Chapter 7. PyTorch

In Chapters 6 and 5, you learned how convolutional and recurrent neural networks worked by implementing them from scratch. Nevertheless, while understanding how they work is necessary, that knowledge alone won't get them to work on a real-world problem; for that, you need to be able to implement them in a high-performance library. We could devote an entire book to building a high-performance neural network library, but that would be a much different (or simply much longer) book, for a much different audience. Instead, we'll devote this last chapter to introducing PyTorch, an increasingly popular neural network framework based on automatic differentiation, which we introduced at the beginning of Chapter 6.

As in the rest of the book, we'll write our code in a way that maps to the mental models of how neural networks work, writing classes for `Layer`s, `Trainer`s, and so on. In doing so, we won't be writing our code in line with common PyTorch practices, but we'll include links on the book's GitHub repo for you to learn more about expressing neural networks the way PyTorch was designed to express them. Before we get there, let's start by learning the data type at the core of PyTorch that enables its automatic differentiation and thus its ability to express neural network training cleanly: the `Tensor`.

PyTorch Tensors

In the last chapter, we showed a simple `NumberWithGrad` accumulate gradients by keeping track of the operations performed on it. This meant that if we wrote:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e.backward()
```

then `a.grad` would equal 35, which is actually the partial derivative of `e` with respect to `a`.

ints. Let's rewrite the preceding example using a PyTorch Tensor. First we'll initialize a Tensor manually:

```
a = torch.Tensor([[3., 3.,],
                  [3., 3.]], requires_grad=True)
```

Note a couple of things here:

1. We can initialize a Tensor by simply wrapping the data contained in it in a torch.Tensor, just as we did with ndarrays.

2. When initializing a Tensor this way, we have to pass in the argument requires_grad=True to tell the Tensor to accumulate gradients.

Once we've done this, we can perform computations as before:

```
b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e_sum = e.sum()
e_sum.backward()
```

You can see that there's an extra step here compared to the NumberWithGrad example: we have to *sum* e before calling backward on its sum. This is because, as we argued in the first chapter, it doesn't make sense to think of "the derivative of a number with respect to an array": we can, however, reason about what the partial derivative of e_sum with respect to each element of a would be—and indeed, we see that the answer is consistent with what we found in the prior chapters:

```
print(a.grad)
```

```
tensor([[35., 35.],
        [35., 35.]], dtype=torch.float64)
```

This feature of PyTorch enables us to define models simply by defining the forward pass, computing a loss, and calling .backward on the loss to automatically compute the derivative of each of the parameters with respect to that loss. In particular, we don't have to worry about reusing the same quantity multiple times in the forward pass (which was the limitation of the Operation framework we used in the first few chapters); as this simple example shows, gradients will automatically be computed correctly once we call backward on the output of our computations.

In the next several sections, we'll show how the training framework we laid out earlier in the book can be implemented with PyTorch's data types.

Deep Learning with PyTorch

As we've seen, deep learning models have several elements that work together to produce a trained model:

- A Model, which contains Layers
- An Optimizer
- A Loss
- A Trainer

It turns out that with PyTorch, the Optimizer and the Loss are one-liners, and

PyTorch Elements: Model, Layer, Optimizer, and Loss

A key feature of PyTorch is the ability to define models and layers as easy-to-use objects that handle sending gradients backward and storing parameters automatically, simply by having them inherit from the `torch.nn.Module` class. You'll see how these pieces come together later in this chapter; for now, just know that `PyTorchLayer` can be written as:

```
from torch import nn, Tensor

class PyTorchLayer(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

and `PyTorchModel` can also be written this way:

```
class PyTorchModel(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

In other words, each subclass of a `PyTorchLayer` or a `PyTorchModel` will just need to implement `__init__` and `forward` methods, which will allow us to use them in intuitive ways.[1]

THE INFERENCE FLAG

As we saw in Chapter 4, because of dropout, we need the ability to change our model's behavior depending on whether we are running it in training mode or in inference mode. In PyTorch, we can switch a model or layer from training mode (its default behavior) to inference mode by running `m.eval` on the model or layer (any object that inherits from `nn.Module`). Furthermore, PyTorch has an elegant way to quickly change the behavior of all subclasses of a layer using the `apply` function. If we define:

```
def inference_mode(m: nn.Module):
    m.eval()
```

then we can include:

```
if inference:
    self.apply(inference_mode)
```

in the `forward` method of each subclass of `PyTorchModel` or `PyTorchLayer` we define, thus getting the flag we desire.

Let's see how this comes together.

Implementing Neural Network Building Blocks Using PyTorch: DenseLayer

We now have all the prerequisites to start implementing the `Layer`s we've seen previously, but with PyTorch operations. A `DenseLayer` layer would be written as follows:

```
                 neurons: int,
                 dropout: float = 1.0,
                 activation: nn.Module = None) -> None:
        super().__init__()
        self.linear = nn.Linear(input_size, neurons)
        self.activation = activation
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 - dropout)


