I've provided you with a lot of information, and you may be overwhelmed by the calculus concepts, but don't worry. While this calculus jargon is a means to explain PyTorch's autograd component, all you need to take away is that PyTorch takes care of the calculus for us via the `.backward` method—we won't need to compute any derivatives or gradients by hand.

A.5 Implementing multilayer neural networks

Next, we focus on PyTorch as a library for implementing deep neural networks. To provide a concrete example, let's look at a multilayer perceptron, a fully connected neural network, as illustrated in figure A.9.

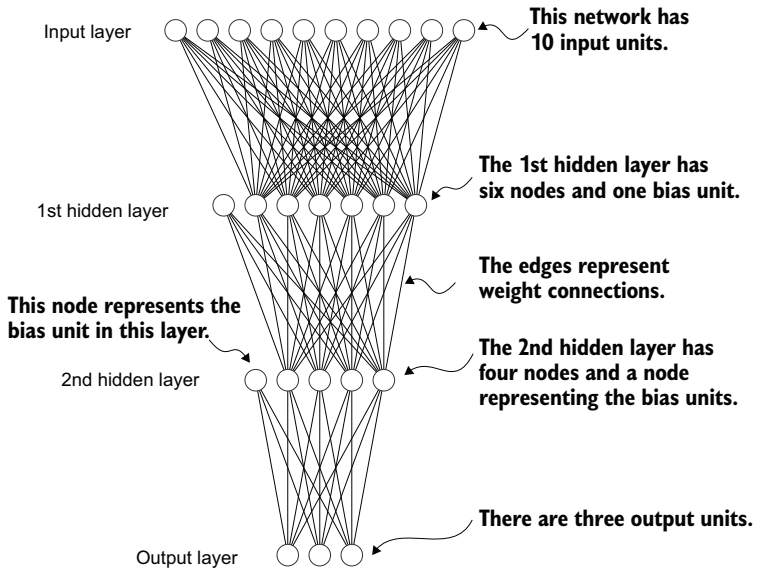

Figure A.9 A multilayer perceptron with two hidden layers. Each node represents a unit in the respective layer. For illustration purposes, each layer has a very small number of nodes.

When implementing a neural network in PyTorch, we can subclass the `torch.nn.Module` class to define our own custom network architecture. This `Module` base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.

Within this subclass, we define the network layers in the `__init__` constructor and specify how the layers interact in the forward method. The forward method describes how the input data passes through the network and comes together as a computation graph. In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function given the model parameters (see section A.7). The code in the following listing implements a

Licensed to Jason Walsh <manning@jwalsh.net>

classic multilayer perceptron with two hidden layers to illustrate a typical usage of the `Module` class.

Listing A.4 A multilayer perceptron with two hidden layers

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),

            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

Coding the number of inputs and outputs as variables allows us to reuse the same code for datasets with different numbers of features and classes

The Linear layer takes the number of input and output nodes as arguments.

Nonlinear activation functions are placed between the hidden layers.

The number of output nodes of one hidden layer has to match the number of inputs of the next layer.

The outputs of the last layer are called logits.

We can then instantiate a new neural network object as follows:

```
model = NeuralNetwork(50, 3)
```

Before using this new `model` object, we can call `print` on the model to see a summary of its structure:

```
print(model)
```

This prints

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Note that we use the `Sequential` class when we implement the `NeuralNetwork` class. `Sequential` is not required, but it can make our life easier if we have a series of layers we want to execute in a specific order, as is the case here. This way, after instantiating `self.layers = Sequential(...)` in the `__init__` constructor, we just have to

Licensed to Jason Walsh <manning@jwalsh.net>

call the `self.layers` instead of calling each layer individually in the `NeuralNetwork`'s `forward` method.

Next, let's check the total number of trainable parameters of this model:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

This prints

```
Total number of trainable model parameters: 2213
```

Each parameter for which `requires_grad=True` counts as a trainable parameter and will be updated during training (see section A.7).

In the case of our neural network model with the preceding two hidden layers, these trainable parameters are contained in the `torch.nn.Linear` layers. A `Linear` layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes referred to as a *feedforward* or *fully connected* layer.

Based on the `print(model)` call we executed here, we can see that the first `Linear` layer is at index position 0 in the layers attribute. We can access the corresponding weight parameter matrix as follows:

```
print(model.layers[0].weight)
```

This prints

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227,  ...,  0.0275, -0.0520, -0.0192],
        [-0.0169,  0.1265,  0.0255,  ..., -0.1247,  0.1191, -0.0698],
        [-0.0973, -0.0974, -0.0739,  ..., -0.0068, -0.0892,  0.1070],
        ...,
        [-0.0681,  0.1058, -0.0315,  ..., -0.1081, -0.0290, -0.1374],
        [-0.0159,  0.0587, -0.0916,  ..., -0.1153,  0.0700,  0.0770],
        [-0.1019,  0.1345, -0.0176,  ...,  0.0114, -0.0559, -0.0088]],
       requires_grad=True)
```

Since this large matrix is not shown in its entirety, let's use the `.shape` attribute to show its dimensions:

```
print(model.layers[0].weight.shape)
```

The result is

```
torch.Size([30, 50])
```

(Similarly, you could access the bias vector via `model.layers[0].bias`.)

The weight matrix here is a 30 × 50 matrix, and we can see that `requires_grad` is set to `True`, which means its entries are trainable—this is the default setting for weights and biases in `torch.nn.Linear`.

Licensed to Jason Walsh <manning@jwalsh.net>