Chapter 3. Deep Learning from Scratch

You may not realize it, but you now have all the mathematical and conceptual foundations to answer the key questions about deep learning models that I posed at the beginning of the book: you understand *how* neural networks work—the computations involved with the matrix multiplications, the loss, and the partial derivatives with respect to that loss—as well as *why* those computations work (namely, the chain rule from calculus). We achieved this understanding by building neural networks from first principles, representing them as a series of "building blocks" where each building block was a single mathematical function. In this chapter, you'll learn to represent these building blocks themselves as abstract Python classes and then use these classes to build deep learning models; by the end of this chapter, you will indeed have done "deep learning from scratch"!

We'll also map the descriptions of neural networks in terms of these building blocks to more conventional descriptions of deep learning models that you may have heard before. For example, by the end of this chapter, you'll know what it means for a deep learning model to have "multiple hidden layers." This is really the essence of understanding a concept: being able to translate between high-level descriptions and low-level details of what is actually going on. Let's begin building toward this translation. So far, we've described models just in terms of the operations that happen at a low level. In the first part of this chapter, we'll map this description of models to common higher-level concepts such as "layers" that will ultimately allow us to more easily describe more complex models.

Deep Learning Definition: A First Pass

What *is* a "deep learning" model? In the previous chapter, we defined a model as a mathematical function represented by a computational graph. The purpose of such a model was to try to map inputs, each drawn from some dataset with common characteristics (such as separate inputs representing different features of houses) to outputs drawn from a related distribution (such as the prices of those houses). We found that if we defined the model as a function that included *parameters* as inputs to some of its operations, we could "fit" it to optimally describe the data using the following procedure:

2. Calculate a *loss* representing how far off our model's predictions were from the desired outputs or *target*.

3. Using the quantities computed on the forward pass and the chain rule math worked out in Chapter 1, compute how much each of the input *parameters* ultimately affects this loss.

4. Update the values of the parameters so that the loss will hopefully be reduced when the next set of observations is passed through the model.

We started out with a model containing just a series of linear operations transforming our features into the target (which turned out to be equivalent to a traditional linear regression model). This had the expected limitation that, even when fit "optimally," the model could nevertheless represent only linear relationships between our features and our target.

We then defined a function structure that applied these linear operations first, then a *non*linear operation (the `sigmoid` function), and then a final set of linear operations. We showed that with this modification, our model *could* learn something closer to the true, nonlinear relationship between input and output, while having the additional benefit that it could learn relationships between *combinations* of our input features and the target.

What is the connection between models like these and deep learning models? We'll start with a somewhat clumsy attempt at a definition: deep learning models are represented by series of operations that have *at least two, nonconsecutive* nonlinear functions involved.

I'll show where this definition comes from shortly, but first note that since deep learning models are just a series of operations, the process of training them is in fact *identical* to the process we've been using for the simpler models we've already seen. After all, what allows this training process to work is the differentiability of the model with respect to its inputs; and as mentioned in Chapter 1, the composition of differentiable functions is differentiable, so as long as the individual operations making up the function are differentiable, the whole function will be differentiable, and we'll be able to train it using the same four-step training procedure just described.

However, so far our approach to actually training these models has been to compute these derivatives by manually coding the forward and backward passes and then multiplying the appropriate quantities together to get the derivatives. For the simple neural network model in Chapter 2, this required 17 steps. Because we're describing the model at such a low level, it isn't immediately clear how we could add more complexity to this model (or what exactly what that would mean) or even make a simple change such as swapping out a different nonlinear function for the sigmoid function. To transition to being able to build arbitrarily "deep" and otherwise "complex" deep learning models, we'll have to think about where in these 17 steps we can create reusable components, at a higher level than individual operations, that we can swap in and out to build different models. To guide us in the right direction as far as which abstractions to create, we'll try to map the operations we've been using to traditional descriptions of neural networks as being made up of "layers," "neurons," and so on.

As our first step, we'll have to create an abstraction to represent the individual operations we've been working with so far, instead of continuing to code the same matrix multiplication and bias addition over and over again.

The Building Blocks of Neural Networks: Operations

The `Operation` class will represent one of the constituent functions in our neural networks. We know that at a high level, based on the way we've used such functions in our models, it should have `forward` and `backward` methods, each of which receives an `ndarray` as an input and outputs an `ndarray`. Some another class that inherits from it—we should allow for `params` as another instance variable.

Another insight is that there seem to be two types of `Operation`s: some, such as the matrix multiplication, return an `ndarray` as output that is a different shape than the `ndarray` they received as input; by contrast, some `Operation`s, such as the `sigmoid` function, simply apply some function to each element of the input `ndarray`. What, then, is the "general rule" about the shapes of the `ndarray`s that get passed between our operations? Let's consider the `ndarray`s passed through our `Operation`s: each `Operation` will send outputs forward on the forward pass and will receive an "output gradient" on the backward pass, which will represent the partial derivative of the loss with respect to every element of the `Operation`'s output (computed by the other `Operation`s that make up the network). Also on the backward pass, each `Operation` will send an "input gradient" backward, representing the partial derivative of the loss with respect to each element of the input.

These facts place a few important restrictions on the workings of our `Operation`s that will help us ensure we're computing the gradients correctly:

- The shape of the *output gradient* `ndarray` must match the shape of the *output*.
- The shape of the *input gradient* that the `Operation` sends backward during the backward pass must match the shape of the `Operation`'s *input*.

This will all be clearer once you see it in a diagram; let's look at that next.

Diagram

This is all summarized in Figure 3-1, for an operation `O` that is receiving inputs from an operation `N` and passing outputs on to another operation `P`.

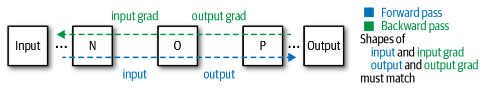

Figure 3-1. An Operation, with input and output

Figure 3-2 covers the case of an `Operation` with parameters.

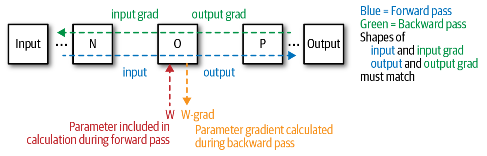

Figure 3-2. A ParamOperation, with input and output and parameters

Code

With all this, we can write the fundamental building block for our neural network, an `Operation`, as:

```
class Operation(object):
    '''
```

```
            pass

    def forward(self, input_: ndarray):
        '''
        Stores input in the self._input instance variable
        Calls the self._output() function.
        '''
        self.input_ = input_

        self.output = self._output()

        return self.output


    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Calls the self._input_grad() function.
        Checks that the appropriate shapes match.
        '''
        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        return self.input_grad


    def _output(self) -> ndarray:
        '''
        The _output method must be defined for each Operation.
        '''
        raise NotImplementedError()


    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        The _input_grad method must be defined for each Operation.
        '''
        raise NotImplementedError()
```

For any individual `Operation` that we define, we'll have to implement the `_output` and `_input_grad` functions, so named because of the quantities they compute.

NOTE

We're defining base classes like this primarily for pedagogical reasons: it is important to have the mental model that *all* `Operation`s you'll encounter throughout deep learning fit this blueprint of sending inputs forward and gradients backward, with the shapes of what they receive on the forward pass matching the shapes of what they send backward on the backward pass, and vice versa.

We'll define the specific `Operation`s we've used thus far—matrix multiplication and so on—later in this chapter. First we'll define another class that inherits from `Operation` that we'll use specifically for `Operation`s that involve parameters:

```
class ParamOperation(Operation):
    '''
    An Operation with parameters.
    '''

    def __init__(self, param: ndarray) -> ndarray:
        '''
        The ParamOperation method
    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Calls self._input_grad and self._param_grad.
        Checks appropriate shapes.
        '''

        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)
        self.param_grad = self._param_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        assert_same_shape(self.param, self.param_grad)

        return self.input_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Every subclass of ParamOperation must implement _param_grad.
        '''
        raise NotImplementedError()
```

Similar to the base `Operation`, an individual `ParamOperation` would have to define the `_param_grad` function in addition to the `_output` and `_input_grad` functions.

