Let's consider the last token output using a concrete example:

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

The values of the tensor corresponding to the last token are

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

We can obtain the class label:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

In this case, the code returns `1`, meaning the model predicts that the input text is "spam." Using the `softmax` function here is optional because the largest outputs directly correspond to the highest probability scores. Hence, we can simplify the code without using softmax:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

This concept can be used to compute the classification accuracy, which measures the percentage of correct predictions across a dataset.

To determine the classification accuracy, we apply the `argmax`-based prediction code to all examples in the dataset and calculate the proportion of correct predictions by defining a `calc_accuracy_loader` function.

Listing 6.8 Calculating the classification accuracy

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)


            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]
            predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (
```

Logits of last output token

```
                (predicted_labels == target_batch).sum().item()
            )

        else:
            break
    return correct_predictions / num_examples
```

Let's use the function to determine the classification accuracies across various datasets estimated from 10 batches for efficiency:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

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

Via the `device` setting, the model automatically runs on a GPU if a GPU with Nvidia CUDA support is available and otherwise runs on a CPU. The output is

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

As we can see, the prediction accuracies are near a random prediction, which would be 50% in this case. To improve the prediction accuracies, we need to fine-tune the model.

However, before we begin fine-tuning the model, we must define the loss function we will optimize during training. Our objective is to maximize the spam classification accuracy of the model, which means that the preceding code should output the correct class labels: `0` for non-spam and `1` for spam.

Because classification accuracy is not a differentiable function, we use cross-entropy loss as a proxy to maximize accuracy. Accordingly, the `calc_loss_batch` function remains the same, with one adjustment: we focus on optimizing only the last token, `model(input_batch)[:, -1, :]`, rather than all tokens, `model(input_batch)`:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)[:, -1, :]
```

Logits of last output token

```
    loss = torch.nn.functional.cross_entropy(logits, target_batch)
    return loss
```

We use the `calc_loss_batch` function to compute the loss for a single batch obtained from the previously defined data loaders. To calculate the loss for all batches in a data loader, we define the `calc_loss_loader` function as before.

Listing 6.9 Calculating the classification loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Ensures number of batches doesn't exceed batches in data loader

Similar to calculating the training accuracy, we now compute the initial loss for each data set:

```
with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

Disables gradient tracking for efficiency because we are not training yet

The initial loss values are

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Next, we will implement a training function to fine-tune the model, which means adjusting the model to minimize the training set loss. Minimizing the training set loss will help increase the classification accuracy, which is our overall goal.