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()
```