We have now formalized the neural network building blocks we've been using in our models so far. We could skip ahead and define neural networks directly in terms of these `Operation`s, but there is an intermediate class we've been dancing around for a chapter and a half that we'll define first: the `Layer`.

The Building Blocks of Neural Networks: Layers

In terms of `Operation`s, layers are a series of linear operations followed by a nonlinear operation. For example, our neural network from the last chapter could be said to have had five total operations: two linear operations—a weight multiplication and the addition of a bias term—followed the `sigmoid` function and then two more linear operations. In this case, we would say that the first three operations, up to and including the nonlinear one, would constitute the first layer, and the last two operations would constitute the second layer. In addition, we say that the input itself represents a special kind of layer called the *input* layer (in terms of numbering the layers, this layer doesn't count, so that we can think of it as the "zeroth" layer). The last layer, similarly, is called the *output* layer. The middle layer—the "first one," according to our numbering—also has an important name: it is called a *hidden* layer, since it is the only layer whose values we don't typically see explicitly during the course of training.

The output layer is an important exception to this definition of layers, in that it does not *have* to have a nonlinear operation applied to it; this is simply because we often want the values that come out of this layer to have values between negative infinity and infinity (or at least between 0 and infinity), whereas nonlinear functions typically "squash down" their input to some subset of that range relevant to the particular problem we're trying to solve (for example, the `sigmoid` function squashes down its input to between 0 and 1).

Diagrams

To make the connection explicit, Figure 3-3 shows the diagram of the neural network from the prior chapter with the individual operations grouped into layers.

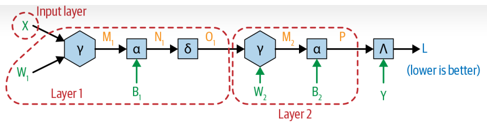

Figure 3-3. The neural network from the prior chapter with the operations grouped into layers

You can see that the input represents an "input" layer, the next three operations (ending with the `sigmoid` function) represent the next layer, and the last two operations represent the last layer.

This is, of course, rather cumbersome. And that's the point: representing neural networks as a series of individual operations, while showing clearly how neural networks work and how to train them, is too "low level" for anything more complicated than a two-layer neural network. That's why the more common way to represent neural networks is in terms of layers, as shown in Figure 3-4.

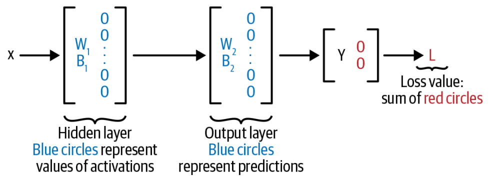
X
W1
B1
Hidden layer
Blue circles represent values of activations
W2
B2
Output layer
Blue circles represent predictions
Y
L
Loss value: sum of red circles

Figure 3-4. The neural network from the prior chapter in terms of layers

CONNECTION TO THE BRAIN

Finally, let's make one last connection between what we've seen so far and a notion you've likely heard before: each layer can be said to have a certain number of *neurons* equal to *the dimensionality of the vector that represents each observation in the layer's output*. The neural network from the prior example can thus be thought of as having 13 neurons in the input layer, then 13 neurons (again) in the hidden layer, and one neuron in the output layer.

Neurons in the brain have the property that they can receive inputs from many other neurons and will "fire" and send a signal forward only if the signals they receive cumulatively reach a certain "activation energy." Neurons in the context of neural networks have a loosely analogous property: they do indeed send signals forward based on their inputs, but the inputs are transformed into outputs simply via a nonlinear function. Thus, this nonlinear function is called the *activation function*, and the values that come out of it are called the *activations* for that layer.[1]

Now that we've defined layers, we can state the more conventional definition of deep learning: *deep learning models are neural networks with more than one hidden layer.*

We can see that this is equivalent to the earlier definition that was purely in terms of `Operation`s, since a layer is just a series of `Operation`s with a nonlinear operation at the end.

Now that we've defined a base class for our `Operation`s, let's show how it can serve as the fundamental building block of the models we saw in the prior chapter.

