The model produces random text that is different from the target text because it has not been trained yet. We now want to evaluate the performance of the model's generated text numerically via a loss (figure 5.5). Not only is this useful for measuring the quality of the generated text, but it's also a building block for implementing the training function, which we will use to update the model's weight to improve the generated text.

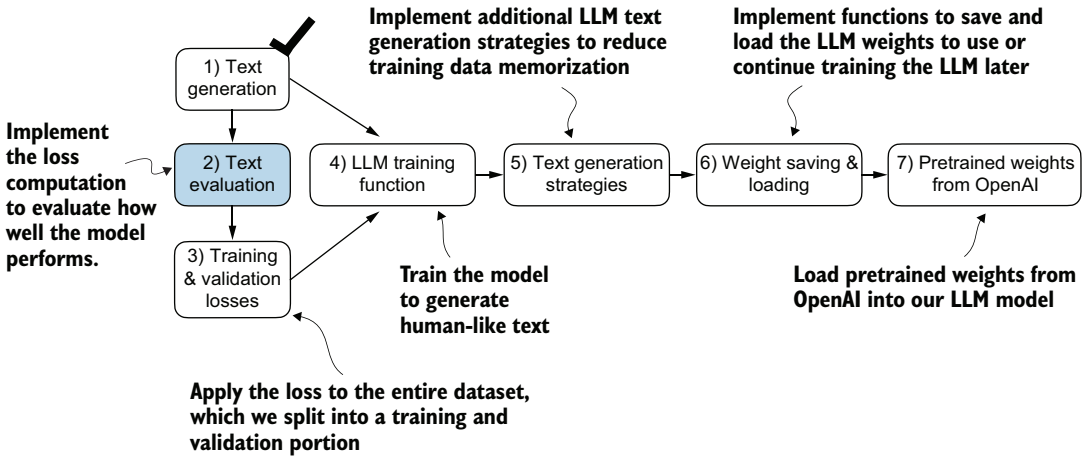

Figure 5.5 An overview of the topics covered in this chapter. We have completed step 1. We are now ready to implement the text evaluation function (step 2).

Part of the text evaluation process that we implement, as shown in figure 5.5, is to measure "how far" the generated tokens are from the correct predictions (targets). The training function we implement later will use this information to adjust the model weights to generate text that is more similar to (or, ideally, matches) the target text.

The model training aims to increase the softmax probability in the index positions corresponding to the correct target token IDs, as illustrated in figure 5.6. This softmax probability is also used in the evaluation metric we will implement next to numerically assess the model's generated outputs: the higher the probability in the correct positions, the better.

Remember that figure 5.6 displays the softmax probabilities for a compact seven-token vocabulary to fit everything into a single figure. This implies that the starting random values will hover around 1/7, which equals approximately 0.14. However, the vocabulary we are using for our GPT-2 model has 50,257 tokens, so most of the initial probabilities will hover around 0.00002 (1/50,257).

