To get the model ready for classification fine-tuning, we first *freeze* the model, meaning that we make all layers nontrainable:

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

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

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
)
```

To keep the code more general, we use `BASE_CONFIG["emb_dim"]`, which is equal to 768 in the `"gpt2-small (124M)"` model. Thus, 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, fine-tuning additional layers can noticeably improve the predictive performance of the model. (For more details, refer to appendix B.) We also 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.

To make the final `LayerNorm` and last transformer block trainable, 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 Fine-tuning the whole model

Instead of fine-tuning just the final transformer block, fine-tune the entire model and assess the effect on predictive performance.

Even though we added a new output layer and marked certain layers as trainable or nontrainable, we can still use this model similarly to how we have previously. For

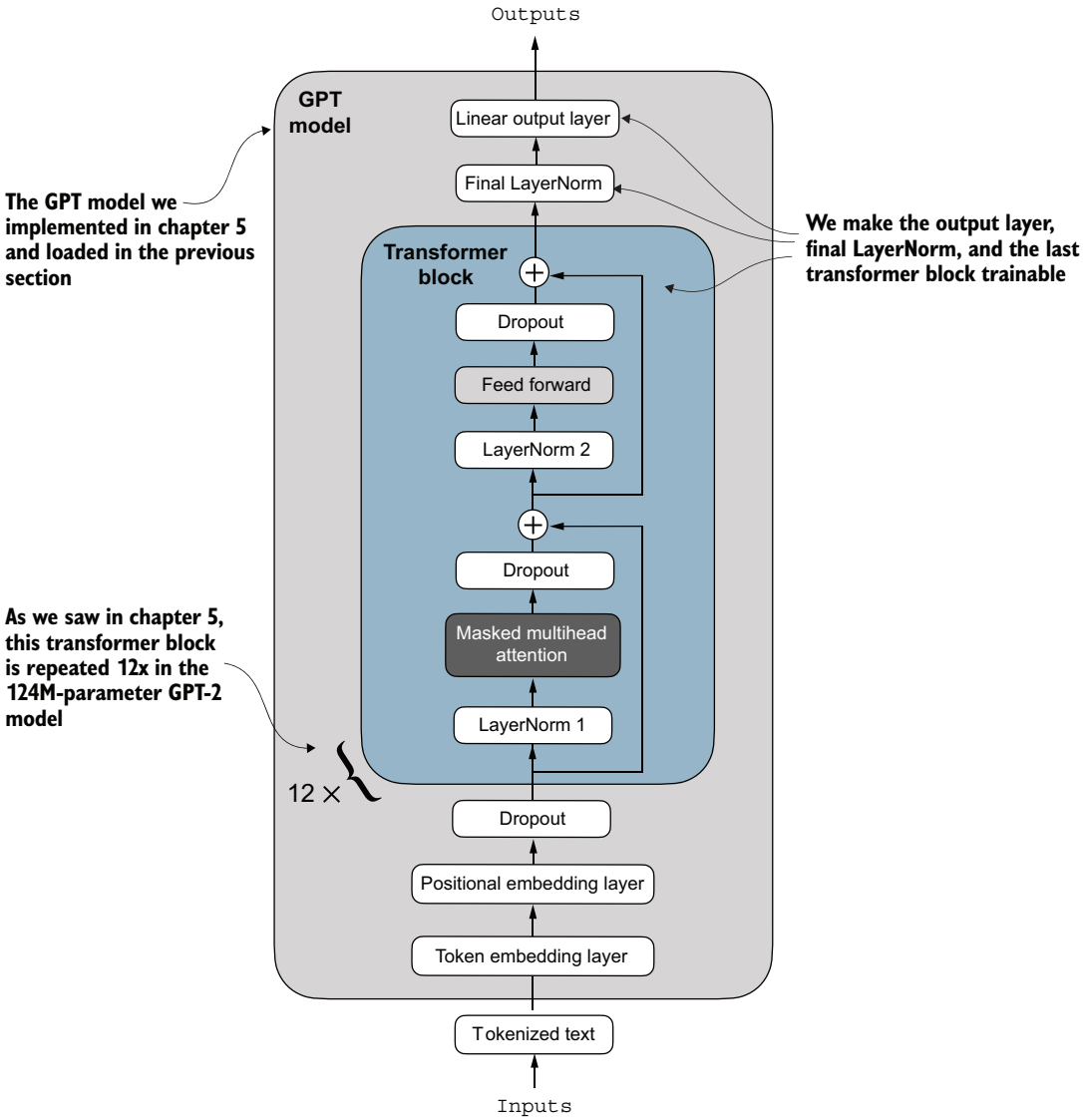

Figure 6.10 The GPT model includes 12 repeated transformer blocks. Alongside the output layer, we set the final `LayerNorm` and the last transformer block as trainable. The remaining 11 transformer blocks and the embedding layers are kept nontrainable.

instance, we can feed it an example text identical to our previously used 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)

The print output shows that the preceding code encodes the inputs into a tensor consisting of four 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)
```

The output tensor looks like the following:

```
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])
```

A similar input would have previously produced an output tensor of `[1, 4, 50257]`, where `50257` represents the vocabulary size. The number of output rows corresponds to the number of input tokens (in this case, four). However, each output's embedding dimension (the number of columns) is now 2 instead of 50,257 since we replaced the output layer of the model.

Remember that we are interested in fine-tuning this model to return a class label indicating whether a model input is "spam" or "not spam." We don't need to fine-tune all four output rows; instead, we can focus on a single output token. In particular, we will focus on the last row corresponding to the last output token, as shown in figure 6.11.

To extract the last output token from the output tensor, we use the following code:

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

This prints

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

We still need to convert the values into a class-label prediction. But first, let's understand why we are particularly interested in the last output token only.

We have already explored the attention mechanism, which establishes a relationship between each input token and every other input token, and the concept of a *causal attention mask*, commonly used in GPT-like models (see chapter 3). This mask restricts a