Building Blocks on Building Blocks

step by step, we know there are three kinds:

- The matrix multiplication of the input with the matrix of parameters
- The addition of a bias term
- The `sigmoid` activation function

Let's start with the `WeightMultiply Operation`:

```
class WeightMultiply(ParamOperation):
    '''
    Weight multiplication operation for a neural network.
    '''

    def __init__(self, W: ndarray):
        '''
        Initialize Operation with self.param = W.
        '''
        super().__init__(W)

    def _output(self) -> ndarray:
        '''
        Compute output.
        '''
        return np.dot(self.input_, self.param)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Compute input gradient.
        '''
        return np.dot(output_grad, np.transpose(self.param, (1, 0)))

    def _param_grad(self, output_grad: ndarray)  -> ndarray:
        '''
        Compute parameter gradient.
        '''
        return np.dot(np.transpose(self.input_, (1, 0)), output_grad)
```

Here we simply code up the matrix multiplication on the forward pass, as well as the rules for "sending gradients backward" to both the inputs and the parameters on the backward pass (using the rules for doing so that we reasoned through at the end of Chapter 1). As you'll see shortly, we can now use this as a *building block* that we can simply plug into our `Layer`s.

Next up is the addition operation, which we'll call `BiasAdd`:

```
class BiasAdd(ParamOperation):
    '''
    Compute bias addition.
    '''

    def __init__(self,
                 B: ndarray):
        '''
        Initialize Operation with self.param = B.
        Check appropriate shape.
        '''
        assert B.shape[0] == 1

        super().__init__(B)

    def _output(self) -> ndarray:
        '''
        Compute output.
        '''
        return self.input_ + self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Compute input gradient.
        '''
        '''
        Compute parameter gradient.
        '''
        param_grad = np.ones_like(self.param) * output_grad
        return np.sum(param_grad, axis=0).reshape(1, param_grad.shape
```

Finally, let's do `sigmoid`:

```
class Sigmoid(Operation):
    '''
    Sigmoid activation function.
    '''

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

    def _output(self) -> ndarray:
        '''
        Compute output.
        '''
        return 1.0/(1.0+np.exp(-1.0 * self.input_))

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Compute input gradient.
        '''
        sigmoid_backward = self.output * (1.0 - self.output)
        input_grad = sigmoid_backward * output_grad
        return input_grad
```

This simply implements the math described in the previous chapter.

NOTE

For both `sigmoid` and the `ParamOperation`, the step during the backward pass where we compute:

```
input_grad = <something> * output_grad
```

is the step where we are applying the chain rule, and the corresponding rule for `WeightMultiply`:

```
np.dot(output_grad, np.transpose(self.param, (1, 0)))
```

is, as I argued in Chapter 1, the analogue of the chain rule when the function in question is a matrix multiplication.

Now that we've defined these `Operation`s precisely, we can use *them* as building blocks to define a `Layer`.

The Layer Blueprint

Because of the way we've written the `Operation`s, writing the `Layer` class is easy:

- The `forward` and `backward` methods simply involve sending the input successively forward through a series of `Operation`s—exactly as we've

    - Defining the correct series of `Operation`s in the `_setup_layer` function and initializing and storing the parameters in these `Operation`s (which will also take place in the `_setup_layer` function)

    - Storing the correct values in `self.input_` and `self.output` on the `forward` method

    - Performing the correct assertion checking in the `backward` method

- Finally, the `_params` and `_param_grads` functions simply extract the parameters and their gradients (with respect to the loss) from the `ParamOperation`s within the layer.

Here's what all that looks like:

```
class Layer(object):
    '''
    A "layer" of neurons in a neural network.
    '''

    def __init__(self,
                 neurons: int):
        '''
        The number of "neurons" roughly corresponds to the "breadth" of
        layer
        '''
        self.neurons = neurons
        self.first = True
        self.params: List[ndarray] = []
        self.param_grads: List[ndarray] = []
        self.operations: List[Operation] = []

    def _setup_layer(self, num_in: int) -> None:
        '''
        The _setup_layer function must be implemented for each layer.
        '''
        raise NotImplementedError()

    def forward(self, input_: ndarray) -> ndarray:
        '''
        Passes input forward through a series of operations.
        '''
        if self.first:
            self._setup_layer(input_)
            self.first = False

        self.input_ = input_

        for operation in self.operations:

            input_ = operation.forward(input_)

        self.output = input_

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Passes output_grad backward through a series of operations.
        Checks appropriate shapes.
        '''

        assert_same_shape(self.output, output_grad)

        for operation in reversed(self.operations):
            output_grad = operation.backward(output_grad)

        input_grad = output_grad

        self._param_grads()

        return input_grad

    def _param_grads(self) -> ndarray:
        self.param_grads = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.param_grads.append(operation.param_grad)

    def _params(self) -> ndarray:
        '''
        Extracts the _params from a layer's operations.
        '''

        self.params = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.params.append(operation.param)
```

