# 13

# Going Deeper – The Mechanics of PyTorch

In *Chapter 12, Parallelizing Neural Network Training with PyTorch*, we covered how to define and manipulate tensors and worked with the `torch.utils.data` module to build input pipelines. We further built and trained a multilayer perceptron to classify the Iris dataset using the PyTorch neural network module (`torch.nn`).

Now that we have some hands-on experience with PyTorch neural network training and machine learning, it's time to take a deeper dive into the PyTorch library and explore its rich set of features, which will allow us to implement more advanced deep learning models in upcoming chapters.

In this chapter, we will use different aspects of PyTorch's API to implement NNs. In particular, we will again use the `torch.nn` module, which provides multiple layers of abstraction to make the implementation of standard architectures very convenient. It also allows us to implement custom NN layers, which is very useful in research-oriented projects that require more customization. Later in this chapter, we will implement such a custom layer.

To illustrate the different ways of model building using the `torch.nn` module, we will also consider the classic **exclusive or** (**XOR**) problem. Firstly, we will build multilayer perceptrons using the `Sequential` class. Then, we will consider other methods, such as subclassing `nn.Module` for defining custom layers. Finally, we will work on two real-world projects that cover the machine learning steps from raw input to prediction.

The topics that we will cover are as follows:

- Understanding and working with PyTorch computation graphs
- Working with PyTorch tensor objects
- Solving the classic XOR problem and understanding model capacity
- Building complex NN models using PyTorch's `Sequential` class and the `nn.Module` class
- Computing gradients using automatic differentiation and `torch.autograd`

## The key features of PyTorch

In the previous chapter, we saw that PyTorch provides us with a scalable, multiplatform programming interface for implementing and running machine learning algorithms. After its initial release in 2016 and its 1.0 release in 2018, PyTorch has evolved into one of the two most popular frameworks for deep learning. It uses dynamic computational graphs, which have the advantage of being more flexible compared to its static counterparts. Dynamic computational graphs are debugging friendly: PyTorch allows for interleaving the graph declaration and graph evaluation steps. You can execute the code line by line while having full access to all variables. This is a very important feature that makes the development and training of NNs very convenient.

While PyTorch is an open-source library and can be used for free by everyone, its development is funded and supported by Facebook. This involves a large team of software engineers who expand and improve the library continuously. Since PyTorch is an open-source library, it also has strong support from other developers outside of Facebook, who avidly contribute and provide user feedback. This has made the PyTorch library more useful to both academic researchers and developers. A further consequence of these factors is that PyTorch has extensive documentation and tutorials to help new users.

Another key feature of PyTorch, which was also noted in the previous chapter, is its ability to work with single or multiple **graphical processing units** (**GPUs**). This allows users to train deep learning models very efficiently on large datasets and large-scale systems.

Last but not least, PyTorch supports mobile deployment, which also makes it a very suitable tool for production.

In the next section, we will look at how a tensor and function in PyTorch are interconnected via a computation graph.

## PyTorch's computation graphs

PyTorch performs its computations based on a **directed acyclic graph** (**DAG**). In this section, we will see how these graphs can be defined for a simple arithmetic computation. Then, we will see the dynamic graph paradigm, as well as how the graph is created on the fly in PyTorch.

### Understanding computation graphs

PyTorch relies on building a computation graph at its core, and it uses this computation graph to derive relationships between tensors from the input all the way to the output. Let's say that we have rank 0 (scalar) tensors $a$, $b$, and $c$ and we want to evaluate $z = 2 \times (a - b) + c$.

This evaluation can be represented as a computation graph, as shown in *Figure 13.1*:

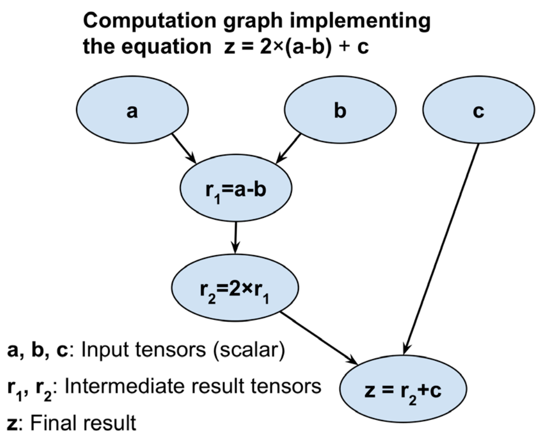

Computation graph implementing the equation z = 2×(a-b) + c
a
b
c
$r_1$=a-b
$r_2$=2×$r_1$
z = $r_2$+c
**a, b, c**: Input tensors (scalar)
**$r_1$, $r_2$**: Intermediate result tensors
**z**: Final result

*Figure 13.1: How a computation graph works*

As you can see, the computation graph is simply a network of nodes. Each node resembles an operation, which applies a function to its input tensor or tensors and returns zero or more tensors as the output. PyTorch builds this computation graph and uses it to compute the gradients accordingly. In the next subsection, we will see some examples of creating a graph for this computation using PyTorch.

## Creating a graph in PyTorch

Let's look at a simple example that illustrates how to create a graph in PyTorch for evaluating $z = 2 \times (a - b) + c$, as shown in the previous figure. The variables $a$, $b$, and $c$ are scalars (single numbers), and we define these as PyTorch tensors. To create the graph, we can simply define a regular Python function with `a`, `b`, and `c` as its input arguments, for example:

```
>>> import torch
>>> def compute_z(a, b, c):
...     r1 = torch.sub(a, b)
...     r2 = torch.mul(r1, 2)
...     z = torch.add(r2, c)
...     return z
```

Now, to carry out the computation, we can simply call this function with tensor objects as function arguments. Note that PyTorch functions such as `add`, `sub` (or `subtract`), and `mul` (or `multiply`) also allow us to provide inputs of higher ranks in the form of a PyTorch tensor object. In the following code example, we provide scalar inputs (rank 0), as well as rank 1 and rank 2 inputs, as lists:

```
>>> print('Scalar Inputs:', compute_z(torch.tensor(1),
...       torch.tensor(2), torch.tensor(3)))
Scalar Inputs: tensor(1)
>>> print('Rank 1 Inputs:', compute_z(torch.tensor([1]),
...       torch.tensor([2]), torch.tensor([3])))
Rank 1 Inputs: tensor([1])
>>> print('Rank 2 Inputs:', compute_z(torch.tensor([[1]]),
...       torch.tensor([[2]]), torch.tensor([[3]])))
Rank 2 Inputs: tensor([[1]])
```

In this section, you saw how simple it is to create a computation graph in PyTorch. Next, we will look at PyTorch tensors that can be used for storing and updating model parameters.

## PyTorch tensor objects for storing and updating model parameters

We covered tensor objects in *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*. In PyTorch, a special tensor object for which gradients need to be computed allows us to store and update the parameters of our models during training. Such a tensor can be created by just assigning `requires_grad` to `True` on user-specified initial values. Note that as of now (mid-2021), only tensors of floating point and complex `dtype` can require gradients. In the following code, we will generate tensor objects of type `float32`:

```
>>> a = torch.tensor(3.14, requires_grad=True)
>>> print(a)
tensor(3.1400, requires_grad=True)
>>> b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> print(b)
tensor([1., 2., 3.], requires_grad=True)
```

Notice that `requires_grad` is set to `False` by default. This value can be efficiently set to `True` by running `requires_grad_()`.

`method_()` is an in-place method in PyTorch that is used for operations without making a copy of the input.

Let's take a look at the following example:

```
>>> w = torch.tensor([1.0, 2.0, 3.0])
>>> print(w.requires_grad)
False
>>> w.requires_grad_()
>>> print(w.requires_grad)
True
```

You will recall that for NN models, initializing model parameters with random weights is necessary to break the symmetry during backpropagation—otherwise, a multilayer NN would be no more useful than a single-layer NN like logistic regression. When creating a PyTorch tensor, we can also use a random initialization scheme. PyTorch can generate random numbers based on a variety of probability distributions (see `https://pytorch.org/docs/stable/torch.html#random-sampling`). In the following example, we will take a look at some standard initialization methods that are also available in the `torch.nn.init` module (see `https://pytorch.org/docs/stable/nn.init.html`).

So, let's look at how we can create a tensor with Glorot initialization, which is a classic random initialization scheme that was proposed by Xavier Glorot and Yoshua Bengio. For this, we first create an empty tensor and an operator called `init` as an object of class `GlorotNormal`. Then, we fill this tensor with values according to the Glorot initialization by calling the `xavier_normal_()` method. In the following example, we initialize a tensor of shape 2×3:

```
>>> import torch.nn as nn
>>> torch.manual_seed(1)
>>> w = torch.empty(2, 3)
>>> nn.init.xavier_normal_(w)
>>> print(w)
tensor([[ 0.4183,  0.1688,  0.0390],
        [ 0.3930, -0.2858, -0.1051]])
```

**Xavier (or Glorot) initialization**

In the early development of deep learning, it was observed that random uniform or random normal weight initialization could often result in poor model performance during training.

In 2010, Glorot and Bengio investigated the effect of initialization and proposed a novel, more robust initialization scheme to facilitate the training of deep networks. The general idea behind Xavier initialization is to roughly balance the variance of the gradients across different layers. Otherwise, some layers may get too much attention during training while the other layers lag behind.

According to the research paper by Glorot and Bengio, if we want to initialize the weights in a uniform distribution, we should choose the interval of this uniform distribution as follows:

$$W \sim Uniform\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$$

Here, $n_{in}$ is the number of input neurons that are multiplied by the weights, and $n_{out}$ is the number of output neurons that feed into the next layer. For initializing the weights from Gaussian (normal) distribution, we recommend that you choose the standard deviation of this Gaussian to be:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in}+n_{out}}}$$

PyTorch supports Xavier initialization in both uniform and normal distributions of weights.

For more information about Glorot and Bengio's initialization scheme, including the rationale and mathematical motivation, we recommend the original paper (*Understanding the difficulty of deep feedforward neural networks*, *Xavier Glorot* and *Yoshua Bengio*, 2010), which is freely available at `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`.

Now, to put this into the context of a more practical use case, let's see how we can define two `Tensor` objects inside the base `nn.Module` class:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.w1 = torch.empty(2, 3, requires_grad=True)
...         nn.init.xavier_normal_(self.w1)
...         self.w2 = torch.empty(1, 2, requires_grad=True)
...         nn.init.xavier_normal_(self.w2)
```

These two tensors can be then used as weights whose gradients will be computed via automatic differentiation.

## Computing gradients via automatic differentiation

As you already know, optimizing NNs requires computing the gradients of the loss with respect to the NN weights. This is required for optimization algorithms such as **stochastic gradient descent** (**SGD**). In addition, gradients have other applications, such as diagnosing the network to find out why an NN model is making a particular prediction for a test example. Therefore, in this section, we will cover how to compute gradients of a computation with respect to its input variables.

### Computing the gradients of the loss with respect to trainable variables

PyTorch supports *automatic differentiation*, which can be thought of as an implementation of the *chain rule* for computing gradients of nested functions. Note that for the sake of simplicity, we will use the term *gradient* to refer to both partial derivatives and gradients.

**Partial derivatives and gradients**

A partial derivative $\frac{\partial f}{\partial x_1}$ can be understood as the rate of change of a multivariate function—a function with multiple inputs, $f(x_1, x_2, ...)$, with respect to one of its inputs (here: $x_1$). The gradient, $\nabla f$, of a function is a vector composed of all the inputs' partial derivatives, $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots\right)$.

When we define a series of operations that results in some output or even intermediate tensors, PyTorch provides a context for calculating gradients of these computed tensors with respect to its dependent nodes in the computation graph. To compute these gradients, we can call the `backward` method from the `torch.autograd` module. It computes the sum of gradients of the given tensor with regard to leaf nodes (terminal nodes) in the graph.

Let's work with a simple example where we will compute $z = wx + b$ and define the loss as the squared loss between the target $y$ and prediction $z$, $Loss = (y - z)^2$. In the more general case, where we may have multiple predictions and targets, we compute the loss as the sum of the squared error, $Loss = \sum_i (y_i - z_i)^2$. In order to implement this computation in PyTorch, we will define the model parameters, $w$ and $b$, as variables (tensors with the `requires_gradient` attribute set to `True`), and the input, $x$ and $y$, as default tensors. We will compute the loss tensor and use it to compute the gradients of the model parameters, $w$ and $b$, as follows:

```
>>> w = torch.tensor(1.0, requires_grad=True)
>>> b = torch.tensor(0.5, requires_grad=True)
>>> x = torch.tensor([1.4])
>>> y = torch.tensor([2.1])
>>> z = torch.add(torch.mul(w, x), b)
>>> loss = (y-z).pow(2).sum()
>>> loss.backward()
>>> print('dL/dw : ', w.grad)
>>> print('dL/db : ', b.grad)
dL/dw :  tensor(-0.5600)
dL/db :  tensor(-0.4000)
```

Computing the value $z$ is a forward pass in an NN. We used the `backward` method on the `loss` tensor to compute $\frac{\partial Loss}{\partial w}$ and $\frac{\partial Loss}{\partial b}$. Since this is a very simple example, we can obtain $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$ symbolically to verify that the computed gradients match the results we obtained in the previous code example:

```
>>> # verifying the computed gradient
>>> print(2 * x * ((w * x + b) - y))
tensor([-0.5600], grad_fn=<MulBackward0>)
```

We leave the verification of $b$ as an exercise for the reader.

## Understanding automatic differentiation

Automatic differentiation represents a set of computational techniques for computing gradients of arbitrary arithmetic operations. During this process, gradients of a computation (expressed as a series of operations) are obtained by accumulating the gradients through repeated applications of the chain rule. To better understand the concept behind automatic differentiation, let's consider a series of nested computations, $y = f(g(h(x)))$, with input $x$ and output $y$. This can be broken into a series of steps:

- $u_0 = x$
- $u_1 = h(x)$
- $u_2 = g(u_1)$
- $u_3 = f(u_2) = y$

The derivative $\frac{dy}{dx}$ can be computed in two different ways: forward accumulation, which starts with $\frac{du_3}{dx} = \frac{du_3}{du_2}\frac{du_2}{du_0}$, and reverse accumulation, which starts with $\frac{dy}{du_0} = \frac{dy}{du_1}\frac{du_1}{du_0}$. Note that PyTorch uses the latter, reverse accumulation, which is more efficient for implementing backpropagation.

## Adversarial examples

Computing gradients of the loss with respect to the input example is used for generating *adversarial examples* (or *adversarial attacks*). In computer vision, adversarial examples are examples that are generated by adding some small, imperceptible noise (or perturbations) to the input example, which results in a deep NN misclassifying them. Covering adversarial examples is beyond the scope of this book, but if you are interested, you can find the original paper by *Christian Szegedy et al.*, *Intriguing properties of neural networks* at `https://arxiv.org/pdf/1312.6199.pdf`.

## Simplifying implementations of common architectures via the torch.nn module

You have already seen some examples of building a feedforward NN model (for instance, a multilayer perceptron) and defining a sequence of layers using the `nn.Module` class. Before we take a deeper dive into `nn.Module`, let's briefly look at another approach for conjuring those layers via `nn.Sequential`.

### Implementing models based on nn.Sequential

With `nn.Sequential` (`https://pytorch.org/docs/master/generated/torch.nn.Sequential.html#sequential`), the layers stored inside the model are connected in a cascaded way. In the following example, we will build a model with two densely (fully) connected layers:

```
>>> model = nn.Sequential(
...     nn.Linear(4, 16),
...     nn.ReLU(),
...     nn.Linear(16, 32),
...     nn.ReLU()
... )
>>> model
Sequential(
  (0): Linear(in_features=4, out_features=16, bias=True)
  (1): ReLU()
  (2): Linear(in_features=16, out_features=32, bias=True)
  (3): ReLU()
)
```

We specified the layers and instantiated the `model` after passing the layers to the `nn.Sequential` class. The output of the first fully connected layer is used as the input to the first ReLU layer. The output of the first ReLU layer becomes the input for the second fully connected layer. Finally, the output of the second fully connected layer is used as the input to the second ReLU layer.

We can further configure these layers, for example, by applying different activation functions, initializers, or regularization methods to the parameters. A comprehensive and complete list of available options for most of these categories can be found in the official documentation:

- Choosing activation functions: `https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity`
- Initializing the layer parameters via `nn.init`: `https://pytorch.org/docs/stable/nn.init.html`
- Applying L2 regularization to the layer parameters (to prevent overfitting) via the parameter `weight_decay` of some optimizers in `torch.optim`: `https://pytorch.org/docs/stable/optim.html`

- Applying L1 regularization to the layer parameters (to prevent overfitting) by adding the L1 penalty term to the loss tensor, which we will implement next

In the following code example, we will configure the first fully connected layer by specifying the initial value distribution for the weight. Then, we will configure the second fully connected layer by computing the L1 penalty term for the weight matrix:

```
>>> nn.init.xavier_uniform_(model[0].weight)
>>> l1_weight = 0.01
>>> l1_penalty = l1_weight * model[2].weight.abs().sum()
```

Here, we initialized the weight of the first linear layer with Xavier initialization. And we computed the L1 norm of the weight of the second linear layer.

Furthermore, we can also specify the type of optimizer and the loss function for training. Again, a comprehensive list of all available options can be found in the official documentation:

- Optimizers via `torch.optim`: `https://pytorch.org/docs/stable/optim.html#algorithms`
- Loss functions: `https://pytorch.org/docs/stable/nn.html#loss-functions`

## Choosing a loss function

Regarding the choices for optimization algorithms, SGD and Adam are the most widely used methods. The choice of loss function depends on the task; for example, you might use mean square error loss for a regression problem.

The family of cross-entropy loss functions supplies the possible choices for classification tasks, which are extensively discussed in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*.

Furthermore, you can use the techniques you have learned from previous chapters (such as techniques for model evaluation from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*) combined with the appropriate metrics for the problem. For example, precision and recall, accuracy, **area under the curve** (AUC), and false negative and false positive scores are appropriate metrics for evaluating classification models.

In this example, we will use the SGD optimizer, and cross-entropy loss for binary classification:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Next, we will look at a more practical example: solving the classic XOR classification problem. First, we will use the `nn.Sequential()` class to build the model. Along the way, you will also learn about the capacity of a model for handling nonlinear decision boundaries. Then, we will cover building a model via `nn.Module` that will give us more flexibility and control over the layers of the network.

## Solving an XOR classification problem

The XOR classification problem is a classic problem for analyzing the capacity of a model with regard to capturing the nonlinear decision boundary between two classes. We generate a toy dataset of 200 training examples with two features $(x_0, x_1)$ drawn from a uniform distribution between [–1, 1). Then, we assign the ground truth label for training example $i$ according to the following rule:

$$y^{(i)} = \begin{cases} 0 & \text{if } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{otherwise} \end{cases}$$

We will use half of the data (100 training examples) for training and the remaining half for validation. The code for generating the data and splitting it into the training and validation datasets is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> torch.manual_seed(1)
>>> np.random.seed(1)
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1]<0] = 0
>>> n_train = 100
>>> x_train = torch.tensor(x[:n_train, :], dtype=torch.float32)
>>> y_train = torch.tensor(y[:n_train], dtype=torch.float32)
>>> x_valid = torch.tensor(x[n_train:, :], dtype=torch.float32)
>>> y_valid = torch.tensor(y[n_train:], dtype=torch.float32)
>>> fig = plt.figure(figsize=(6, 6))
>>> plt.plot(x[y==0, 0], x[y==0, 1], 'o', alpha=0.75, markersize=10)
>>> plt.plot(x[y==1, 0], x[y==1, 1], '<', alpha=0.75, markersize=10)
>>> plt.xlabel(r'$x_1$', size=15)
>>> plt.ylabel(r'$x_2$', size=15)
>>> plt.show()
```