    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        if inference:
            self.apply(inference_mode)

        x = self.linear(x) # does weight multiplication + bias
        if self.activation:
            x = self.activation(x)
        if hasattr(self, "dropout"):
            x = self.dropout(x)

        return x
```

Here, with `nn.Linear`, we see our first example of a PyTorch operation that automatically handles backpropagation for us. This object not only handles the weight multiplication and the addition of a bias term on the forward pass but also causes `x`'s gradients to accumulate so that the correct derivatives of the loss with respect to the parameters can be computed on the backward pass. Note also that since all PyTorch operations inherit from `nn.Module`, we can call them like mathematical functions: in the preceding case, for example, we write `self.linear(x)` rather than `self.linear.forward(x)`. This also holds true for the `DenseLayer` itself, as we'll see when we use it in the upcoming model.

Example: Boston Housing Prices Model in PyTorch

Using this `Layer` as a building block, we can implement the now-familiar housing prices model from Chapters 2 and 3. Recall that this model simply had one hidden layer with a `sigmoid` activation; in Chapter 3, we implemented this within our object-oriented framework that had a class for the `Layer`s and a model that had a list of length 2 as its `layers` attribute. Similarly, we can define a `HousePricesModel` class that inherits from `PyTorchModel` as follows:

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13,
                 hidden_dropout: float = 1.0):
        super().__init__()
        self.dense1 = DenseLayer(13, hidden_size,
                                 activation=nn.Sigmoid(),
                                 dropout = hidden_dropout)
        self.dense2 = DenseLayer(hidden_size, 1)

    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)

        assert x.shape[1] == 13

        x = self.dense1(x)
        return self.dense2(x)
```

We can then instantiate this via:

```
pytorch_boston_model = HousePricesModel(hidden_size=13)
```

Note that it is not conventional to write a separate `Layer` class for PyTorch

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13):
        super().__init__()
        self.fc1 = nn.Linear(13, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)

        assert x.shape[1] == 13

        x = self.fc1(x)
        x = torch.sigmoid(x)
        return self.fc2(x)
```

When building PyTorch models on your own in the future, you may want to write your code in this way rather than creating a separate `Layer` class—and when *reading* others' code, you'll almost always see something similar to the preceding code.

`Layer`s and `Model`s are more involved than `Optimizer`s and `Loss`es, which we'll cover next.

PyTorch Elements: Optimizer and Loss

`Optimizer`s and `Loss`es are implemented in PyTorch as one-liners. For example, the `SGDMomentum` loss we covered in Chapter 4 can be written as:

```
import torch.optim as optim

optimizer = optim.SGD(pytorch_boston_model.parameters(), lr=0.001)
```

> **NOTE**
>
> In PyTorch, models are passed into the `Optimizer` as an argument; this ensures that the optimizer is "pointed at" the correct model's parameters so it knows what to update on each iteration (we did this using the `Trainer` class earlier).

Furthermore, the mean squared error loss we saw in Chapter 2 and the `SoftmaxCrossEntropyLoss` we discussed in Chapter 4 can simply be written as:

```
mean_squared_error_loss = nn.MSELoss()
softmax_cross_entropy_loss = nn.CrossEntropyLoss()
```

Like the preceding `Layer`s, these inherit from `nn.Module`, so they can be called in the same way as `Layer`s.

NOTE

Note that even though the word *softmax* is not in the name of the `nn.CrossEntropyLoss` class, the softmax operation is indeed performed on the inputs, so that we can pass in "raw outputs" from the neural network rather than outputs that have already passed through the softmax function, just as we did before.

These `Loss`es inherit from `nn.Module`, just like the `Layer`s from earlier, so they can be called the same way, using `loss(x)` instead of `loss.forward(x)`, for example.

PyTorch Elements: Trainer

The `Trainer` pulls all of these elements together. Let's consider the requirements for the `Trainer`. We know that it has to implement the general pattern for training neural networks that we've seen many times throughout this book:

1. Feed a batch of inputs through the model.
2. Feed the outputs and targets into a loss function to compute a loss value.
3. Compute the gradient of the loss with respect to all of the parameters.
4. Use the `Optimizer` to update the parameters according to some rule.

With PyTorch, this all works the same way, except there are two small implementation caveats:

- By default, `Optimizer`s will retain the gradients of the parameters (what we referred to as `param_grads` earlier in the book) after each iteration of a parameter update. To clear these gradients before the next parameter update, we'll call `self.optim.zero_grad`.
- As illustrated previously in the simple automatic differentiation example, to kick off the backpropagation, we'll have to call `loss.backward` after computing the loss value.

This leads to the following sequence of code that is seen throughout PyTorch training loops, and will in fact be used in the `PyTorchTrainer` class. As the `Trainer` class from prior chapters did, `PyTorchTrainer` will take in an `Optimizer`, a `PyTorchModel`, and a `Loss` (either `nn.MSELoss` or `nn.CrossEntropyLoss`) for a batch of data `(X_batch, y_batch)`; with these objects in place as `self.optim`, `self.model`, and `self.loss`, respectively, the following five lines of code train the model:

```
# First, zero the gradients
self.optim.zero_grad()

# feed X_batch through the model
output = self.model(X_batch)

# Compute the loss
loss = self.loss(output, y_batch)

# Call backward on the loss to kick off backpropagation
loss.backward()

# Call self.optim.step() (as before) to update the parameters
```

Those are the most important lines; still, here's the rest of the code for the `PyTorchTrainer`, much of which is similar to the code for the `Trainer` that we saw in prior chapters:

```
class PyTorchTrainer(object):
    def __init__(self,
                 model: PyTorchModel,
                 optim: Optimizer,
                 criterion: _Loss):
        self.model = model
        self.optim = optim
        self.loss = criterion
        self._check_optim_net_aligned()

    def _check_optim_net_aligned(self):
        assert self.optim.param_groups[0]['params']\
        == list(self.model.parameters())

    def _generate_batches(self,
                          X: Tensor,
                          y: Tensor,
                          size: int = 32) -> Tuple[Tensor]:

        N = X.shape[0]

        for ii in range(0, N, size):
            X_batch, y_batch = X[ii:ii+size], y[ii:ii+size]

            yield X_batch, y_batch


    def fit(self, X_train: Tensor, y_train: Tensor,
            X_test: Tensor, y_test: Tensor,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32):

        for e in range(epochs):
            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self._generate_batches(X_train, y_train,
                                                     batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):

                self.optim.zero_grad()
                output = self.model(X_batch)
                loss = self.loss(output, y_batch)
                loss.backward()
                self.optim.step()

            output = self.model(X_test)
            loss = self.loss(output, y_test)
            print(e, loss)
```

NOTE

Since we're passing a `Model`, an `Optimizer`, and a `Loss` into the `Trainer`, we need to check that the parameters that the `Optimizer` refers to are in fact the same as the model's parameters; `_check_optim_net_aligned` does this.

Now training the model is as simple as:

```
net = HousePricesModel()
optimizer = optim.SGD(net.parameters(), lr=0.001)
```

```
trainer.fit(X_train, y_train, X_test, y_test,
            epochs=10,
            eval_every=1)
```

This code is nearly identical to the code we used to train models using the framework we built in the first three chapters. Whether you're using PyTorch, TensorFlow, or Theano under the hood, the elements of training a deep learning model remain the same!

Next, we'll explore more features of PyTorch by showing how to implement the tricks to improve training that we saw in Chapter 4.

Tricks to Optimize Learning in PyTorch

We learned four tricks to accelerate learning in Chapter 4:

- Momentum
- Dropout
- Weight initialization
- Learning rate decay

These are all easy to implement in PyTorch. For example, to include momentum in our optimizer, we can simply include a `momentum` keyword in `SGD`, so that the optimizer becomes:

```
optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Dropout is similarly easy. Just as PyTorch has a built-in `Module` `nn.Linear(n_in, n_out)` that computes the operations of a `Dense` layer from before, the `Module` `nn.Dropout(dropout_prob)` implements the `Dropout` operation, with the caveat that the probability passed in is by default the probability of *dropping* a given neuron, rather than keeping it as it was in our implementation from before.

We don't need to worry about weight initialization at all: the weights in most PyTorch operations involving parameters, including `nn.Linear`, are automatically scaled based on the size of the layer.

Finally, PyTorch has an `lr_scheduler` class that can be used to decay the learning rate over the epochs. The key import you need to get started is from `torch.optim import lr_scheduler`.[2] Now you can easily use these techniques we covered from first principles in any future deep learning project you work on!

Convolutional Neural Networks in PyTorch

In Chapter 5, we systematically covered how convolutional neural networks work, focusing in particular on the multichannel convolution operation. We saw that the operation transforms the pixels of input images into layers of neurons organized into feature maps, where each neuron represents whether a given visual feature (defined by a convolutional filter) is present at that location in the image. The multichannel convolution operation had the following shapes for its two inputs and its output:

- The data input shape `[batch_size, in_channels, image_height, image_width]`
- The parameters input shape `[in_channels, out_channels, filter_size, filter_size]`

In terms of this notation, the multichannel convolution operation in PyTorch is:

```
nn.Conv2d(in_channels, out_channels, filter_size)
```

With this defined, wrapping a `ConvLayer` around this operation is straightforward:

```
class ConvLayer(PyTorchLayer):
    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 filter_size: int,
                 activation: nn.Module = None,
                 flatten: bool = False,
                 dropout: float = 1.0) -> None:
        super().__init__()

        # the main operation of the layer
        self.conv = nn.Conv2d(in_channels, out_channels, filter_size,
                              padding=filter_size // 2)

        # the same "activation" and "flatten" operations from before
        self.activation = activation
        self.flatten = flatten
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 - dropout)

    def forward(self, x: Tensor) -> Tensor:

        # always apply the convolution operation
        x = self.conv(x)

        # optionally apply the convolution operation
        if self.activation:
            x = self.activation(x)
        if self.flatten:
            x = x.view(x.shape[0], x.shape[1] * x.shape[2] * x.shape
        if hasattr(self, "dropout"):
            x = self.dropout(x)

        return x
```

NOTE

In Chapter 5, we automatically padded the output based on the filter size to keep the output image the same size as the input image. PyTorch does not do that; to achieve the same behavior we had before, we add an argument to the `nn.Conv2d` operation setting `padding = filter_size // 2`.

From there, all we have to do is define a `PyTorchModel` with its operations in the `__init__` function and the sequence of operations defined in the `forward` function to begin to train. Next is a simple architecture we can use on the MNIST dataset we saw in Chapters 4 and 5, with:

- A convolutional layer that transforms the input from 1 "channel" to 16 channels
- Another layer that transforms these 16 channels into 8 (with each channel still containing 28 × 28 neurons)

The pattern of several convolutional layers followed by a smaller number of fully connected layers is common for convolutional architectures; here, we just use two of each:

```
class MNIST_ConvNet(PyTorchModel):
    def __init__(self):
        super().__init__()
        self.conv1 = ConvLayer(1, 16, 5, activation=nn.Tanh(),
                               dropout=0.8)
        self.conv2 = ConvLayer(16, 8, 5, activation=nn.Tanh(), flatte
                               dropout=0.8)
        self.dense1 = DenseLayer(28 * 28 * 8, 32, activation=nn.Tanh(
                                 dropout=0.8)
        self.dense2 = DenseLayer(32, 10)

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)

        x = self.dense1(x)
        x = self.dense2(x)
        return x
```

Then we can train this model the same way we trained the `HousePricesModel`:

```
model = MNIST_ConvNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

trainer = PyTorchTrainer(model, optimizer, criterion)

trainer.fit(X_train, y_train,
            X_test, y_test,
            epochs=5,
            eval_every=1)
```

There is an important caveat related to the `nn.CrossEntropyLoss` class. Recall that in the custom framework from previous chapters, our `Loss` class expected an input of the same shape as the target. To get this, we one-hot encoded the 10 distinct values of the target in the MNIST data so that, for each batch of data, the target had shape `[batch_size, 10]`.

With PyTorch's `nn.CrossEntropyLoss` class—which works exactly the same as our `SoftmaxCrossEntropyLoss` from before—we don't have to do that. This loss function expects two `Tensor`s:

- A prediction `Tensor` of size `[batch_size, num_classes]`, just as our `SoftmaxCrossEntropyLoss` class did before
- A target `Tensor` of size `[batch_size]` with `num_classes` different values

So in the preceding example, `y_train` is simply an array of size `[60000]` (the number of observations in the training set of MNIST), and `y_test` simply has size `[10000]` (the number of observations in the test set).

Now that we're dealing with larger datasets, we should cover another best practice. It is clearly very memory inefficient to load the entire training and testing sets into memory to train the model, as we're doing with `X_train`, `y_train`, `X_test`, and `y_test`. PyTorch has a way around this: the `DataLoader` class.

DataLoader and Transforms

dividing by the global standard deviation to roughly "normalize" the data:

```
X_train, X_test = X_train - X_train.mean(), X_test - X_train.mean()
X_train, X_test = X_train / X_train.std(), X_test / X_train.std()
```

Still, this required us to first fully read these two arrays into memory; it would be much more efficient to perform this preprocessing on the fly, as batches are fed into the neural network. PyTorch has built-in functions that do this, and they are especially commonly used with image data—transformations via the `transforms` module, and a `DataLoader` via `torch.utils.data`:

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

Previously, we read in the entire training set into `X_train` via:

```
mnist_trainset = MNIST(root="../data/", train=True)
X_train = mnist_trainset.train_data
```

We then performed transformations on `X_train` to get it to a form where it was ready for modeling.

PyTorch has some convenience functions that allow us to compose many transformations to each batch of data as it is read in; this allows us both to avoid reading the entire dataset into memory and to use PyTorch's transformations.

We first define a list of transformations to perform on each batch of data read in. For example, the following transformations convert each MNIST image to a `Tensor` (most PyTorch datasets are "PIL images" by default, so `transforms.ToTensor()` is often the first transformation in the list), and then "normalize" the dataset—subtracting off the mean and then dividing by the standard deviation—using the overall MNIST mean and standard deviation of `0.1305` and `0.3081`, respectively:

```
img_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1305,), (0.3081,))
])
```

NOTE

`Normalize` actually subtracts the mean and standard deviation *from each channel* of the input image. Thus, it is common when dealing with color images with three input channels to have a `Normalize` transformation that has two tuples of three numbers each—for example, `transforms.Normalize((0.1, 0.3, 0.6), (0.4, 0.2, 0.5))`, which would tell the `DataLoader` to:

- Normalize the first channel using a mean of 0.1 and a standard deviation of 0.4
- Normalize the second channel using a mean of 0.3 and a standard deviation of 0.2
- Normalize the third channel using a mean of 0.6 and a standard deviation of 0.5

Second, once these transformations have been applied, we apply these to the `dataset` as we read in batches:

```
dataset = MNIST("../mnist_data/", transform=img_transforms)
```