Just as we moved from an abstract definition of an `Operation` to the implementation of specific `Operation`s needed for the neural network from Chapter 2, let's now implement the `Layer` from that network as well.

The Dense Layer

We called the `Operation`s we've been dealing with `WeightMultiply`, `BiasAdd`, and so on. What should we call the layer we've been using so far? A `LinearNonLinear` layer?

A defining characteristic of this layer is that *each output neuron is a function of all of the input neurons*. That is what the matrix multiplication is really doing: if the matrix is n_{in} rows by n_{out} columns, the multiplication itself is computing n_{out} new features, each of which is a weighted linear combination of *all* of the n_{in} input features.[2] Thus these layers are often called *fully connected* layers; recently, in the popular `Keras` library, they are also often called `Dense` layers, a more concise term that gets across the same idea.

Now that we know what to call it and why, let's define the `Dense` layer in terms of the operations we've already defined—as you'll see, because of how we defined our `Layer` base class, all we need to do is to put the `Operation`s defined in the previous section in as a list in the `_setup_layer` function.

```
class Dense(Layer):
    '''
    A fully connected layer that inherits from "Layer."
    '''
    def __init__(self,
                 neurons: int,
                 activation: Operation = Sigmoid()) -> None:
        '''
        Requires an activation function upon initialization.
        '''
        super().__init__(neurons)
        self.activation = activation

    def _setup_layer(self, input_: ndarray) -> None:
        '''
        Defines the operations of a fully connected layer.
        '''
        if self.seed:
            np.random.seed(self.seed)

        self.params = []

        # weights
        self.params.append(np.random.randn(input_.shape[1], self.neurc

        # bias
        self.params.append(np.random.randn(1, self.neurons))

        self.operations = [WeightMultiply(self.params[0]),
                           BiasAdd(self.params[1]),
                           self.activation]
```

Note that we'll make the default activation a `Linear` activation, which really means we apply no activation, and simply apply the identity function to the output of the layer.

What building blocks should we now add on top of `Operation` and `Layer`? To train our model, we know we'll need a `NeuralNetwork` class to wrap around `Layer`s, just as `Layer`s wrapped around `Operation`s. It isn't obvious what other classes will be needed, so we'll just dive in and build `NeuralNetwork` and figure out the other classes we'll need as we go.

The NeuralNetwork Class, and Maybe Others

What should our `NeuralNetwork` class be able to do? At a high level, it should be able to *learn from data*: more precisely, it should be able to take in batches of data representing "observations" (`X`) and "correct answers" (`y`) and learn the relationship between `X` and `y`, which means learning a function that can transform `X` into predictions `p` that are very close to `y`.

How exactly will this learning take place, given the `Layer` and `Operation` classes just defined? Recalling how the model from the last chapter worked, we'll implement the following:

1. The neural network should take `X` and pass it successively forward through each `Layer` (which is really a convenient wrapper around feeding it through many `Operation`s), at which point the result will represent the `prediction`.

2. Next, `prediction` should be compared with the value `y` to calculate the loss and generate the "loss gradient," which is the partial derivative of the loss with respect to each element in the last layer in the network (namely, the one that generated the `prediction`).

3. Finally, we'll send this loss gradient successively backward through each layer, along the way computing the "parameter gradients"—the partial derivative of the loss with respect to each of the parameters—and storing them in the corresponding `Operation`s.

Diagram

Figure 3-5 captures this description of a neural network in terms of `Layer`s.

