```
        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).