Finally, we can define a `DataLoader` that takes in this dataset and defines rules for successively generating batches of data:

```
dataloader = DataLoader(dataset, batch_size=60, shuffle=True)
```

We can then modify the `Trainer` to use the `dataloader` to generate the batches used to train the network instead of loading the entire dataset into memory and then manually generating them using the `batch_generator` function, as we did before. On the book's website,[3] I show an example of training a convolutional neural network using these `DataLoader`s. The main change in the `Trainer` is simply changing the line:

```
for X_batch, y_batch in enumerate(batch_generator):
```

to:

```
for X_batch, y_batch in enumerate(train_dataloader):
```

In addition, instead of feeding in the entire training set into the `fit` function, we now feed in `DataLoader`s:

```
trainer.fit(train_dataloader = train_loader,
            test_dataloader = test_loader,
            epochs=1,
            eval_every=1)
```

Using this architecture and calling the `fit` method, as we just did, gets us to about 97% accuracy on MNIST after one epoch. More important than the accuracy, however, is that you've seen how to implement the concepts we reasoned through from first principles into a high-performance framework. Now that you understand both the underlying concepts and the framework, I encourage you to modify the code in the book's GitHub repo and try out other convolutional architectures, other datasets, and so on.

CNNs were one of two advanced architectures we covered earlier in the book; let's now turn to the other one and show how to implement the most advanced RNN variant we've covered, LSTMs, in PyTorch.

LSTMs in PyTorch

We saw in the last chapter how to code LSTMs from scratch. We coded an `LSTMLayer` to take in an input `ndarray` of size [`batch_size`, `sequence_length`, `feature_size`], and output an `ndarray` of size [`batch_size`, `sequence_length`, `feature_size`]. In addition, each layer took in a hidden state and a cell state, each initialized with shape `[1, hidden_size]`, expanded to shape `[batch_size, hidden_size]` when a batch is passed in, and then collapsed back down to `[1, hidden_size]` after the iteration is complete.

Based on this, we define the `__init__` method for our `LSTMLayer` as:

```
class LSTMLayer(PyTorchLayer):
    def __init__(self,
                 sequence_length: int,
                 input_size: int,
```

```
        self.h_init = torch.zeros((1, hidden_size))
        self.c_init = torch.zeros((1, hidden_size))
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = DenseLayer(hidden_size, output_size)
```

As with convolutional layers, PyTorch has an `nn.lstm` operation for implementing LSTMs. Note that in our custom `LSTMLayer` we store a `DenseLayer` in the `self.fc` attribute. You may recall from the last chapter that the last step of an LSTM cell is putting the final hidden state through the operations of a `Dense` layer (a weight multiplication and addition of a bias) to transform the hidden state into dimension `output_size` for each operation. PyTorch does things a bit differently: the `nn.lstm` operation simply outputs the hidden states for each time step. Thus, to enable our `LSTMLayer` to output a different dimension than its input—as we would want all of our neural network layers to be able to do—we add a `DenseLayer` at the end to transform the hidden state into dimension `output_size`.

With this modification, the `forward` function is now straightforward, looking similar to the `forward` function of the `LSTMLayer` from Chapter 6:

```
def forward(self, x: Tensor) -> Tensor:

    batch_size = x.shape[0]

    h_layer = self._transform_hidden_batch(self.h_init,
                                           batch_size,
                                           before_layer=True)
    c_layer = self._transform_hidden_batch(self.c_init,
                                           batch_size,
                                           before_layer=True)


    x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))

    self.h_init, self.c_init = (
        self._transform_hidden_batch(h_out,
                                     batch_size,
                                     before_layer=False).detach(),
        self._transform_hidden_batch(c_out,
                                     batch_size,
                                     before_layer=False).detach()
                                    )

    x = self.fc(x)

    return x
```

The key line here, which should look familiar given our implementation of LSTMs in Chapter 6, is:

```
x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))
```

Aside from that, there's some reshaping of the hidden and cell states before and after the `self.lstm` function via a helper function `self._transform_hidden_batch`. You can see the full function in the book's GitHub repo.

Finally, wrapping a model around this is easy:

```
class NextCharacterModel(PyTorchModel):
    def __init__(self,
                 vocab_size: int,
                 hidden_size: int = 256,
                 sequence_length: int = 25):
        super().__init__()
        self.vocab_size = vocab_size
```

```
        self.lstm = LSTMLayer(self.sequence_length,
                              self.vocab_size,
                              hidden_size,
                              self.vocab_size)

    def forward(self,
                inputs: Tensor):
        assert_dim(inputs, 3) # batch_size, sequence_length, vocab_siz

        out = self.lstm(inputs)

        return out.permute(0, 2, 1)
```

NOTE

The `nn.CrossEntropyLoss` function expects the first two dimensions to be the `batch_size` and the distribution over the classes; the way we've been implementing our LSTMs, however, we have the distribution over the classes as the last dimension (`vocab_size`) coming out of the `LSTMLayer`. To prepare the final model output to be fed into the loss, therefore, we move the dimension containing the distribution over letters to the second dimension using `out.permute(0, 2, 1)`.

