```
Validation loader:
torch.Size([2, 256]) torch.Size([2, 256])
```

Based on the preceding code output, we have nine training set batches with two samples and 256 tokens each. Since we allocated only 10% of the data for validation, there is only one validation batch consisting of two input examples. As expected, the input data (`x`) and target data (`y`) have the same shape (the batch size times the number of tokens in each batch) since the targets are the inputs shifted by one position, as discussed in chapter 2.

Next, we implement a utility function to calculate the cross entropy loss of a given batch returned via the training and validation loader:

```
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)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

The transfer to a given device allows us to transfer the data to a GPU.

We can now use this `calc_loss_batch` utility function, which computes the loss for a single batch, to implement the following `calc_loss_loader` function that computes the loss over all the batches sampled by a given data loader.

Listing 5.2 Function to compute the training and validation 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
```

Iteratives over all batches if no fixed num_batches is specified

Reduces the number of batches to match the total number of batches in the data loader if num_batches exceeds the number of batches in the data loader

Sums loss for each batch

Averages the loss over all batches

By default, the `calc_loss_loader` function iterates over all batches in a given data loader, accumulates the loss in the `total_loss` variable, and then computes and

averages the loss over the total number of batches. Alternatively, we can specify a smaller number of batches via `num_batches` to speed up the evaluation during model training.

Let's now see this `calc_loss_loader` function in action, applying it to the training and validation set loaders:

If you have a machine with a CUDA-supported GPU, the LLM will train on the GPU without making any changes to the code.

Disables gradient tracking for efficiency because we are not training yet

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

Via the "device" setting, we ensure the data is loaded onto the same device as the LLM model.

The resulting loss values are

```
Training loss: 10.98758347829183
Validation loss: 10.98110580444336
```

The loss values are relatively high because the model has not yet been trained. For comparison, the loss approaches 0 if the model learns to generate the next tokens as they appear in the training and validation sets.

Now that we have a way to measure the quality of the generated text, we will train the LLM to reduce this loss so that it becomes better at generating text, as illustrated in figure 5.10.

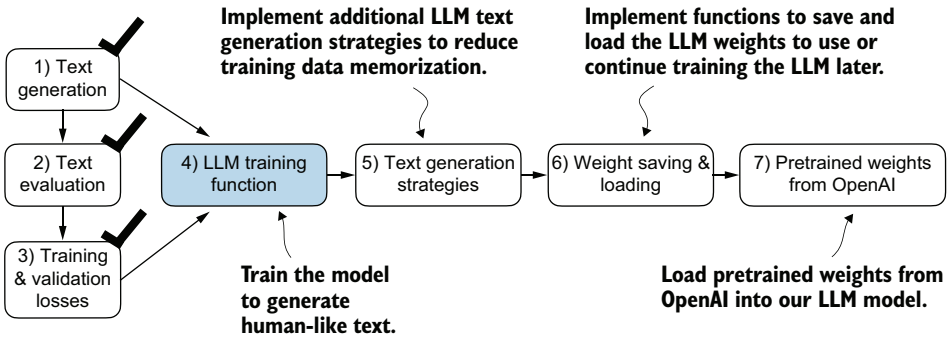

Figure 5.10 We have recapped the text generation process (step 1) and implemented basic model evaluation techniques (step 2) to compute the training and validation set losses (step 3). Next, we will go to the training functions and pretrain the LLM (step 4).

Next, we will focus on pretraining the LLM. After model training, we will implement alternative text generation strategies and save and load pretrained model weights.

5.2 Training an LLM

It is finally time to implement the code for pretraining the LLM, our `GPTModel`. For this, we focus on a straightforward training loop to keep the code concise and readable.

NOTE Interested readers can learn about more advanced techniques, including *learning rate warmup*, *cosine annealing*, and *gradient clipping*, in appendix D.

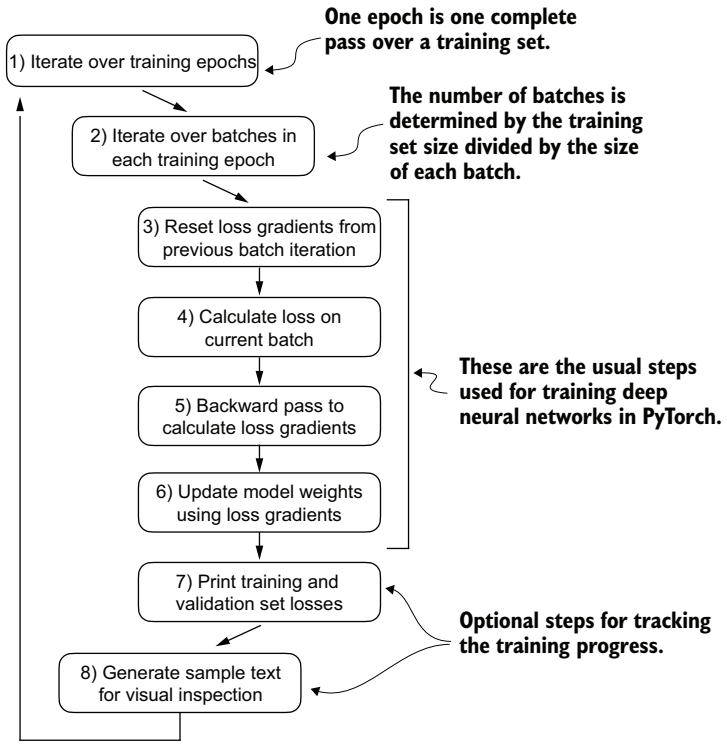

Figure 5.11 A typical training loop for training deep neural networks in PyTorch consists of numerous steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights so that the training set loss is minimized.

The flowchart in figure 5.11 depicts a typical PyTorch neural network training workflow, which we use for training an LLM. It outlines eight steps, starting with iterating over each epoch, processing batches, resetting gradients, calculating the loss and new