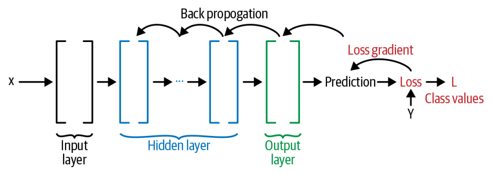

Figure 3-5. Backpropagation, now in terms of Layers instead of Operations

Code

How should we implement this? First, we'll want our neural network to ultimately deal with `Layer`s the same way our `Layer`s dealt with `Operation`s. For example, we want the `forward` method to receive `X` as input and simply do something like:

```
for layer in self.layers:
```

Similarly, we'll want our `backward` method to take in an argument—let's initially call it `grad`—and do something like:

```
for layer in reversed(self.layers):
    grad = layer.backward(grad)
```

Where will `grad` come from? It has to come from the *loss*, a special function that takes in the `prediction` along with `y` and:

- Computes a single number representing the "penalty" for the network making that `prediction`.
- Sends backward a gradient for every element of the `prediction` with respect to the loss. This gradient is what the last `Layer` in the network will receive as the input to its `backward` function.

In the example from the prior chapter, the loss function was the squared difference between the `prediction` and the target, and the gradient of the `prediction` with respect to the loss was computed accordingly.

How should we implement this? It seems like this concept is important enough to deserve its own class. Furthermore, this class can be implemented similarly to the `Layer` class, except the `forward` method will produce an actual number (a `float`) as the loss, instead of an `ndarray` to be sent forward to the next `Layer`. Let's formalize this.

Loss Class

The `Loss` base class will be similar to `Layer`—the `forward` and `backward` methods will check that the shapes of the appropriate `ndarray`s are identical and define two methods, `_output` and `_input_grad`, that any subclass of `Loss` will have to define:

```
class Loss(object):
    '''
    The "loss" of a neural network.
    '''

    def __init__(self):
        '''Pass'''
        pass

    def forward(self, prediction: ndarray, target: ndarray) -> float:
        '''
        Computes the actual loss value.
        '''
        assert_same_shape(prediction, target)

        self.prediction = prediction
        self.target = target

        loss_value = self._output()

        return loss_value

    def backward(self) -> ndarray:
        '''
        Computes gradient of the loss value with respect to the input t
        loss function.
        '''
        self.input_grad = self._input_grad()

        assert_same_shape(self.prediction, self.input_grad)

        return self.input_grad

    def _output(self) -> float:
        '''
    def _input_grad(self) -> ndarray:
        '''
        Every subclass of "Loss" must implement the _input_grad functio
        '''
        raise NotImplementedError()
```

As in the `Operation` class, we check that the gradient that the loss sends backward is the same shape as the `prediction` received as input from the last layer of the network:

```
class MeanSquaredError(Loss):

    def __init__(self)
        '''Pass'''
        super().__init__()

    def _output(self) -> float:
        '''
        Computes the per-observation squared error loss.
        '''
        loss =
            np.sum(np.power(self.prediction - self.target, 2)) /
            self.prediction.shape[0]

        return loss

    def _input_grad(self) -> ndarray:
        '''
        Computes the loss gradient with respect to the input for MSE lo
        '''

        return 2.0 * (self.prediction - self.target) / self.predictior
```

Here, we simply code the forward and backward rules of the mean squared error loss formula.

This is the last key building block we need to build deep learning from scratch. Let's review how these pieces fit together and then proceed with building a model!

Deep Learning from Scratch

We ultimately want to build a `NeuralNetwork` class, using Figure 3-5 as a guide, that we can use to define and train deep learning models. Before we dive in and start coding, let's describe precisely what such a class would be and how it would interact with the `Operation`, `Layer`, and `Loss` classes we just defined:

1. A `NeuralNetwork` will have a list of `Layer`s as an attribute. The `Layer`s would be as defined previously, with `forward` and `backward` methods. These methods take in `ndarray` objects and return `ndarray` objects.

2. Each `Layer` will have a list of `Operation`s saved in the `operations` attribute of the layer during the `_setup_layer` function.