Finally, in the book's GitHub repo, I show how to write a class `LSTMTrainer` to inherit from `PyTorchTrainer` and use it to train a `NextCharacterModel` to generate text. We use the same text preprocessing that we did in Chapter 6: selecting sequences of text, one-hot encoding the letters, and grouping the sequences of one-hot encoded letters into batches.

That wraps up how to translate the three neural network architectures for supervised learning we saw in this book—fully connected neural networks, convolutional neural networks, and recurrent neural networks—into PyTorch. To conclude, we'll briefly cover how neural networks can be used for the other half of machine learning: *un*-supervised learning.

Postscript: Unsupervised Learning via Autoencoders

Throughout this book we've focused on how deep learning models can be used to solve *supervised* learning problems. There is, of course, a whole other side to machine learning: unsupervised learning; which involves what is often described as "finding structure in data without labels"; I like to think of it, though, as finding relationships between characteristics in your data that have not yet been measured, whereas supervised learning involves finding relationships between characteristics in your data that have already been measured.

Suppose you had a dataset of images with no labels. You don't know much about these images—for example, you're not sure whether there are 10 distinct digits represented, or 5, or 20 (these images could be from a strange alphabet)—and you want to know the answers to questions like:

- How many distinct digits are there?
- Which digits are visually similar to one another?
- Are there "outlier" images that are distinctly *dis*similar to other images?

To understand how deep learning can help with this, we'll have to take a quick

Representation Learning

We've seen that deep learning models can learn to make accurate predictions. They do this by transforming the input they receive into representations that are progressively both more abstract and more tuned to directly making predictions for whatever the relevant problem is. In particular, the final layer of the network, directly before the layer with the predictions themselves (which would have just one neuron for a regression problem and `num_classes` neurons for a classification problem), is the network's attempt at creating a representation of the input data that is as useful as possible for the task of making predictions. This is shown in Figure 7-1.

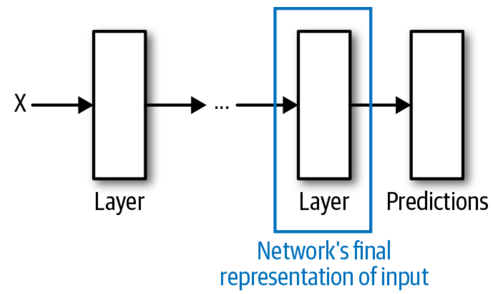

Figure 7-1. The final layer of a neural network, immediately before the predictions, represents the network's representation of the input that it has found most useful to the task of predicting

Once trained, then, a model can not only make predictions for new data points, *but also generate representations of these data points*. These could then be used for clustering, similarity analysis, or outlier detection—in addition to prediction.

An Approach for Situations with No Labels Whatsoever

A limitation with this whole approach is that it *requires labels to train the model to generate the representations in the first place*. The question is: how can we train a model to generate "useful" representations without any labels? If we don't have labels, we need to generate representations of our data using the only thing we do have: the training data itself. This is the idea behind a class of neural network architectures known as autoencoders, which involve training neural networks to *reconstruct* the training data, forcing the network to learn the representation of each data point most helpful for this reconstruction.

DIAGRAM

Figure 7-2 shows a high-level overview of an autoencoder:

1. One set of layers transforms the data into a compressed representation of the data.
2. Another set of layers transforms this representation into an output of the same size and shape as the original data.

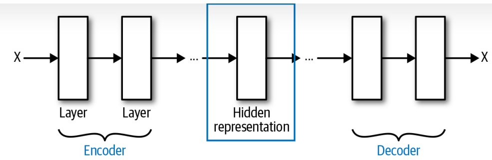

Figure 7-2. An autoencoder has one set of layers (which can be thought of as the "encoder" network) that maps the input to a lower-dimensional representation, and another set of layers (which can be thought of as the "decoder" network) that maps the lower-dimensional representation back to the input; this structure forces the network to learn a lower-dimensional representation that is most useful for reconstructing the input

Implementing such an architecture illustrates some features of PyTorch we haven't had a chance to introduce yet.

Implementing an Autoencoder in PyTorch

We'll now show a simple autoencoder that takes in an input image, feeds it through two convolutional layers and then a `Dense` layer to generate a representation, and then feeds this representation back through a `Dense` layer and two convolutional layers to generate an output of the same size as the input. We'll use this to illustrate two common practices when implementing more advanced architectures in PyTorch. First, we can include `PyTorchModel`s as attributes of another `PyTorchModel`, just as we defined `PyTorchLayer`s as attributes of such models previously. In the following example, we'll implement our autoencoder as having two `PyTorchModel`s as attributes: an `Encoder` and a `Decoder`. Once we train the model, we'll be able to use the trained `Encoder` as its own model to generate the representations.