The code results in the following scatterplot of the training and validation examples, shown with different markers based on their class label:

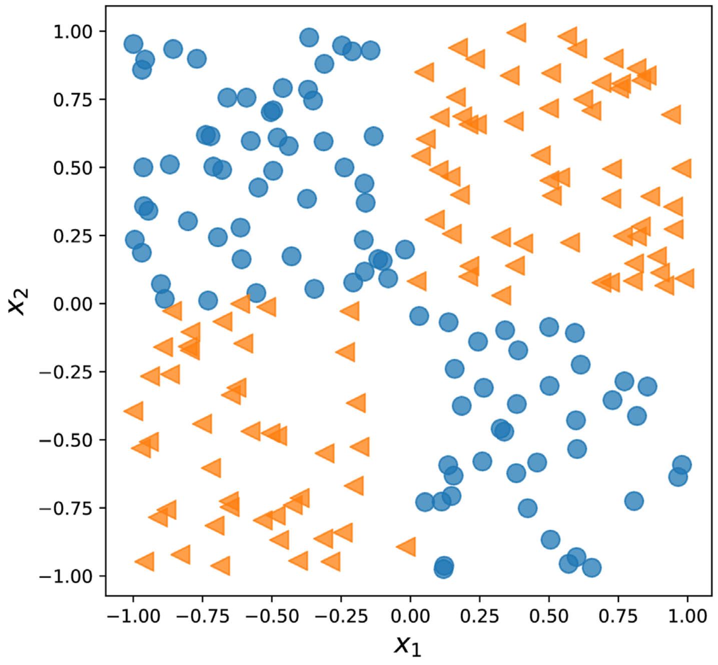


*Figure 13.2: Scatterplot of training and validation examples*

In the previous subsection, we covered the essential tools that we need to implement a classifier in PyTorch. We now need to decide what architecture we should choose for this task and dataset. As a general rule of thumb, the more layers we have, and the more neurons we have in each layer, the larger the capacity of the model will be. Here, the model capacity can be thought of as a measure of how readily the model can approximate complex functions. While having more parameters means the network can fit more complex functions, larger models are usually harder to train (and prone to overfitting). In practice, it is always a good idea to start with a simple model as a baseline, for example, a single-layer NN like logistic regression:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 1),
...     nn.Sigmoid()
... )
>>> model
Sequential(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

After defining the model, we will initialize the cross-entropy loss function for binary classification and the SGD optimizer:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Next, we will create a data loader that uses a batch size of 2 for the train data:

```
>>> from torch.utils.data import DataLoader, TensorDataset
>>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 2
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Now we will train the model for 200 epochs and record a history of training epochs:

```
>>> torch.manual_seed(1)
>>> num_epochs = 200
>>> def train(model, num_epochs, train_dl, x_valid, y_valid):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)[:, 0]
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()
...             is_correct = ((pred>=0.5).float() == y_batch).float()
...             accuracy_hist_train[epoch] += is_correct.mean()
...         loss_hist_train[epoch] /= n_train/batch_size
...         accuracy_hist_train[epoch] /= n_train/batch_size
...         pred = model(x_valid)[:, 0]
...         loss = loss_fn(pred, y_valid)
...         loss_hist_valid[epoch] = loss.item()
...         is_correct = ((pred>=0.5).float() == y_valid).float()
...         accuracy_hist_valid[epoch] += is_correct.mean()
...     return loss_hist_train, loss_hist_valid, \
...            accuracy_hist_train, accuracy_hist_valid
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Notice that the history of training epochs includes the train loss and validation loss and the train accuracy and validation accuracy, which is useful for visual inspection after training. In the following code, we will plot the learning curves, including the training and validation loss, as well as their accuracies.

The following code will plot the training performance:

```
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
```

This results in the following figure, with two separate panels for the losses and accuracies:

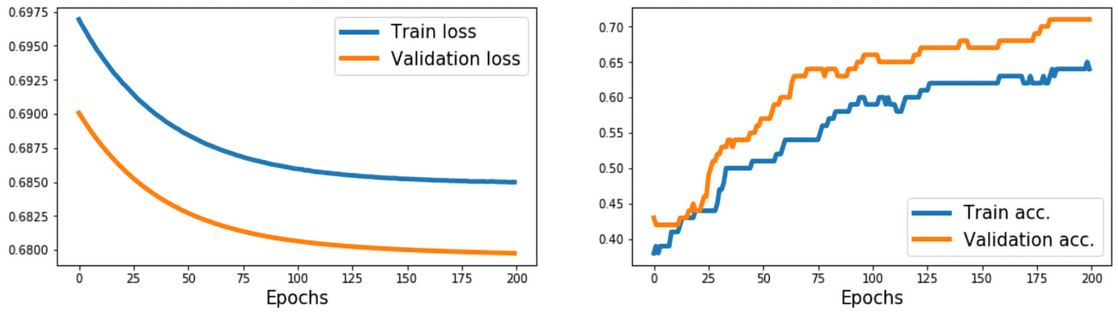


*Figure 13.3: Loss and accuracy results*

As you can see, a simple model with no hidden layer can only derive a linear decision boundary, which is unable to solve the XOR problem. As a consequence, we can observe that the loss terms for both the training and the validation datasets are very high, and the classification accuracy is very low.

To derive a nonlinear decision boundary, we can add one or more hidden layers connected via nonlinear activation functions. The universal approximation theorem states that a feedforward NN with a single hidden layer and a relatively large number of hidden units can approximate arbitrary continuous functions relatively well. Thus, one approach for tackling the XOR problem more satisfactorily is to add a hidden layer and compare different numbers of hidden units until we observe satisfactory results on the validation dataset. Adding more hidden units would correspond to increasing the width of a layer.

Alternatively, we can also add more hidden layers, which will make the model deeper. The advantage of making a network deeper rather than wider is that fewer parameters are required to achieve a comparable model capacity.

However, a downside of deep (versus wide) models is that deep models are prone to vanishing and exploding gradients, which make them harder to train.

As an exercise, try adding one, two, three, and four hidden layers, each with four hidden units. In the following example, we will take a look at the results of a feedforward NN with two hidden layers:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 4),
...     nn.ReLU(),
...     nn.Linear(4, 4),
...     nn.ReLU(),
...     nn.Linear(4, 1),
...     nn.Sigmoid()
... )
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> model
Sequential(
  (0): Linear(in_features=2, out_features=4, bias=True)
  (1): ReLU()
  (2): Linear(in_features=4, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
  (5): Sigmoid()
)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

We can repeat the previous code for visualization, which produces the following:

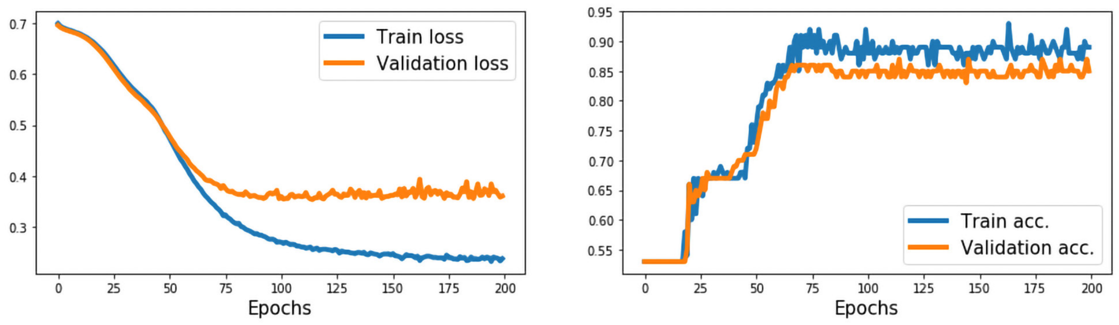


*Figure 13.4: Loss and accuracy results after adding two hidden layers*

Now, we can see that the model is able to derive a nonlinear decision boundary for this data, and the model reaches 100 percent accuracy on the training dataset. The validation dataset's accuracy is 95 percent, which indicates that the model is slightly overfitting.

## Making model building more flexible with nn.Module

In the previous example, we used the PyTorch `Sequential` class to create a fully connected NN with multiple layers. This is a very common and convenient way of building models. However, it unfortunately doesn't allow us to create more complex models that have multiple input, output, or intermediate branches. That's where `nn.Module` comes in handy.

The alternative way to build complex models is by subclassing `nn.Module`. In this approach, we create a new class derived from `nn.Module` and define the method, `__init__()`, as a constructor. The `forward()` method is used to specify the forward pass. In the constructor function, `__init__()`, we define the layers as attributes of the class so that they can be accessed via the `self` reference attribute. Then, in the `forward()` method, we specify how these layers are to be used in the forward pass of the NN. The code for defining a new class that implements the previous model is as follows:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         l1 = nn.Linear(2, 4)
...         a1 = nn.ReLU()
...         l2 = nn.Linear(4, 4)
...         a2 = nn.ReLU()
...         l3 = nn.Linear(4, 1)
...         a3 = nn.Sigmoid()
...         l = [l1, a1, l2, a2, l3, a3]
...         self.module_list = nn.ModuleList(l)
...
...     def forward(self, x):
...         for f in self.module_list:
...             x = f(x)
...         return x
```

Notice that we put all layers in the `nn.ModuleList` object, which is just a `list` object composed of `nn.Module` items. This makes the code more readable and easier to follow.

Once we define an instance of this new class, we can train it as we did previously:

```
>>> model = MyModule()
>>> model
MyModule(
  (module_list): ModuleList(
    (0): Linear(in_features=2, out_features=4, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4, out_features=4, bias=True)
    (3): ReLU()
    (4): Linear(in_features=4, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Next, besides the train history, we will use the mlxtend library to visualize the validation data and the decision boundary.