3. These `Operation`s, just like the `Layer` itself, have `forward` and `backward` methods that take in `ndarray` objects as arguments and return `ndarray` objects as outputs.

4. In each operation, the shape of the `output_grad` received in the `backward` method must be the same as the shape of the `output` attribute of the `Layer`. The same is true for the shapes of the `input_grad` passed backward during the `backward` method and the `input_` attribute.

5. Some operations have parameters (stored in the `param` attribute); these operations inherit from the `ParamOperation` class. The same constraints

ndarray objects, and the shapes of the `input` and `output` attributes and their corresponding gradients must match.

6. A `NeuralNetwork` will also have a `Loss`. This class will take the output of the last operation from the `NeuralNetwork` and the target, check that their shapes are the same, and calculate both a loss value (a number) and an `ndarray` `loss_grad` that will be fed into the output layer, starting backpropagation.

Implementing Batch Training

We've covered several times the high-level steps for training a model one batch at a time. They are important and worth repeating:

1. Feed input through the model function (the "forward pass") to get a prediction.
2. Calculate the number representing the loss.
3. Calculate the gradient of the loss with respect to the parameters, using the chain rule and the quantities computed during the forward pass.
4. Update the parameters using these gradients.

We would then feed a new batch of data through and repeat these steps.

Translating these steps into the `NeuralNetwork` framework just described is straightforward:

1. Receive `X` and `y` as inputs, both `ndarray`s.
2. Feed `X` successively forward through each `Layer`.
3. Use the `Loss` to produce loss value and the loss gradient to be sent backward.
4. Use the loss gradient as input to the `backward` method for the network, which will calculate the `param_grads` for each layer in the network.
5. Call the `update_params` function on each layer, which will use the overall learning rate for the `NeuralNetwork` as well as the newly calculated `param_grads`.

We finally have our full definition of a neural network that can accommodate batch training. Now let's code it up.

NeuralNetwork: Code

Coding all of this up is pretty straightforward:

```
class NeuralNetwork(object):
    '''
    The class for a neural network.
    '''
    def __init__(self, layers: List[Layer],
                 loss: Loss,
                 seed: float = 1)
        '''
        Neural networks need layers, and a loss.
        '''
        self.layers = layers
        self.loss = loss
        self.seed = seed
        if seed:
            for layer in self.layers:
```

```
        Passes data forward through a series of layers.
        '''
        x_out = x_batch
        for layer in self.layers:
            x_out = layer.forward(x_out)

        return x_out

    def backward(self, loss_grad: ndarray) -> None:
        '''
        Passes data backward through a series of layers.
        '''

        grad = loss_grad
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

        return None

    def train_batch(self,
                    x_batch: ndarray,
                    y_batch: ndarray) -> float:
        '''
        Passes data forward through the layers.
        Computes the loss.
        Passes data backward through the layers.
        '''

        predictions = self.forward(x_batch)

        loss = self.loss.forward(predictions, y_batch)

        self.backward(self.loss.backward())

        return loss

    def params(self):
        '''
        Gets the parameters for the network.
        '''
        for layer in self.layers:
            yield from layer.params

    def param_grads(self):
        '''
        Gets the gradient of the loss with respect to the parameters fo
        network.
        '''
        for layer in self.layers:
            yield from layer.param_grads
```

With this `NeuralNetwork` class, we can implement the models from the prior chapter in a more modular, flexible way and define other models to represent complex nonlinear relationships between input and output. For example, here's how to easily instantiate the two models we covered in the last chapter—the linear regression and the neural network:[3]

```
linear_regression = NeuralNetwork(
    layers=[Dense(neurons = 1)],
            loss = MeanSquaredError(),
            learning_rate = 0.01
            )

neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=Linear())],
            loss = MeanSquaredError(),
            learning_rate = 0.01
            )
```

We're basically done; now we just feed data repeatedly through the network in order for it to learn. To make this process cleaner and easier to extend to the more

NeuralNetwork parameters given the gradients computed on the backward pass. Let's quickly define these two classes.