We define the `Encoder` as:

```
class Encoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Encoder, self).__init__()
        self.conv1 = ConvLayer(1, 14, activation=nn.Tanh())
        self.conv2 = ConvLayer(14, 7, activation=nn.Tanh(), flatten=T

        self.dense1 = DenseLayer(7 * 28 * 28, hidden_dim, activation=

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)
        x = self.dense1(x)

        return x
```

And we define the `Decoder` as:

```
class Decoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Decoder, self).__init__()
        self.dense1 = DenseLayer(hidden_dim, 7 * 28 * 28, activation=

        self.conv1 = ConvLayer(7, 14, activation=nn.Tanh())
```

```python
        x = self.dense1(x)

        x = x.view(-1, 7, 28, 28)
        x = self.conv1(x)
        x = self.conv2(x)

        return x
```

> **NOTE**
>
> If we were using a stride greater than 1, we wouldn't simply be able to use a regular convolution to transform the encoding into an output, as we do here, but instead would have to use a *transposed convolution*, where the image size of the output of the operation would be larger than the image size of the input. See the `nn.ConvTranspose2d` operation in the PyTorch documentation for more.

Then the `Autoencoder` itself can wrap around these and become:

```python
class Autoencoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Autoencoder, self).__init__()

        self.encoder = Encoder(hidden_dim)

        self.decoder = Decoder(hidden_dim)

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        encoding = self.encoder(x)
        x = self.decoder(encoding)

        return x, encoding
```

The `forward` method of the `Autoencoder` illustrates a second common practice in PyTorch: since we'll ultimately want to see the hidden representation that the model produces, the `forward` method returns *two* elements: this "encoding," `encoding`, along with the output that will be used to train the network, `x`.

Of course, we would have to modify our `Trainer` class to accommodate this; specifically, `PyTorchModel` as currently written outputs only a single `Tensor` from its `forward` method. As it turns out, modifying it so that it returns a `Tuple` of `Tensor`s by default, even if that `Tuple` is only of length 1, will both be useful—enabling us to easily write models like the `Autoencoder`—and not difficult. All we have to do is three small things: first, make the function signature of the `forward` method of our base `PyTorchModel` class:

```python
def forward(self, x: Tensor) -> Tuple[Tensor]:
```

Then, at the end of the `forward` method of any model that inherits from the `PyTorchModel` base class, we'll write `return x,` instead of `return x` as we were doing before.

Second, we'll modify our `Trainer` to always take as output the first element of whatever the model returns:

```
    output = self.model(X_test)[0]
```

There is one other notable feature of the `Autoencoder` model: we apply a `Tanh` activation function to the last layer, meaning the model output will be between –1 and 1. With any model, the model outputs should be on the same scale as the target they are compared to, and here, the target is our input itself. So we should scale our input to range from a minimum of –1 and a maximum of 1, as in the following code:

```
X_train_auto = (X_train - X_train.min())
                / (X_train.max() - X_train.min()) * 2 - 1
X_test_auto = (X_test - X_train.min())
                / (X_train.max() - X_train.min()) * 2 - 1
```

Finally, we can train our model using training code, which by now should look familiar (we somewhat arbitrarily use 28 as the dimensionality of the output of the encoding):

```
model = Autoencoder(hidden_dim=28)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

trainer = PyTorchTrainer(model, optimizer, criterion)

trainer.fit(X_train_auto, X_train_auto,
            X_test_auto, X_test_auto,
            epochs=1,
            batch_size=60)
```

Once we run this code and train the model, we can look at both the reconstructed images and the image representations simply by passing `X_test_auto` through the model (since the `forward` method was defined to return two quantities):

```
reconstructed_images, image_representations = model(X_test_auto)
```

Each element of `reconstructed_images` is a `[1, 28, 28]` `Tensor` and represents the neural network's best attempt to reconstruct the corresponding original image after passing it through an autoencoder architecture that forced the image through a layer with lower dimensionality. Figure 7-3 shows a randomly chosen reconstructed image alongside the original image.

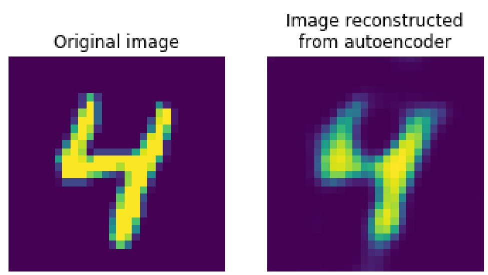

Figure 7-3. An image from the MNIST test set alongside the reconstruction of that image after it was fed through the autoencoder

Visually, the images look similar, telling us that the neural network does indeed seem to have taken the original images, which were 784 pixels, and mapped them to a space of lower dimensionality—specifically, 28—such that most of the information about the 784-pixel image is encoded in this vector of length 28. How can we examine the whole dataset to see whether the neural network has dimensional space should ideally be of the same digit, or at least visually be very similar, since visual similarity is how we as humans distinguish between different images. We can test whether this is the case by applying a dimensionality reduction technique invented by Laurens van der Maaten when he was a graduate student under Geoffrey Hinton (who was one of the "founding fathers" of neural networks): *t-Distributed Stochastic Neighbor Embedding*, or t-SNE. t-SNE performs its dimensionality reduction in a way that is analogous to how neural networks are trained: it starts with an initial lower-dimensional representation and then updates it so that, over time, it approaches a solution with the property that points that are "close together" in the high-dimensional space are "close together" in the low-dimensional space, and vice versa.[4]

We'll try the following:

- Feed the 10,000 images through t-SNE and reduce the dimensionality to 2.
- Visualize the resulting two-dimensional space, coloring the different points by their *actual* label (which the autoencoder did not see).

Figure 7-4 shows the result.

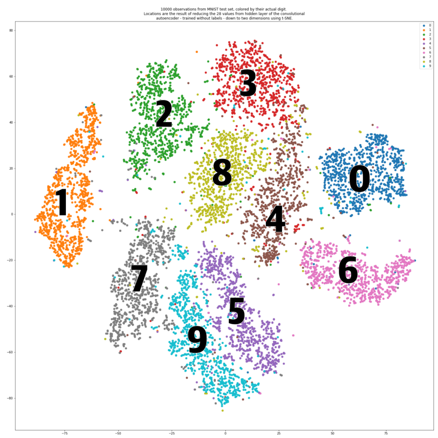

Figure 7-4. Result of running t-SNE on 28-dimensional learned space of the autoencoder

It appears that images of each digit are largely grouped together in their own separate cluster; this shows that training our autoencoder architecture to learn to reconstruct the original images from just a lower-dimensional representation has indeed enabled it to discover much of the underlying structure of these images without seeing any labels.[5] And not only are the 10 digits represented as distinct clusters, but visually similar digits are also closer together: at the top and slightly to the right, we have clusters of the digits 3, 5, and 8, and at the bottom we see 4 and 9 clustered tightly together, with 7 not far away. Finally, the most distinct digits—0, 1, and 6—form the most distinct clusters.

A Stronger Test for Unsupervised Learning, and a Solution

What we've just seen is a fairly weak test for whether our model has learned an underlying structure to the space of input images—by this point, it shouldn't be too surprising that a convolutional neural network can learn representations of images of digits with the property that visually similar images have similar representations. A stronger test would be to examine if the neural network has discovered a "smooth" underlying space: a space in which *any* vector of length 28, rather than just the vectors resulting from feeding real digits through the random vectors of length 28 and feeding them through the decoder network, using the fact that the `Autoencoder` contained a `Decoder` as an attribute:

```
test_encodings = np.random.uniform(low=-1.0, high=1.0, size=(5, 28))
test_imgs = model.decoder(Tensor(test_encodings))
```

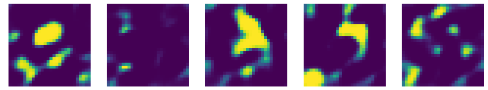

Figure 7-5. Result of feeding five randomly generated vectors through the decoder

You can see that the resulting images don't look like digits; thus, while our autoencoder can map our data to a lower-dimensional space in a sensible way, it doesn't appear to be able to learn a "smooth" space such as the one described a moment ago.

Solving the problem, of training a neural network to learn to represent images in a training set in a "smooth" underlying space, is one of the major accomplishments of *generative adversarial networks* (GANs). Invented in 2014, GANs are most widely known for allowing neural networks to generate realistic-looking images via a training procedure in which two neural networks are trained simultaneously. GANs were truly pushed forward in 2015, however, when researchers used them with deep convolutional architectures in both networks not just to generate realistic-looking 64 × 64 color images of bedrooms but also to generate a large sample of said images from randomly generated 100-dimensional vectors.[6] This signaled that the neural networks really had learned an underlying representation of the "space" of these unlabeled images. GANs deserve a book of their own, so we won't cover them in more detail than this.

Conclusion

You now have a deep understanding of the mechanics of some of the most popular advanced deep learning architectures out there, as well as how to implement these architectures in one of the most popular high-performance deep learning frameworks. The only thing stopping you from using deep learning models to solve real-world problems is practice. Luckily, it has never been easier to read others' code and quickly get up to speed on the details and implementation tricks that make certain model architectures work on certain problems. A list of recommended next steps is listed in the book's GitHub repo.

Onward!

---

1 Writing `Layer`s and `Model`s in this way isn't the most common or recommended use of PyTorch; we show it here because it most closely maps to the concepts we've covered so far. To see a more common way to build neural network building blocks with PyTorch, see this introductory tutorial from the official documentation.

2 In the book's GitHub repo, you can find an example of code that implements exponential learning rate decay as part of a `PyTorchTrainer`. The documentation for the `ExponentialLR` class used there can be found on the PyTorch website.

3 Look in the "CNNs using PyTorch" section.

4 The original 2008 paper is "Visualizing Data using t-SNE", by Laurens van der Maaten and Geoffrey Hinton.

training neural networks, such as learning rate decay, since we're training for only one epoch. This illustrates that the underlying idea of using an autoencoder-like architecture to learn the structure of a dataset without labels is a good one in general and didn't just "happen to work" here.

6 Check out the DCGAN paper, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" by Alec Radford et al., as well as this PyTorch documentation.