As we can see based on the sharp downward slope in figure 6.16, the model is learning well from the training data, and there is little to no indication of overfitting; that is, there is no noticeable gap between the training and validation set losses.

Choosing the number of epochs

Earlier, when we initiated the training, we set the number of epochs to five. The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation, although an epoch number of five is usually a good starting point. If the model overfits after the first few epochs as a loss plot (see figure 6.16), you may need to reduce the number of epochs. Conversely, if the trendline suggests that the validation loss could improve with further training, you should increase the number of epochs. In this concrete case, five epochs is a reasonable number as there are no signs of early overfitting, and the validation loss is close to 0.

Using the same `plot_values` function, let's now plot the classification accuracies:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="accuracy"
)
```

Figure 6.17 graphs the resulting accuracy. The model achieves a relatively high training and validation accuracy after epochs 4 and 5. Importantly, we previously set `eval_iter=5`

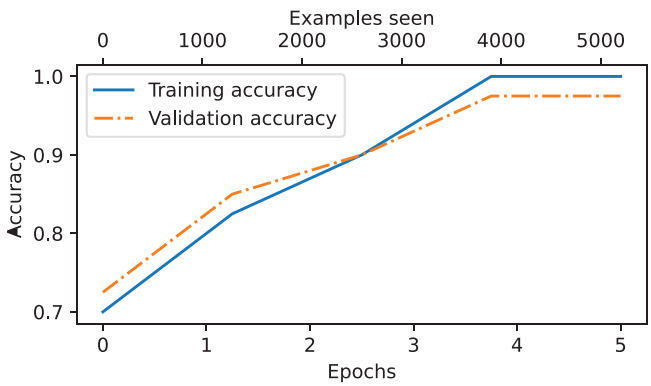

Figure 6.17 Both the training accuracy (solid line) and the validation accuracy (dashed line) increase substantially in the early epochs and then plateau, achieving almost perfect accuracy scores of 1.0. The close proximity of the two lines throughout the epochs suggests that the model does not overfit the training data very much.

when using the `train_classifier_simple` function, which means our estimations of training and validation performance are based on only five batches for efficiency during training.

Now we must calculate the performance metrics for the training, validation, and test sets across the entire dataset by running the following code, this time without defining the `eval_iter` value:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

The training and test set performances are almost identical. The slight discrepancy between the training and test set accuracies suggests minimal overfitting of the training data. Typically, the validation set accuracy is somewhat higher than the test set accuracy because the model development often involves tuning hyperparameters to perform well on the validation set, which might not generalize as effectively to the test set. This situation is common, but the gap could potentially be minimized by adjusting the model's settings, such as increasing the dropout rate (`drop_rate`) or the `weight_decay` parameter in the optimizer configuration.

6.8 Using the LLM as a spam classifier

Having fine-tuned and evaluated the model, we are now ready to classify spam messages (see figure 6.18). Let's use our fine-tuned GPT-based spam classification model. The following `classify_review` function follows data preprocessing steps similar to those we used in the `SpamDataset` implemented earlier. Then, after processing text into token IDs, the function uses the model to predict an integer class label, similar to what we implemented in section 6.6, and then returns the corresponding class name.

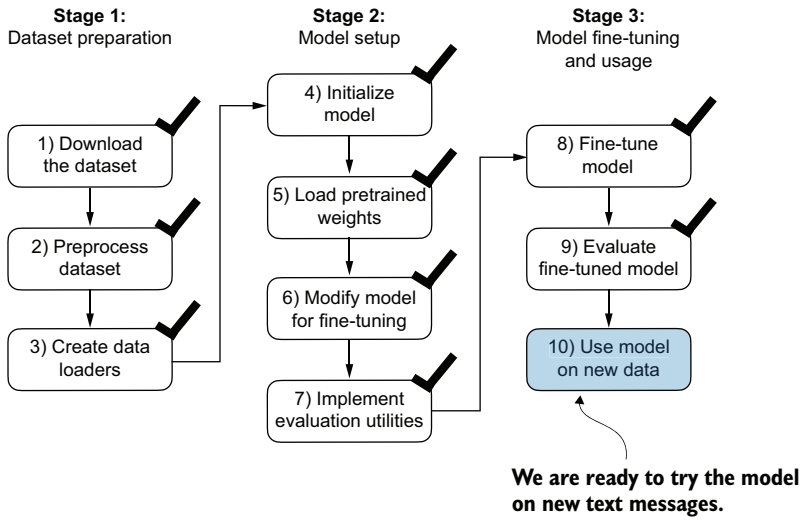

Figure 6.18 The three-stage process for classification fine-tuning our LLM. Step 10 is the final step of stage 3—using the fine-tuned model to classify new spam messages.

Listing 6.12 Using the model to classify new texts

```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[0]

    input_ids = input_ids[:min(
        max_length, supported_context_length
    )]

    input_ids += [pad_token_id] * (max_length - len(input_ids))

    input_tensor = torch.tensor(
        input_ids, device=device
    ).unsqueeze(0)

    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam"
```

Prepares inputs to the model

Truncates sequences if they are too long

Pads sequences to the longest sequence

Adds batch dimension

Models inference without gradient tracking

Logits of the last output token

Returns the classified result