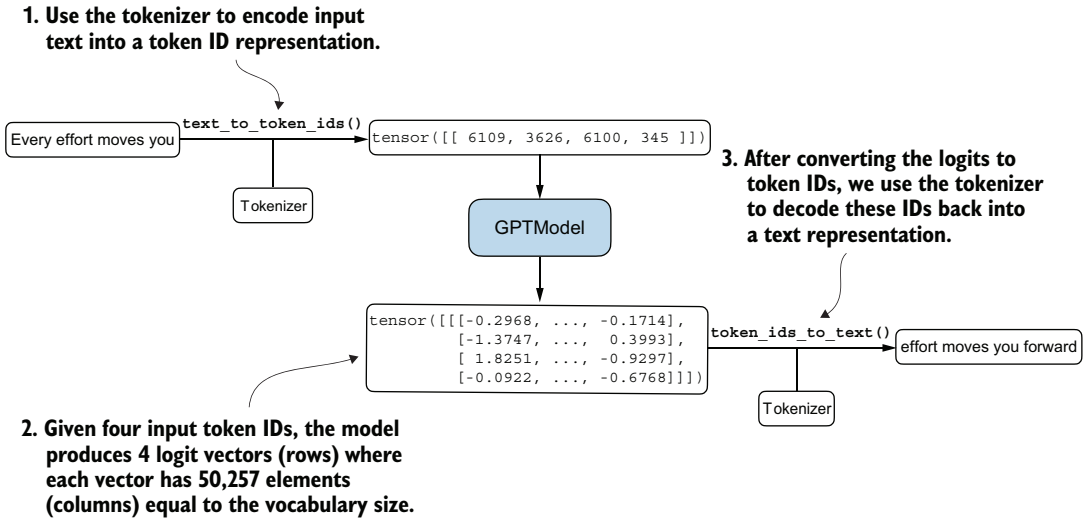

Figure 5.3 Generating text involves encoding text into token IDs that the LLM processes into logit vectors. The logit vectors are then converted back into token IDs, detokenized into a text representation.

Figure 5.3 illustrates a three-step text generation process using a GPT model. First, the tokenizer converts input text into a series of token IDs (see chapter 2). Second, the model receives these token IDs and generates corresponding logits, which are vectors representing the probability distribution for each token in the vocabulary (see chapter 4). Third, these logits are converted back into token IDs, which the tokenizer decodes into human-readable text, completing the cycle from textual input to textual output.

We can implement the text generation process, as shown in the following listing.

Listing 5.1 Utility functions for text to token ID conversion

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
```