Mlxtend can be installed via `conda` or `pip` as follows:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

To compute the decision boundary of our model, we need to add a `predict()` method in the `MyModule` class:

```
>>>     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
```

It will return the predicted class (0 or 1) for a sample.

The following code will plot the training performance along with the decision region bias:

```
>>> from mlxtend.plotting import plot_decision_regions
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid.numpy(),
...                       y=y_valid.numpy().astype(np.integer),
...                       clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

This results in *Figure 13.5*, with three separate panels for the losses, accuracies, and the scatterplot of the validation examples, along with the decision boundary:

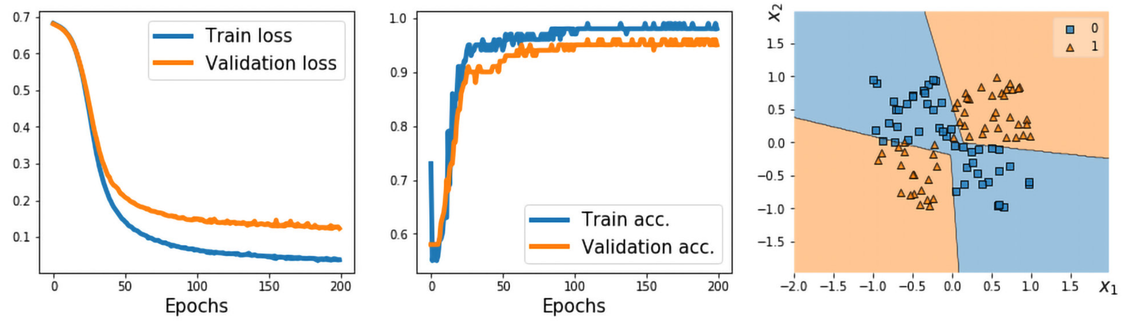


*Figure 13.5: Results, including a scatterplot*

## Writing custom layers in PyTorch

In cases where we want to define a new layer that is not already supported by PyTorch, we can define a new class derived from the `nn.Module` class. This is especially useful when designing a new layer or customizing an existing layer.

To illustrate the concept of implementing custom layers, let's consider a simple example. Imagine we want to define a new linear layer that computes $w(x + \epsilon) + b$, where $\epsilon$ refers to a random variable as a noise variable. To implement this computation, we define a new class as a subclass of `nn.Module`. For this new class, we have to define both the constructor `__init__()` method and the `forward()` method. In the constructor, we define the variables and other required tensors for our customized layer. We can create variables and initialize them in the constructor if the `input_size` is given to the constructor. Alternatively, we can delay the variable initialization (for instance, if we do not know the exact input shape upfront) and delegate it to another method for late variable creation.

To look at a concrete example, we are going to define a new layer called `NoisyLinear`, which implements the computation $w(x + \epsilon) + b$, which was mentioned in the preceding paragraph:

```
>>> class NoisyLinear(nn.Module):
...     def __init__(self, input_size, output_size,
...                  noise_stddev=0.1):
...         super().__init__()
...         w = torch.Tensor(input_size, output_size)
...         self.w = nn.Parameter(w)  # nn.Parameter is a Tensor
...                                   # that's a module parameter.
...         nn.init.xavier_uniform_(self.w)
...         b = torch.Tensor(output_size).fill_(0)
...         self.b = nn.Parameter(b)
...         self.noise_stddev = noise_stddev
...
...     def forward(self, x, training=False):
...         if training:
...             noise = torch.normal(0.0, self.noise_stddev, x.shape)
...             x_new = torch.add(x, noise)
...         else:
...             x_new = x
...         return torch.add(torch.mm(x_new, self.w), self.b)
```

In the constructor, we have added an argument, `noise_stddev`, to specify the standard deviation for the distribution of $\epsilon$, which is sampled from a Gaussian distribution. Furthermore, notice that in the `forward()` method, we have used an additional argument, `training=False`. We use it to distinguish whether the layer is used during training or only for prediction (this is sometimes also called *inference*) or evaluation. Also, there are certain methods that behave differently in training and prediction modes. You will encounter an example of such a method, `Dropout`, in the upcoming chapters. In the previous code snippet, we also specified that the random vector, $\epsilon$, was to be generated and added to the input during training only and not used for inference or evaluation.

Before we go a step further and use our custom `NoisyLinear` layer in a model, let's test it in the context of a simple example.

1. In the following code, we will define a new instance of this layer, and execute it on an input tensor. Then, we will call the layer three times on the same input tensor:

```
>>> torch.manual_seed(1)
>>> noisy_layer = NoisyLinear(4, 2)
>>> x = torch.zeros((1, 4))
>>> print(noisy_layer(x, training=True))
tensor([[ 0.1154, -0.0598]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=True))
tensor([[ 0.0432, -0.0375]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=False))
tensor([[0., 0.]], grad_fn=<AddBackward0>)
```

Note that the outputs for the first two calls differ because the `NoisyLinear` layer added random noise to the input tensor. The third call outputs [0, 0] as we didn't add noise by specifying `training=False`.

2. Now, let's create a new model similar to the previous one for solving the XOR classification task. As before, we will use the `nn.Module` class for model building, but this time, we will use our `NoisyLinear` layer as the first hidden layer of the multilayer perceptron. The code is as follows:

```
>>> class MyNoisyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.l1 = NoisyLinear(2, 4, 0.07)
...         self.a1 = nn.ReLU()
...         self.l2 = nn.Linear(4, 4)
...         self.a2 = nn.ReLU()
...         self.l3 = nn.Linear(4, 1)
...         self.a3 = nn.Sigmoid()
...
...     def forward(self, x, training=False):
...         x = self.l1(x, training)
...         x = self.a1(x)
...         x = self.l2(x)
...         x = self.a2(x)
...         x = self.l3(x)
...         x = self.a3(x)
...         return x
...
...     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
...
>>> torch.manual_seed(1)
>>> model = MyNoisyModule()
>>> model
MyNoisyModule(
  (l1): NoisyLinear()
  (a1): ReLU()
  (l2): Linear(in_features=4, out_features=4, bias=True)
```

```
    (a2): ReLU()
    (l3): Linear(in_features=4, out_features=1, bias=True)
    (a3): Sigmoid()
  )
```

3. Similarly, we will train the model as we did previously. At this time, to compute the prediction on the training batch, we use `pred = model(x_batch, True)[:, 0]` instead of `pred = model(x_batch)[:, 0]`:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> torch.manual_seed(1)
>>> loss_hist_train = [0] * num_epochs
>>> accuracy_hist_train = [0] * num_epochs
>>> loss_hist_valid = [0] * num_epochs
>>> accuracy_hist_valid = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch, True)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train[epoch] += loss.item()
...         is_correct = (
...             (pred>=0.5).float() == y_batch
...         ).float()
...         accuracy_hist_train[epoch] += is_correct.mean()
...     loss_hist_train[epoch] /= n_train/batch_size
...     accuracy_hist_train[epoch] /= n_train/batch_size
...     pred = model(x_valid)[:, 0]
...     loss = loss_fn(pred, y_valid)
...     loss_hist_valid[epoch] = loss.item()
...     is_correct = ((pred>=0.5).float() == y_valid).float()
...     accuracy_hist_valid[epoch] += is_correct.mean()
```

