Exercise 5.3

What are the different combinations of settings for the `generate` function to force deterministic behavior, that is, disabling the random sampling such that it always produces the same outputs similar to the `generate_simple` function?

5.4 Loading and saving model weights in PyTorch

Thus far, we have discussed how to numerically evaluate the training progress and pretrain an LLM from scratch. Even though both the LLM and dataset were relatively small, this exercise showed that pretraining LLMs is computationally expensive. Thus, it is important to be able to save the LLM so that we don't have to rerun the training every time we want to use it in a new session.

So, let's discuss how to save and load a pretrained model, as highlighted in figure 5.16. Later, we will load a more capable pretrained GPT model from OpenAI into our `GPTModel` instance.

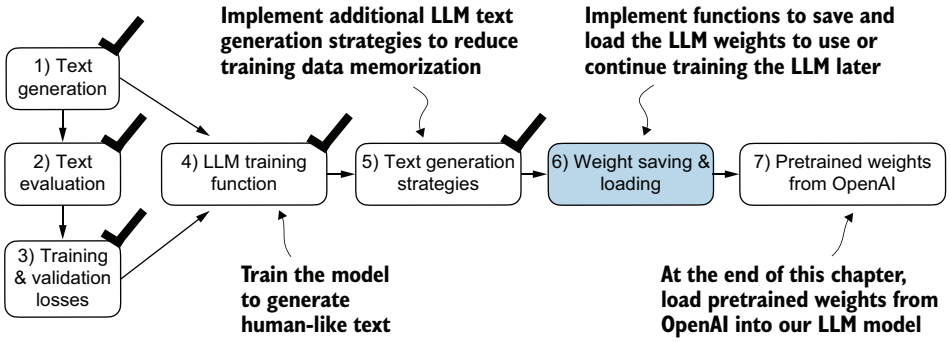

Figure 5.16 After training and inspecting the model, it is often helpful to save the model so that we can use or continue training it later (step 6).

Fortunately, saving a PyTorch model is relatively straightforward. The recommended way is to save a model's `state_dict`, a dictionary mapping each layer to its parameters, using the `torch.save` function:

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

`"model.pth"` is the filename where the `state_dict` is saved. The `.pth` extension is a convention for PyTorch files, though we could technically use any file extension.

Then, after saving the model weights via the `state_dict`, we can load the model weights into a new `GPTModel` model instance:

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

As discussed in chapter 4, dropout helps prevent the model from overfitting to the training data by randomly "dropping out" of a layer's neurons during training. However, during inference, we don't want to randomly drop out any of the information the network has learned. Using `model.eval()` switches the model to evaluation mode for inference, disabling the dropout layers of the `model`. If we plan to continue pretraining a model later—for example, using the `train_model_simple` function we defined earlier in this chapter—saving the optimizer state is also recommended.

Adaptive optimizers such as AdamW store additional parameters for each model weight. AdamW uses historical data to adjust learning rates for each model parameter dynamically. Without it, the optimizer resets, and the model may learn suboptimally or even fail to converge properly, which means it will lose the ability to generate coherent text. Using `torch.save`, we can save both the model and optimizer `state_dict` contents:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

Then we can restore the model and optimizer states by first loading the saved data via `torch.load` and then using the `load_state_dict` method:

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
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"])
model.train();
```

Exercise 5.4

After saving the weights, load the model and optimizer in a new Python session or Jupyter notebook file and continue pretraining it for one more epoch using the `train_model_simple` function.

5.5 Loading pretrained weights from OpenAI

Previously, we trained a small GPT-2 model using a limited dataset comprising a short-story book. This approach allowed us to focus on the fundamentals without the need for extensive time and computational resources.

Fortunately, OpenAI openly shared the weights of their GPT-2 models, thus eliminating the need to invest tens to hundreds of thousands of dollars in retraining the model on a large corpus ourselves. So, let's load these weights into our `GPTModel` class and use the model for text generation. Here, *weights* refer to the weight parameters stored in the `.weight` attributes of PyTorch's `Linear` and `Embedding` layers, for example. We accessed them earlier via `model.parameters()` when training the model. In chapter 6, will reuse these pretrained weights to fine-tune the model for a text classification task and follow instructions similar to ChatGPT.

Note that OpenAI originally saved the GPT-2 weights via TensorFlow, which we have to install to load the weights in Python. The following code will use a progress bar tool called `tqdm` to track the download process, which we also have to install.

You can install these libraries by executing the following command in your terminal:

```
pip install tensorflow>=2.15.0  tqdm>=4.66
```

The download code is relatively long, mostly boilerplate, and not very interesting. Hence, instead of devoting precious space to discussing Python code for fetching files from the internet, we download the `gpt_download.py` Python module directly from this chapter's online repository:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Next, after downloading this file to the local directory of your Python session, you should briefly inspect the contents of this file to ensure that it was saved correctly and contains valid Python code.

We can now import the `download_and_load_gpt2` function from the `gpt_download.py` file as follows, which will load the GPT-2 architecture settings (`settings`) and weight parameters (`params`) into our Python session:

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

Executing this code downloads the following seven files associated with the `124M` parameter GPT-2 model:

```
checkpoint: 100%|███████████████████████████| 77.0/77.0 [00:00<00:00,
                                                          63.9kiB/s]
encoder.json: 100%|████████████████████████| 1.04M/1.04M [00:00<00:00,
                                                           2.20MiB/s]
```