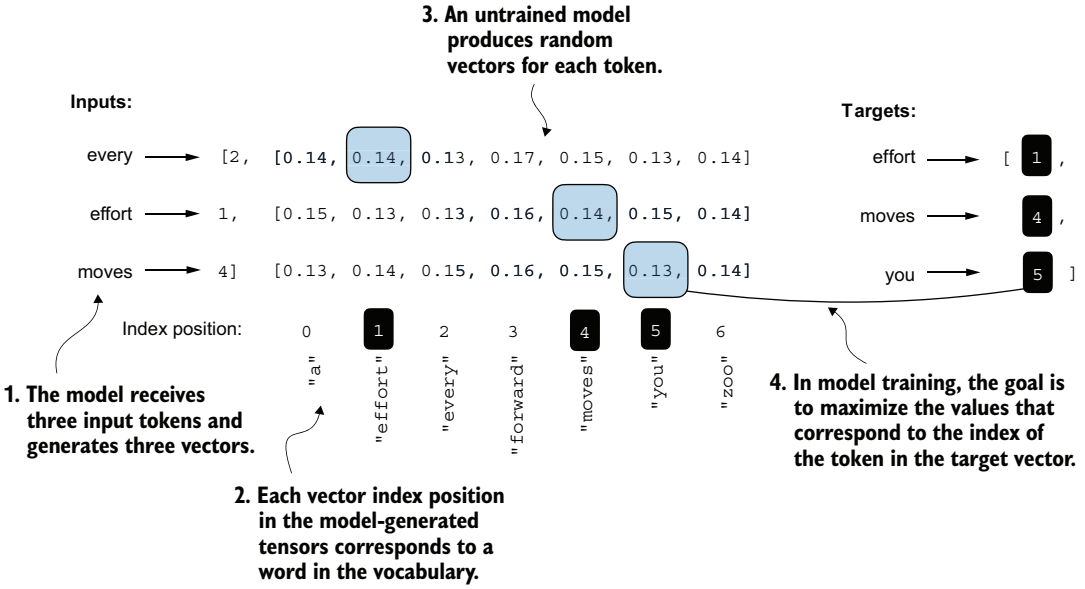

Figure 5.6 Before training, the model produces random next-token probability vectors. The goal of model training is to ensure that the probability values corresponding to the highlighted target token IDs are maximized.

For each of the two input texts, we can print the initial softmax probability scores corresponding to the target tokens using the following code:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
```

The three target token ID probabilities for each batch are

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

The goal of training an LLM is to maximize the likelihood of the correct token, which involves increasing its probability relative to other tokens. This way, we ensure the LLM consistently picks the target token—essentially the next word in the sentence—as the next token it generates.

Backpropagation

How do we maximize the softmax probability values corresponding to the target tokens? The big picture is that we update the model weights so that the model outputs higher values for the respective token IDs we want to generate. The weight update is done via a process called *backpropagation*, a standard technique for training deep neural networks (see sections A.3 to A.7 in appendix A for more details about backpropagation and model training).

Backpropagation requires a loss function, which calculates the difference between the model's predicted output (here, the probabilities corresponding to the target token IDs) and the actual desired output. This loss function measures how far off the model's predictions are from the target values.

Next, we will calculate the loss for the probability scores of the two example batches, `target_probas_1` and `target_probas_2`. The main steps are illustrated in figure 5.7. Since we already applied steps 1 to 3 to obtain `target_probas_1` and `target_probas_2`, we proceed with step 4, applying the *logarithm* to the probability scores:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
```

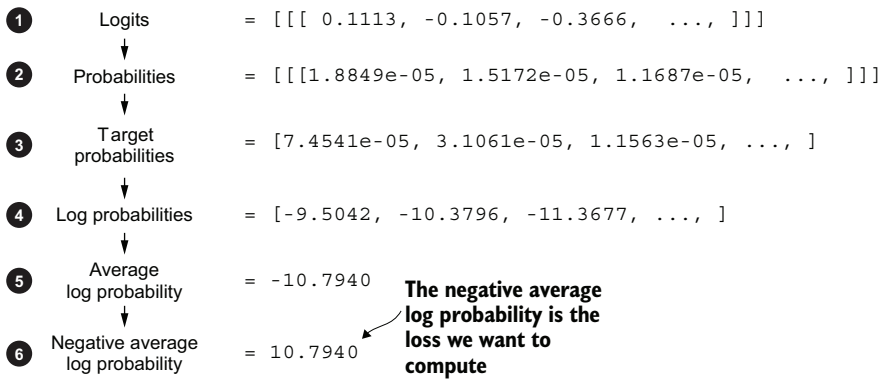

Figure 5.7 Calculating the loss involves several steps. Steps 1 to 3, which we have already completed, calculate the token probabilities corresponding to the target tensors. These probabilities are then transformed via a logarithm and averaged in steps 4 to 6.

This results in the following values:

```
tensor([ -9.5042, -10.3796, -11.3677, -11.4798,  -9.7764, -12.2561])
```