4. After the model is trained, we can plot the losses, accuracies, and the decision boundary:

```
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(loss_hist_train, lw=4)
>>> plt.plot(loss_hist_valid, lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(accuracy_hist_train, lw=4)
>>> plt.plot(accuracy_hist_valid, lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(
...     X=x_valid.numpy(),
...     y=y_valid.numpy().astype(np.integer),
...     clf=model
... )
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

5. The resulting figure will be as follows:

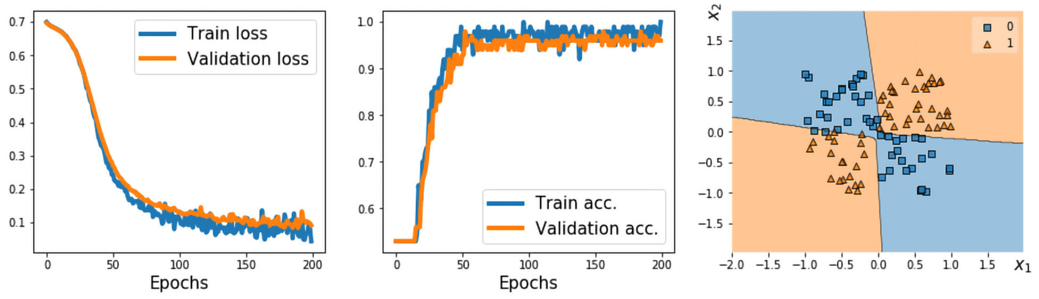


*Figure 13.6: Results using NoisyLinear as the first hidden layer*

Here, our goal was to learn how to define a new custom layer subclassed from `nn.Module` and to use it as we would use any other standard `torch.nn` layer. Although, with this particular example, `NoisyLinear` did not help to improve the performance, please keep in mind that our objective was to mainly learn how to write a customized layer from scratch. In general, writing a new customized layer can be useful in other applications, for example, if you develop a new algorithm that depends on a new layer beyond the existing ones.

## Project one – predicting the fuel efficiency of a car

So far, in this chapter, we have mostly focused on the `torch.nn` module. We used `nn.Sequential` to construct the models for simplicity. Then, we made model building more flexible with `nn.Module` and implemented feedforward NNs, to which we added customized layers. In this section, we will work on a real-world project of predicting the fuel efficiency of a car in miles per gallon (MPG). We will cover the underlying steps in machine learning tasks, such as data preprocessing, feature engineering, training, prediction (inference), and evaluation.

### Working with feature columns

In machine learning and deep learning applications, we can encounter various different types of features: continuous, unordered categorical (nominal), and ordered categorical (ordinal). You will recall that in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, we covered different types of features and learned how to handle each type. Note that while numeric data can be either continuous or discrete, in the context of machine learning with PyTorch, "numeric" data specifically refers to continuous data of floating point type.

Sometimes, feature sets are comprised of a mixture of different feature types. For example, consider a scenario with a set of seven different features, as shown in *Figure 13.7*:

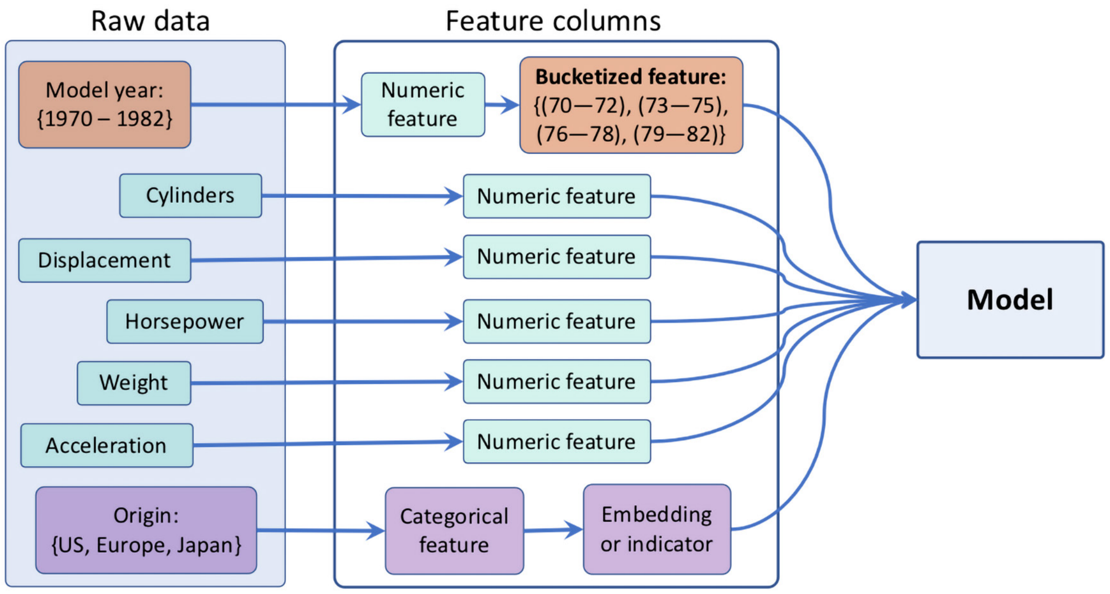


*Figure 13.7: Auto MPG data structure*

The features shown in the figure (model year, cylinders, displacement, horsepower, weight, acceleration, and origin) were obtained from the Auto MPG dataset, which is a common machine learning benchmark dataset for predicting the fuel efficiency of a car in MPG. The full dataset and its description are available from UCI's machine learning repository at `https://archive.ics.uci.edu/ml/datasets/auto+mpg`.

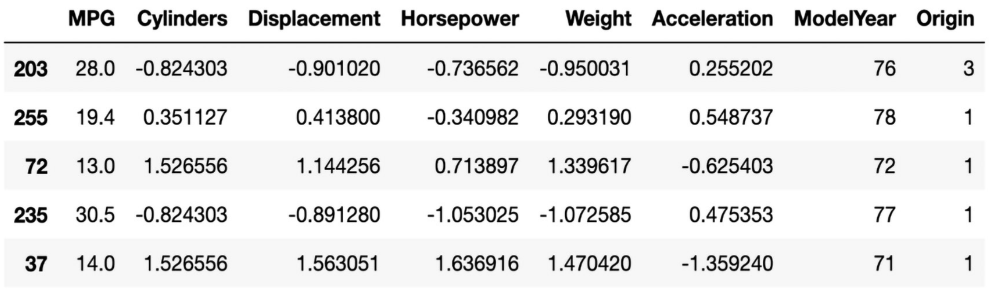
We are going to treat five features from the Auto MPG dataset (number of cylinders, displacement, horsepower, weight, and acceleration) as "numeric" (here, continuous) features. The model year can be regarded as an ordered categorical (ordinal) feature. Lastly, the manufacturing origin can be regarded as an unordered categorical (nominal) feature with three possible discrete values, 1, 2, and 3, which correspond to the US, Europe, and Japan, respectively.

Let's first load the data and apply the necessary preprocessing steps, including dropping the incomplete rows, partitioning the dataset into training and test datasets, as well as standardizing the continuous features:

```
>>> import pandas as pd
>>> url = 'http://archive.ics.uci.edu/ml/' \
...       'machine-learning-databases/auto-mpg/auto-mpg.data'
>>> column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
...                 'Weight', 'Acceleration', 'Model Year', 'Origin']
>>> df = pd.read_csv(url, names=column_names,
...                  na_values = "?", comment='\t',
...                  sep=" ", skipinitialspace=True)
>>>
>>> ## drop the NA rows
>>> df = df.dropna()
>>> df = df.reset_index(drop=True)
>>>
>>> ## train/test splits:
>>> import sklearn
>>> import sklearn.model_selection
>>> df_train, df_test = sklearn.model_selection.train_test_split(
...     df, train_size=0.8, random_state=1
... )
>>> train_stats = df_train.describe().transpose()
>>>
>>> numeric_column_names = [
...     'Cylinders', 'Displacement',
...     'Horsepower', 'Weight',
...     'Acceleration'
... ]
```

```
>>> df_train_norm, df_test_norm = df_train.copy(), df_test.copy()
>>> for col_name in numeric_column_names:
...     mean = train_stats.loc[col_name, 'mean']
...     std  = train_stats.loc[col_name, 'std']
...     df_train_norm.loc[:, col_name] = \
...         (df_train_norm.loc[:, col_name] - mean)/std
...     df_test_norm.loc[:, col_name] = \
...         (df_test_norm.loc[:, col_name] - mean)/std
>>> df_train_norm.tail()
```

This results in the following:



| | MPG | Cylinders | Displacement | Horsepower | Weight | Acceleration | ModelYear | Origin |
|---|---|---|---|---|---|---|---|---|
| **203** | 28.0 | -0.824303 | -0.901020 | -0.736562 | -0.950031 | 0.255202 | 76 | 3 |
| **255** | 19.4 | 0.351127 | 0.413800 | -0.340982 | 0.293190 | 0.548737 | 78 | 1 |
| **72** | 13.0 | 1.526556 | 1.144256 | 0.713897 | 1.339617 | -0.625403 | 72 | 1 |
| **235** | 30.5 | -0.824303 | -0.891280 | -1.053025 | -1.072585 | 0.475353 | 77 | 1 |
| **37** | 14.0 | 1.526556 | 1.563051 | 1.636916 | 1.470420 | -1.359240 | 71 | 1 |

*Figure 13.8: Preprocessed Auto MG data*

The pandas `DataFrame` that we created via the previous code snippet contains five columns with values of the type `float`. These columns will constitute the continuous features.

Next, let's group the rather fine-grained model year (`ModelYear`) information into buckets to simplify the learning task for the model that we are going to train later. Concretely, we are going to assign each car into one of four *year* buckets, as follows:

$$\text{bucket} = \begin{cases} 0 & \text{if year} < 73 \\ 1 & \text{if } 73 \leq \text{year} < 76 \\ 2 & \text{if } 76 \leq \text{year} < 79 \\ 3 & \text{if year} \geq 79 \end{cases}$$

Note that the chosen intervals were selected arbitrarily to illustrate the concepts of "bucketing." In order to group the cars into these buckets, we will first define three cut-off values: [73, 76, 79] for the model year feature. These cut-off values are used to specify half-closed intervals, for instance, $(-\infty, 73)$, [73, 76), [76, 79), and $[76, \infty)$. Then, the original numeric features will be passed to the `torch.bucketize` function (`https://pytorch.org/docs/stable/generated/torch.bucketize.html`) to generate the indices of the buckets. The code is as follows:

```
>>> boundaries = torch.tensor([73, 76, 79])
>>> v = torch.tensor(df_train_norm['Model Year'].values)
>>> df_train_norm['Model Year Bucketed'] = torch.bucketize(
...     v, boundaries, right=True
... )
>>> v = torch.tensor(df_test_norm['Model Year'].values)
>>> df_test_norm['Model Year Bucketed'] = torch.bucketize(
...     v, boundaries, right=True
... )
>>> numeric_column_names.append('Model Year Bucketed')
```

We added this bucketized feature column to the Python list `numeric_column_names`.

Next, we will proceed with defining a list for the unordered categorical feature, `Origin`. In PyTorch, There are two ways to work with a categorical feature: using an embedding layer via `nn.Embedding` (`https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`), or using one-hot-encoded vectors (also called *indicator*). In the encoding approach, for example, index 0 will be encoded as [1, 0, 0], index 1 will be encoded as [0, 1, 0], and so on. On the other hand, the embedding layer maps each index to a vector of random numbers of the type `float`, which can be trained. (You can think of the embedding layer as a more efficient implementation of a one-hot encoding multiplied with a trainable weight matrix.)

When the number of categories is large, using the embedding layer with fewer dimensions than the number of categories can improve the performance.

In the following code snippet, we will use the one-hot-encoding approach on the categorical feature in order to convert it into the dense format:

```
>>> from torch.nn.functional import one_hot
>>> total_origin = len(set(df_train_norm['Origin']))
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_train_norm['Origin'].values) % total_origin)
>>> x_train_numeric = torch.tensor(
...     df_train_norm[numeric_column_names].values)
>>> x_train = torch.cat([x_train_numeric, origin_encoded], 1).float()
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_test_norm['Origin'].values) % total_origin)
>>> x_test_numeric = torch.tensor(
...     df_test_norm[numeric_column_names].values)
>>> x_test = torch.cat([x_test_numeric, origin_encoded], 1).float()
```

After encoding the categorical feature into a three-dimensional dense feature, we concatenated it with the numeric features we processed in the previous step. Finally, we will create the label tensors from the ground truth MPG values as follows:

```
>>> y_train = torch.tensor(df_train_norm['MPG'].values).float()
>>> y_test = torch.tensor(df_test_norm['MPG'].values).float()
```

In this section, we have covered the most common approaches for preprocessing and creating features in PyTorch.

## Training a DNN regression model

Now, after constructing the mandatory features and labels, we will create a data loader that uses a batch size of 8 for the train data:

```
>>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 8
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Next, we will build a model with two fully connected layers where one has 8 hidden units and another has 4:

```
>>> hidden_units = [8, 4]
>>> input_size = x_train.shape[1]
>>> all_layers = []
>>> for hidden_unit in hidden_units:
...     layer = nn.Linear(input_size, hidden_unit)
...     all_layers.append(layer)
...     all_layers.append(nn.ReLU())
...     input_size = hidden_unit
>>> all_layers.append(nn.Linear(hidden_units[-1], 1))
>>> model = nn.Sequential(*all_layers)
>>> model
Sequential(
  (0): Linear(in_features=9, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
)
```

After defining the model, we will define the MSE loss function for regression and use stochastic gradient descent for optimization:

```
>>> loss_fn = nn.MSELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Now we will train the model for 200 epochs and display the train loss for every 20 epochs:

```
>>> torch.manual_seed(1)
>>> num_epochs = 200
>>> log_epochs = 20
>>> for epoch in range(num_epochs):
...     loss_hist_train = 0
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train += loss.item()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch}  Loss '
...               f'{loss_hist_train/len(train_dl):.4f}')

Epoch 0  Loss 536.1047
Epoch 20  Loss 8.4361
Epoch 40  Loss 7.8695
Epoch 60  Loss 7.1891
Epoch 80  Loss 6.7062
Epoch 100  Loss 6.7599
Epoch 120  Loss 6.3124
Epoch 140  Loss 6.6864
Epoch 160  Loss 6.7648
Epoch 180  Loss 6.2156
```

After 200 epochs, the train loss was around 5. We can now evaluate the regression performance of the trained model on the test dataset. To predict the target values on new data points, we can feed their features to the model:

```
>>> with torch.no_grad():
...     pred = model(x_test.float())[:, 0]
...     loss = loss_fn(pred, y_test)
...     print(f'Test MSE: {loss.item():.4f}')
...     print(f'Test MAE: {nn.L1Loss()(pred, y_test).item():.4f}')
Test MSE: 9.6130
Test MAE: 2.1211
```

The MSE on the test set is 9.6, and the **mean absolute error** (**MAE**) is 2.1. After this regression project, we will work on a classification project in the next section.

## Project two – classifying MNIST handwritten digits

For this classification project, we are going to categorize MNIST handwritten digits. In the previous section, we covered the four essential steps for machine learning in PyTorch in detail, which we will need to repeat in this section.

You will recall that in *Chapter 12* you learned the way of loading available datasets from the `torchvision` module. First, we are going to load the MNIST dataset using the `torchvision` module.

1. The setup step includes loading the dataset and specifying hyperparameters (the size of the train set and test set, and the size of mini-batches):

   ```
   >>> import torchvision
   >>> from torchvision import transforms
   >>> image_path = './'
   >>> transform = transforms.Compose([
   ...     transforms.ToTensor()
   ... ])
   >>> mnist_train_dataset = torchvision.datasets.MNIST(
   ...     root=image_path, train=True,
   ...     transform=transform, download=False
   ... )
   >>> mnist_test_dataset = torchvision.datasets.MNIST(
   ...     root=image_path, train=False,
   ...     transform=transform, download=False
   ... )
   >>> batch_size = 64
   >>> torch.manual_seed(1)
   >>> train_dl = DataLoader(mnist_train_dataset,
   ...                       batch_size, shuffle=True)
   ```

   Here, we constructed a data loader with batches of 64 samples. Next, we will preprocess the loaded datasets.

2. We preprocess the input features and the labels. The features in this project are the pixels of the images we read from **Step 1**. We defined a custom transformation using `torchvision.transforms.Compose`. In this simple case, our transformation consisted only of one method, `ToTensor()`. The `ToTensor()` method converts the pixel features into a floating type tensor and also normalizes the pixels from the [0, 255] to [0, 1] range. In *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*, we will see some additional data transformation methods when we work with more complex image datasets. The labels are integers from 0 to 9 representing ten digits. Hence, we don't need to do any scaling or further conversion. Note that we can access the raw pixels using the `data` attribute, and don't forget to scale them to the range [0, 1].

   We will construct the model in the next step once the data is preprocessed.

3. Construct the NN model:

```
>>> hidden_units = [32, 16]
>>> image_size = mnist_train_dataset[0][0].shape
>>> input_size = image_size[0] * image_size[1] * image_size[2]
>>> all_layers = [nn.Flatten()]
>>> for hidden_unit in hidden_units:
...     layer = nn.Linear(input_size, hidden_unit)
...     all_layers.append(layer)
...     all_layers.append(nn.ReLU())
...     input_size = hidden_unit
>>> all_layers.append(nn.Linear(hidden_units[-1], 10))
>>> model = nn.Sequential(*all_layers)
>>> model
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=32, bias=True)
  (2): ReLU()
  (3): Linear(in_features=32, out_features=16, bias=True)
  (4): ReLU()
  (5): Linear(in_features=16, out_features=10, bias=True)
)
```

Note that the model starts with a flatten layer that flattens an input image into a one-dimensional tensor. This is because the input images are in the shape of [1, 28, 28]. The model has two hidden layers, with 32 and 16 units respectively. And it ends with an output layer of ten units representing ten classes, activated by a softmax function. In the next step, we will train the model on the train set and evaluate it on the test set.

4. Use the model for training, evaluation, and prediction:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> for epoch in range(num_epochs):
...     accuracy_hist_train = 0
...     for x_batch, y_batch in train_dl:
```

```
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         is_correct = (
...             torch.argmax(pred, dim=1) == y_batch
...         ).float()
...         accuracy_hist_train += is_correct.sum()
...     accuracy_hist_train /= len(train_dl.dataset)
...     print(f'Epoch {epoch}  Accuracy '
...           f'{accuracy_hist_train:.4f}')
Epoch 0  Accuracy 0.8531
...
Epoch 9  Accuracy 0.9691
...
Epoch 19  Accuracy 0.9813
```

We used the cross-entropy loss function for multiclass classification and the Adam optimizer for gradient descent. We will talk about the Adam optimizer in *Chapter 14*. We trained the model for 20 epochs and displayed the train accuracy for every epoch. The trained model reached an accuracy of 96.3 percent on the training set and we will evaluate it on the testing set:

```
>>> pred = model(mnist_test_dataset.data / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) ==
...     mnist_test_dataset.targets
... ).float()
>>> print(f'Test accuracy: {is_correct.mean():.4f}')
Test accuracy: 0.9645
```

The test accuracy is 95.6 percent. You have learned how to solve a classification problem using PyTorch.

## Higher-level PyTorch APIs: a short introduction to PyTorch-Lightning

In recent years, the PyTorch community developed several different libraries and APIs on top of PyTorch. Notable examples include fastai (`https://docs.fast.ai/`), Catalyst (`https://github.com/catalyst-team/catalyst`), PyTorch Lightning (`https://www.pytorchlightning.ai`), (`https://lightning-flash.readthedocs.io/en/latest/quickstart.html`), and PyTorch-Ignite (`https://github.com/pytorch/ignite`).

In this section, we will explore PyTorch Lightning (Lightning for short), which is a widely used PyTorch library that makes training deep neural networks simpler by removing much of the boilerplate code. However, while Lightning's focus lies in simplicity and flexibility, it also allows us to use many advanced features such as multi-GPU support and fast low-precision training, which you can learn about in the official documentation at `https://pytorch-lightning.rtfd.io/en/latest/`.

There is also a bonus introduction to PyTorch-Ignite at `https://github.com/rasbt/machine-learning-book/blob/main/ch13/ch13_part4_ignite.ipynb`.

In an earlier section, *Project two – classifying MNIST handwritten digits*, we implemented a multilayer perceptron for classifying handwritten digits in the MNIST dataset. In the next subsections, we will reimplement this classifier using Lightning.

**Installing PyTorch Lightning**

Lightning can be installed via pip or conda, depending on your preference. For instance, the command for installing Lightning via pip is as follows:

```
pip install pytorch-lightning
```

The following is the command for installing Lightning via conda:

```
conda install pytorch-lightning -c conda-forge
```

The code in the following subsections is based on PyTorch Lightning version 1.5, which you can install by replacing `pytorch-lightning` with `pytorch-lightning==1.5` in these commands.

