NOTE PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at zero). So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of five nodes.

Next, we create a custom dataset class, `ToyDataset`, by subclassing from PyTorch's `Dataset` parent class, as shown in the following listing.

Listing A.6 Defining a custom `Dataset` class

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

Instructions for retrieving exactly one data record and the corresponding label

Instructions for returning the total length of the dataset

The purpose of this custom `ToyDataset` class is to instantiate a PyTorch `DataLoader`. But before we get to this step, let's briefly go over the general structure of the `ToyDataset` code.

In PyTorch, the three main components of a custom `Dataset` class are the `__init__` constructor, the `__getitem__` method, and the `__len__` method (see listing A.6). In the `__init__` method, we set up attributes that we can access later in the `__getitem__` and `__len__` methods. These could be file paths, file objects, database connectors, and so on. Since we created a tensor dataset that sits in memory, we simply assign `X` and `y` to these attributes, which are placeholders for our tensor objects.

In the `__getitem__` method, we define instructions for returning exactly one item from the dataset via an `index`. This refers to the features and the class label corresponding to a single training example or test instance. (The data loader will provide this `index`, which we will cover shortly.)

Finally, the `__len__` method contains instructions for retrieving the length of the dataset. Here, we use the `.shape` attribute of a tensor to return the number of rows in the feature array. In the case of the training dataset, we have five rows, which we can double-check:

```
print(len(train_ds))
```

The result is

```
5
```

Now that we've defined a PyTorch `Dataset` class we can use for our toy dataset, we can use PyTorch's `DataLoader` class to sample from it, as shown in the following listing.

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader


torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

The ToyDataset instance created earlier serves as input to the data loader.

Whether or not to shuffle the data

The number of background processes

It is not necessary to shuffle a test dataset.

After instantiating the training data loader, we can iterate over it. The iteration over the `test_loader` works similarly but is omitted for brevity:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is

```
Batch 1: tensor([[-1.2000,  3.1000],
                 [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                 [-0.9000,  2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

As we can see based on the preceding output, the `train_loader` iterates over the training dataset, visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using `torch.manual_seed(123)` here, you should get the exact same shuffling order of training examples. However, if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks from getting caught in repetitive update cycles during training.

We specified a batch size of 2 here, but the third batch only contains a single example. That's because we have five training examples, and 5 is not evenly divisible by 2.

In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, set `drop_last=True`, which will drop the last batch in each epoch, as shown in the following listing.

Listing A.8 A training loader that drops the last batch

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

Now, iterating over the training loader, we can see that the last batch is omitted:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is

```
Batch 1: tensor([[-0.9000,  2.9000],
        [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
        [-0.5000,  2.6000]]) tensor([1, 0])
```

Lastly, let's discuss the setting `num_workers=0` in the `DataLoader`. This parameter in PyTorch's `DataLoader` function is crucial for parallelizing data loading and preprocessing. When `num_workers` is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. Instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when `num_workers` is set to a number greater than 0, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources (figure A.11).

However, if we are working with very small datasets, setting `num_workers` to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. So, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing `num_workers` may not provide any noticeable speedup. It may, in fact, lead to some problems. One potential problem is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

Furthermore, for Jupyter notebooks, setting `num_workers` to greater than 0 can sometimes lead to problems related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand