Working with logarithms of probability scores is more manageable in mathematical optimization than handling the scores directly. This topic is outside the scope of this book, but I've detailed it further in a lecture, which can be found in appendix B.

Next, we combine these log probabilities into a single score by computing the average (step 5 in figure 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

The resulting average log probability score is

```
tensor(-10.7940)
```

The goal is to get the average log probability as close to 0 as possible by updating the model's weights as part of the training process. However, in deep learning, the common practice isn't to push the average log probability up to 0 but rather to bring the negative average log probability down to 0. The negative average log probability is simply the average log probability multiplied by –1, which corresponds to step 6 in figure 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

This prints `tensor(10.7940)`. In deep learning, the term for turning this negative value, –10.7940, into 10.7940, is known as the *cross entropy* loss. PyTorch comes in handy here, as it already has a built-in `cross_entropy` function that takes care of all these six steps in figure 5.7 for us.

Cross entropy loss

At its core, the cross entropy loss is a popular measure in machine learning and deep learning that measures the difference between two probability distributions—typically, the true distribution of labels (here, tokens in a dataset) and the predicted distribution from a model (for instance, the token probabilities generated by an LLM).

In the context of machine learning and specifically in frameworks like PyTorch, the `cross_entropy` function computes this measure for discrete outcomes, which is similar to the negative average log probability of the target tokens given the model's generated token probabilities, making the terms "cross entropy" and "negative average log probability" related and often used interchangeably in practice.

Before we apply the `cross_entropy` function, let's briefly recall the shape of the logits and target tensors:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

The resulting shapes are

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

As we can see, the `logits` tensor has three dimensions: batch size, number of tokens, and vocabulary size. The `targets` tensor has two dimensions: batch size and number of tokens.

For the `cross_entropy` loss function in PyTorch, we want to flatten these tensors by combining them over the batch dimension:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

The resulting tensor dimensions are

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Remember that the `targets` are the token IDs we want the LLM to generate, and the `logits` contain the unscaled model outputs before they enter the `softmax` function to obtain the probability scores.

Previously, we applied the `softmax` function, selected the probability scores corresponding to the target IDs, and computed the negative average log probabilities. PyTorch's `cross_entropy` function will take care of all these steps for us:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

The resulting loss is the same that we obtained previously when applying the individual steps in figure 5.7 manually:

```
tensor(10.7940)
```

Perplexity

Perplexity is a measure often used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling. It can provide a more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.

Perplexity measures how well the probability distribution predicted by the model matches the actual distribution of the words in the dataset. Similar to the loss, a lower perplexity indicates that the model predictions are closer to the actual distribution.

(continued)

Perplexity can be calculated as `perplexity = torch.exp(loss)`, which returns `tensor(48725.8203)` when applied to the previously calculated loss.

Perplexity is often considered more interpretable than the raw loss value because it signifies the effective vocabulary size about which the model is uncertain at each step. In the given example, this would translate to the model being unsure about which among 48,725 tokens in the vocabulary to generate as the next token.

We have now calculated the loss for two small text inputs for illustration purposes. Next, we will apply the loss computation to the entire training and validation sets.

5.1.3 Calculating the training and validation set losses

We must first prepare the training and validation datasets that we will use to train the LLM. Then, as highlighted in figure 5.8, we will calculate the cross entropy for the training and validation sets, which is an important component of the model training process.

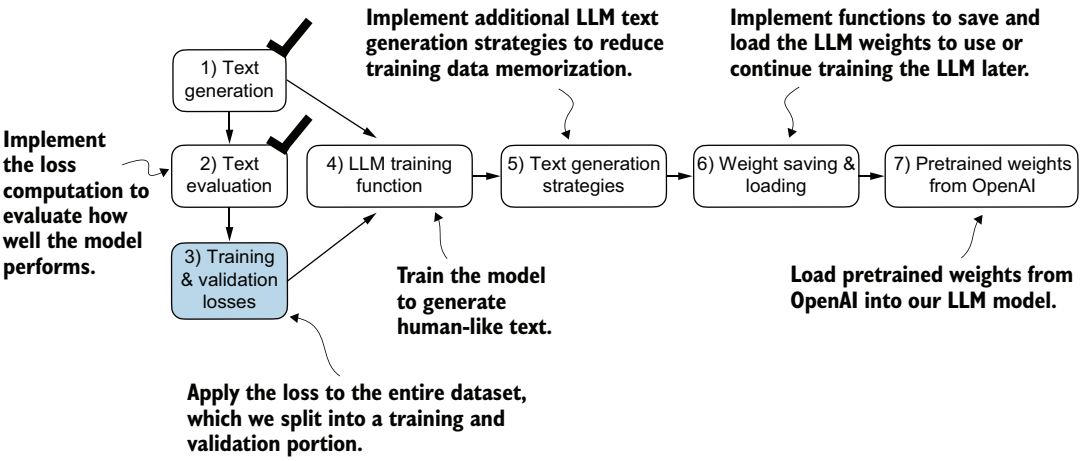

Figure 5.8 Having completed steps 1 and 2, including computing the cross entropy loss, we can now apply this loss computation to the entire text dataset that we will use for model training.

To compute the loss on the training and validation datasets, we use a very small text dataset, the "The Verdict" short story by Edith Wharton, which we have already worked with in chapter 2. By selecting a text from the public domain, we circumvent any concerns related to usage rights. Additionally, using such a small dataset allows for the execution of code examples on a standard laptop computer in a matter of