## Setting up the PyTorch Lightning model

We start by implementing the model, which we will train in the next subsections. Defining a model for Lightning is relatively straightforward as it is based on regular Python and PyTorch code. All that is required to implement a Lightning model is to use `LightningModule` instead of the regular PyTorch module. To take advantage of PyTorch's convenience functions, such as the trainer API and automatic logging, we just define a few specifically named methods, which we will see in the following code:

```python
import pytorch_lightning as pl
import torch
import torch.nn as nn

from torchmetrics import Accuracy

class MultiLayerPerceptron(pl.LightningModule):
    def __init__(self, image_shape=(1, 28, 28), hidden_units=(32, 16)):
        super().__init__()
        # new PL attributes:
        self.train_acc = Accuracy()
        self.valid_acc = Accuracy()
        self.test_acc = Accuracy()

        # Model similar to previous section:
        input_size = image_shape[0] * image_shape[1] * image_shape[2]
        all_layers = [nn.Flatten()]
        for hidden_unit in hidden_units:
            layer = nn.Linear(input_size, hidden_unit)
            all_layers.append(layer)
            all_layers.append(nn.ReLU())
            input_size = hidden_unit

        all_layers.append(nn.Linear(hidden_units[-1], 10))
        self.model = nn.Sequential(*all_layers)

    def forward(self, x):
        x = self.model(x)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.train_acc.update(preds, y)
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def training_epoch_end(self, outs):
        self.log("train_acc", self.train_acc.compute())

    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.valid_acc.update(preds, y)
        self.log("valid_loss", loss, prog_bar=True)
        self.log("valid_acc", self.valid_acc.compute(), prog_bar=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.test_acc.update(preds, y)
        self.log("test_loss", loss, prog_bar=True)
        self.log("test_acc", self.test_acc.compute(), prog_bar=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer
```

Let's now discuss the different methods one by one. As you can see, the `__init__` constructor contains the same model code that we used in a previous subsection. What's new is that we added the accuracy attributes such as `self.train_acc = Accuracy()`. These will allow us to track the accuracy during training. `Accuracy` was imported from the `torchmetrics` module, which should be automatically installed with Lightning. If you cannot import `torchmetrics`, you can try to install it via `pip install torchmetrics`. More information can be found at `https://torchmetrics.readthedocs.io/en/latest/pages/quickstart.html`.

The `forward` method implements a simple forward pass that returns the logits (outputs of the last fully connected layer of our network before the softmax layer) when we call our model on the input data. The logits, computed via the `forward` method by calling `self(x)`, are used for the training, validation, and test steps, which we'll describe next.

The `training_step`, `training_epoch_end`, `validation_step`, `test_step`, and `configure_optimizers` methods are methods that are specifically recognized by Lightning. For instance, `training_step` defines a single forward pass during training, where we also keep track of the accuracy and loss so that we can analyze these later. Note that we compute the accuracy via `self.train_acc.update(preds, y)` but don't log it yet. The `training_step` method is executed on each individual batch during training, and via the `training_epoch_end` method, which is executed at the end of each training epoch, we compute the training set accuracy from the accuracy values we accumulated via training.

The `validation_step` and `test_step` methods define, analogous to the `training_step` method, how the validation and test evaluation process should be computed. Similar to `training_step`, each `validation_step` and `test_step` receives a single batch, which is why we log the accuracy via respective accuracy attributes derived from `Accuracy` of `torchmetric`. However, note that `validation_step` is only called in certain intervals, for example, after each training epoch. This is why we log the validation accuracy inside the validation step, whereas with the training accuracy, we log it after each training epoch, otherwise, the accuracy plot that we inspect later will look too noisy.

Finally, via the `configure_optimizers` method, we specify the optimizer used for training. The following two subsections will discuss how we can set up the dataset and how we can train the model.

## Setting up the data loaders for Lightning

There are three main ways in which we can prepare the dataset for Lightning. We can:

- Make the dataset part of the model
- Set up the data loaders as usual and feed them to the `fit` method of a Lightning Trainer—the Trainer is introduced in the next subsection
- Create a `LightningDataModule`

Here, we are going to use a `LightningDataModule`, which is the most organized approach. The `LightningDataModule` consists of five main methods, as we can see in the following:

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

class MnistDataModule(pl.LightningDataModule):
    def __init__(self, data_path='./'):
        super().__init__()
        self.data_path = data_path
        self.transform = transforms.Compose([transforms.ToTensor()])

    def prepare_data(self):
        MNIST(root=self.data_path, download=True)

    def setup(self, stage=None):
        # stage is either 'fit', 'validate', 'test', or 'predict'
        # here note relevant
        mnist_all = MNIST(
            root=self.data_path,
            train=True,
            transform=self.transform,
            download=False
        )

        self.train, self.val = random_split(
            mnist_all, [55000, 5000], generator=torch.Generator().manual_
seed(1)
        )
        self.test = MNIST(
            root=self.data_path,
            train=False,
            transform=self.transform,
            download=False
        )

    def train_dataloader(self):
        return DataLoader(self.train, batch_size=64, num_workers=4)

    def val_dataloader(self):
        return DataLoader(self.val, batch_size=64, num_workers=4)

    def test_dataloader(self):
        return DataLoader(self.test, batch_size=64, num_workers=4)
```

In the `prepare_data` method, we define general steps, such as downloading the dataset. In the `setup` method, we define the datasets used for training, validation, and testing. Note that MNIST does not have a dedicated validation split, which is why we use the `random_split` function to divide the 60,000-example training set into 55,000 examples for training and 5,000 examples for validation.

The data loader methods are self-explanatory and define how the respective datasets are loaded. Now, we can initialize the data module and use it for training, validation, and testing in the next subsections:

```
torch.manual_seed(1)
mnist_dm = MnistDataModule()
```

## Training the model using the PyTorch Lightning Trainer class

Now we can reap the rewards from setting up the model with the specifically named methods, as well as the Lightning data module. Lightning implements a `Trainer` class that makes the training model super convenient by taking care of all the intermediate steps, such as calling `zero_grad()`, `backward()`, and `optimizer.step()` for us. Also, as a bonus, it lets us easily specify one or more GPUs to use (if available):

```
mnistclassifier = MultiLayerPerceptron()

if torch.cuda.is_available(): # if you have GPUs
    trainer = pl.Trainer(max_epochs=10, gpus=1)
else:
    trainer = pl.Trainer(max_epochs=10)

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

Via the preceding code, we train our multilayer perceptron for 10 epochs. During training, we see a handy progress bar that keeps track of the epoch and core metrics such as the training and validation losses:

```
Epoch 9: 100% 939/939 [00:07<00:00, 130.42it/s, loss=0.1, v_num=0, train_
loss=0.260, valid_loss=0.166, valid_acc=0.949]
```

After the training has finished, we can also inspect the metrics we logged in more detail, as we will see in the next subsection.

## Evaluating the model using TensorBoard

In the previous section, we experienced the convenience of the `Trainer` class. Another nice feature of Lightning is its logging capabilities. Recall that we specified several `self.log` steps in our Lightning model earlier. After, and even during training, we can visualize them in TensorBoard. (Note that Lightning supports other loggers as well; for more information, please see the official documentation at `https://pytorch-lightning.readthedocs.io/en/latest/common/loggers.html`.)

**Installing TensorBoard**

TensorBoard can be installed via pip or conda, depending on your preference. For instance, the command for installing TensorBoard via pip is as follows:

```
pip install tensorboard
```

The following is the command for installing Lightning via conda:

```
conda install tensorboard -c conda-forge
```

The code in the following subsection is based on TensorBoard version 2.4, which you can install by replacing `tensorboard` with `tensorboard==2.4` in these commands.

By default, Lightning tracks the training in a subfolder named `lightning_logs`. To visualize the training runs, you can execute the following code in the command-line terminal, which will open TensorBoard in your browser:

```
tensorboard --logdir lightning_logs/
```

Alternatively, if you are running the code in a Jupyter notebook, you can add the following code to a Jupyter notebook cell to show the TensorBoard dashboard in the notebook directly:

```
%load_ext tensorboard
%tensorboard --logdir lightning_logs/
```

*Figure 13.9* shows the TensorBoard dashboard with the logged training and validation accuracy. Note that there is a `version_0` toggle shown in the lower-left corner. If you run the training code multiple times, Lightning will track them as separate subfolders: `version_0`, `version_1`, `version_2`, and so forth:

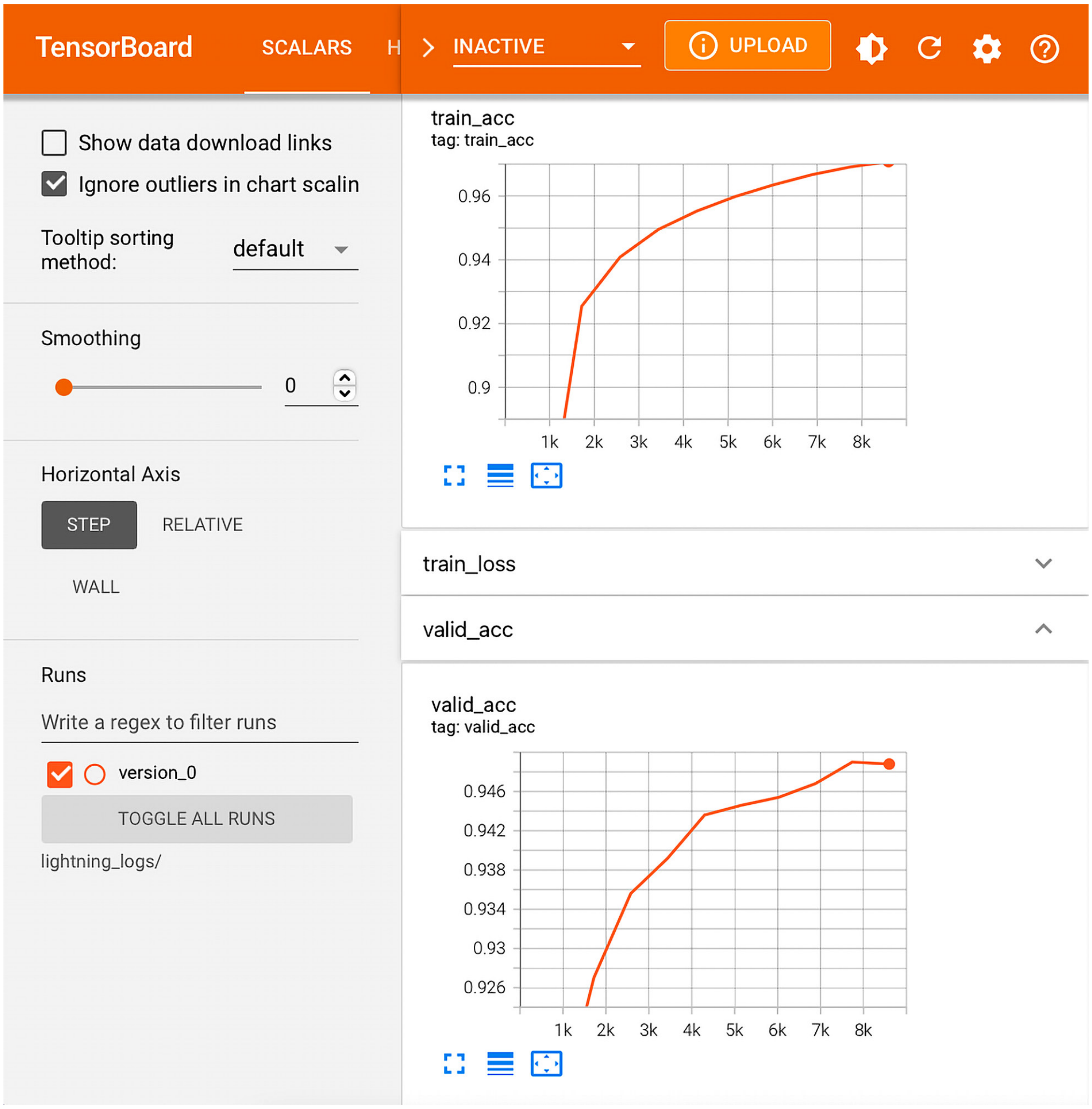
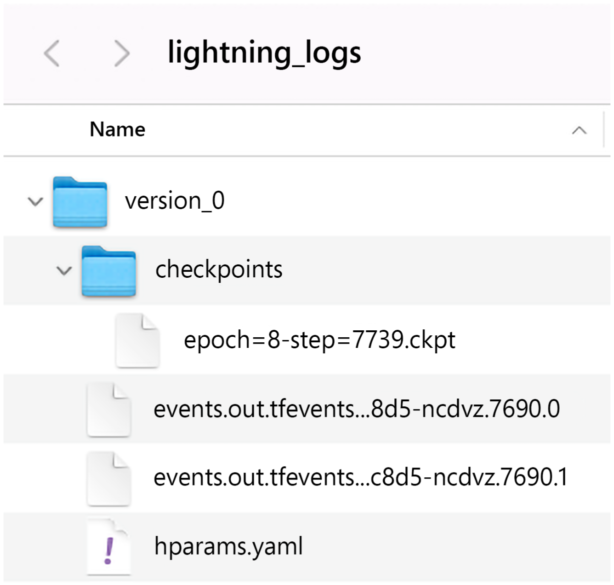


Figure 13.9: TensorBoard dashboard

By looking at the training and validation accuracies in *Figure 13.9*, we can hypothesize that training the model for a few additional epochs can improve performance.

Lightning allows us to load a trained model and train it for additional epochs conveniently. As mentioned previously, Lightning tracks the individual training runs via subfolders. In *Figure 13.10*, we see the contents of the `version_0` subfolder, which contains log files and a model checkpoint for reloading the model:



lightning_logs

Name

version_0
checkpoints
epoch=8-step=7739.ckpt
events.out.tfevents...8d5-ncdvz.7690.0
events.out.tfevents...c8d5-ncdvz.7690.1
hparams.yaml

*Figure 13.10: PyTorch Lightning log files*

For instance, we can use the following code to load the latest model checkpoint from this folder and train the model via `fit`:

```
if torch.cuda.is_available(): # if you have GPUs
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt', gpus=1)
else:
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt')

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

Here, we set `max_epochs` to 15, which trained the model for 5 additional epochs (previously, we trained it for 10 epochs).

Now, let's take a look at the TensorBoard dashboard in *Figure 13.11* and see whether training the model for a few additional epochs was worthwhile:

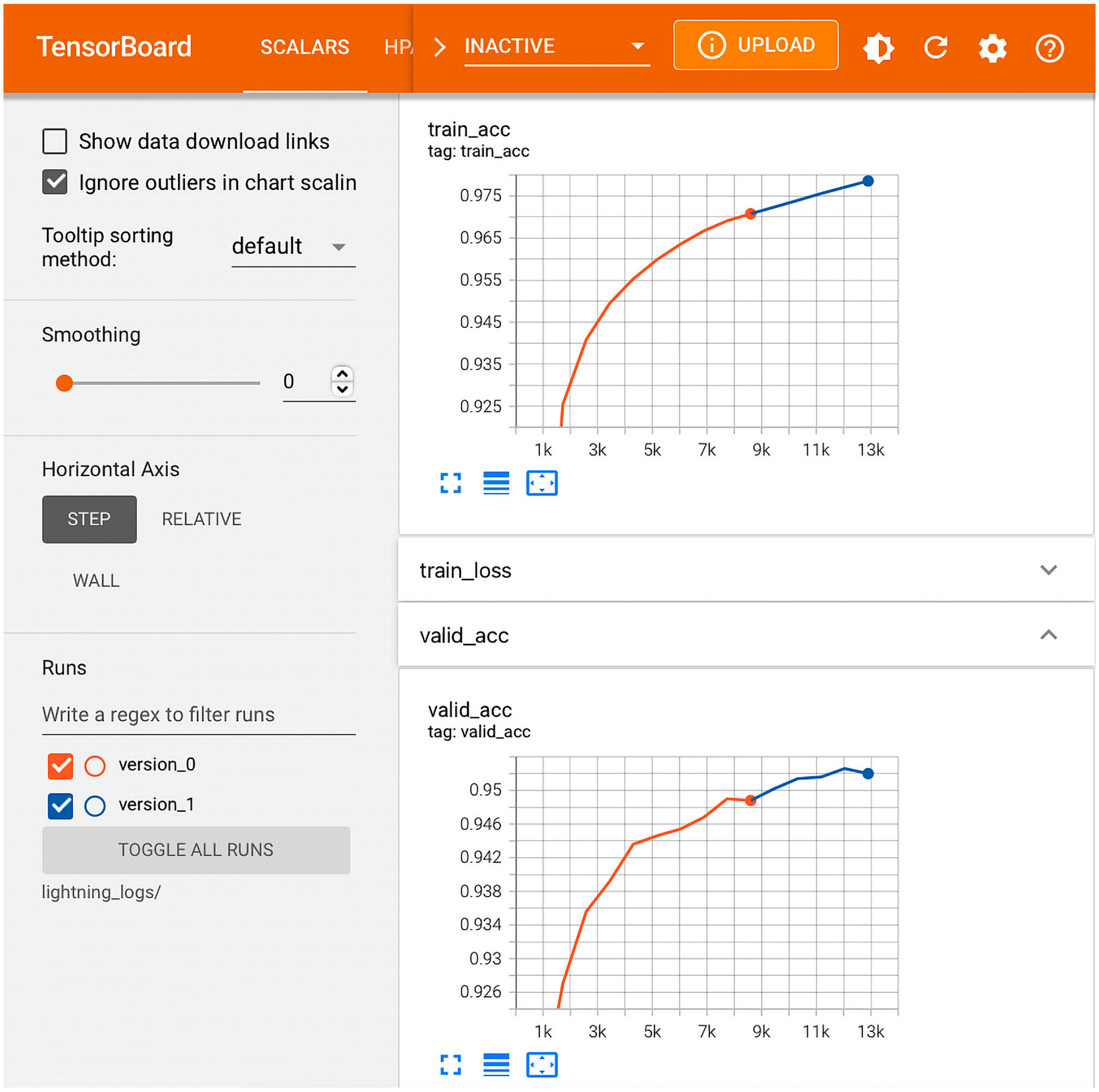


*Figure 13.11: TensorBoard dashboard after training for five more epochs*

As we can see in *Figure 13.11*, TensorBoard allows us to show the results from the additional training epochs (`version_1`) next to the previous ones (`version_0`), which is very convenient. Indeed, we can see that training for five more epochs improved the validation accuracy. At this point, we may decide to train the model for more epochs, which we leave as an exercise to you.

Once we are finished with training, we can evaluate the model on the test set using the following code:

```
trainer.test(model=mnistclassifier, datamodule=mnist_dm)
```

The resulting test set performance, after training for 15 epochs in total, is approximately 95 percent:

```
[{'test_loss': 0.14912301301956177, 'test_acc': 0.9499600529670715}]
```

Note that PyTorch Lightning also saves the model automatically for us. If you want to reuse the model later, you can conveniently load it via the following code:

```
model = MultiLayerPerceptron.load_from_checkpoint("path/to/checkpoint.ckpt")
```

**Learn more about PyTorch Lightning**

To learn more about Lightning, please visit the official website, which contains tutorials and examples, at `https://pytorch-lightning.readthedocs.io`.

Lightning also has an active community on Slack that welcomes new users and contributors. To find out more, please visit the official Lightning website at `https://www.pytorchlightning.ai`.

## Summary

In this chapter, we covered PyTorch's most essential and useful features. We started by discussing PyTorch's dynamic computation graph, which makes implementing computations very convenient. We also covered the semantics of defining PyTorch tensor objects as model parameters.

After we considered the concept of computing partial derivatives and gradients of arbitrary functions, we covered the `torch.nn` module in more detail. It provides us with a user-friendly interface for building more complex deep NN models. Finally, we concluded this chapter by solving a regression and classification problem using what we have discussed so far.

Now that we have covered the core mechanics of PyTorch, the next chapter will introduce the concept behind **convolutional neural network** (**CNN**) architectures for deep learning. CNNs are powerful models and have shown great performance in the field of computer vision.

## Join our book's Discord space