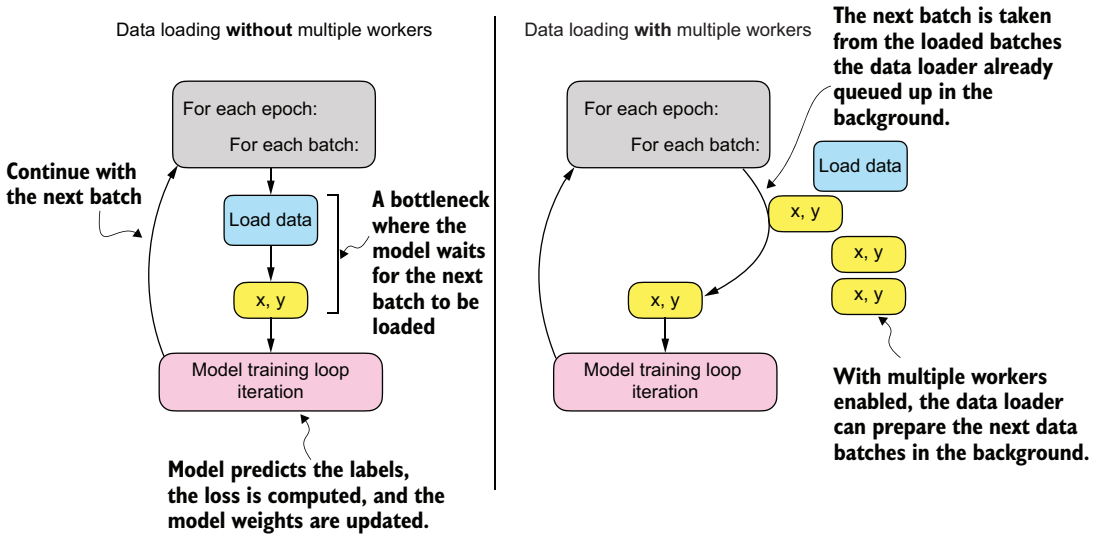

Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded (left). If multiple workers are enabled, the data loader can queue up the next batch in the background (right).

the tradeoff and make a calculated decision on setting the `num_workers` parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

In my experience, setting `num_workers=4` usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the `Dataset` class.

A.7 A typical training loop

Let's now train a neural network on the toy dataset. The following listing shows the training code.

Listing A.9 Neural network training in PyTorch

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(
    model.parameters(), lr=0.5
)

num_epochs = 3
for epoch in range(num_epochs):

    model.train()
```

The dataset has two features and two classes.

The optimizer needs to know which parameters to optimize.

```
for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
          f" | Train Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code
```

Sets the gradients from the previous round to 0 to prevent unintended gradient accumulation

Computes the gradients of the loss given the model parameters

The optimizer uses the gradients to update the model parameters.

Running this code yields the following outputs:

```
Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Trainl Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00
```

As we can see, the loss reaches 0 after three epochs, a sign that the model converged on the training set. Here, we initialize a model with two inputs and two outputs because our toy dataset has two input features and two class labels to predict. We used a stochastic gradient descent (`SGD`) optimizer with a learning rate (`lr`) of 0.5. The learning rate is a hyperparameter, meaning it's a tunable setting that we must experiment with based on observing the loss. Ideally, we want to choose a learning rate such that the loss converges after a certain number of epochs—the number of epochs is another hyperparameter to choose.

Exercise A.3

How many parameters does the neural network introduced in listing A.9 have?

In practice, we often use a third dataset, a so-called validation dataset, to find the optimal hyperparameter settings. A validation dataset is similar to a test set. However, while we only want to use a test set precisely once to avoid biasing the evaluation, we usually use the validation set multiple times to tweak the model settings.

We also introduced new settings called `model.train()` and `model.eval()`. As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training and inference, such as *dropout* or *batch normalization* layers. Since we don't have dropout

or other components in our `NeuralNetwork` class that are affected by these settings, using `model.train()` and `model.eval()` is redundant in our preceding code. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

As discussed earlier, we pass the logits directly into the `cross_entropy` loss function, which will apply the `softmax` function internally for efficiency and numerical stability reasons. Then, calling `loss.backward()` will calculate the gradients in the computation graph that PyTorch constructed in the background. The `optimizer.step()` method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

NOTE To prevent undesired gradient accumulation, it is important to include an `optimizer.zero_grad()` call in each update round to reset the gradients to 0. Otherwise, the gradients will accumulate, which may be undesired.

After we have trained the model, we can use it to make predictions:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

The results are

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

To obtain the class membership probabilities, we can then use PyTorch's `softmax` function:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

This outputs

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
```

Let's consider the first row in the preceding code output. Here, the first value (column) means that the training example has a 99.91% probability of belonging to class