Trainer and Optimizer

First, let's note the similarities between these classes and the code we used to train the network in Chapter 2. There, we used the following code to implement the four steps described earlier for training the model:

```
# pass X_batch forward and compute the loss
forward_info, loss = forward_loss(X_batch, y_batch, weights)

# compute the gradient of the loss with respect to each of the weights
loss_grads = loss_gradients(forward_info, weights)

# update the weights
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

This code was within a `for` loop that repeatedly fed data through the function defining and updated our network.

With the classes we have now, we'll ultimately do this inside a `fit` function within the `Trainer` class that will mostly be a wrapper around the `train` function used in the prior chapter. (The full code for it is in this chapter's Jupyter Notebook on the book's GitHub page.) The main difference is that inside this new function, the first two lines from the preceding code block will be replaced with this line:

```
neural_network.train_batch(X_batch, y_batch)
```

Updating the parameters, which happens in the following two lines, will take place in a separate `Optimizer` class. And finally, the `for` loop that previously wrapped around all of this will take place in the `Trainer` class that wraps around the `NeuralNetwork` and the `Optimizer`.

Next, let's discuss why we need an `Optimizer` class and what it should look like.

Optimizer

In the model we described in the last chapter, each `Layer` contains a simple rule for updating the weights based on the parameters and their gradients. As we'll touch on in the next chapter, there are many other update rules we can use, such as ones involving the *history* of gradient updates rather than just the gradient updates from the specific batch that was fed in at that iteration. Creating a separate `Optimizer` class will give us the flexibility to swap in one update rule for another, something that we'll explore in more detail in the next chapter.

DESCRIPTION AND CODE

The base `Optimizer` class will take in a `NeuralNetwork` and, every time the `step` function is called, will update the parameters of the network based on their current values, their gradients, and any other information stored in the `Optimizer`:

```
class Optimizer(object):
    '''
    Base class for a neural network optimizer.
    '''
    def __init__(self,
                 lr: float = 0.01):
        '''
    def step(self) -> None:
        '''
        Every optimizer must implement the "step" function.
        '''
        pass
```

And here’s how this looks with the straightforward update rule we’ve seen so far, known as *stochastic gradient descent*:

```
class SGD(Optimizer):
    '''
    Stochastic gradient descent optimizer.
    '''
    def __init__(self,
                 lr: float = 0.01) -> None:
        '''Pass'''
        super().__init__(lr)

    def step(self):
        '''
        For each parameter, adjust in the appropriate direction, with t
        magnitude of the adjustment based on the learning rate.
        '''
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):

            param -= self.lr * param_grad
```

NOTE

Note that while our `NeuralNetwork` class does not have an `_update_params` method, we do rely on the `params()` and `param_grads()` methods to extract the correct `ndarray`s for optimization.

That’s the basic `Optimizer` class; let’s cover the `Trainer` class next.

Trainer

In addition to training the model as described previously, the `Trainer` class also links together the `NeuralNetwork` with the `Optimizer`, ensuring the latter trains the former properly. You may have noticed in the previous section that we didn’t pass in a `NeuralNetwork` when initializing our `Optimizer`; instead, we’ll assign the `NeuralNetwork` to be an attribute of the `Optimizer` when we initialize the `Trainer` class shortly, with this line:

```
setattr(self.optim, 'net', self.net)
```

In the following subsection, I show a simplified but working version of the `Trainer` class that for now contains just the `fit` method. This method trains our model for a number of *epochs* and prints out the loss value after each set number of epochs. In each epoch, we:

1. Shuffle the data at the beginning of the epoch
2. Feed the data through the network in batches, updating the parameters after each batch has been fed through

TRAINER CODE

In the following is the code for a simple version of the `Trainer` class, we hide two self-explanatory helper methods used during the `fit` function: `generate_batches`, which generates batches of data from `X_train` and `y_train` for training, and `permute_data`, which shuffles `X_train` and `y_train` at the beginning of each epoch. We also include a `restart` argument in the `train` function: if `True` (default), it will reinitialize the model's parameters to random values upon calling the `train` function:

```
class Trainer(object):
    '''
    Trains a neural network.
    '''
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer)
        '''
        Requires a neural network and an optimizer in order for trainin
        occur. Assign the neural network as an instance variable to the
        '''
        self.net = net
        setattr(self.optim, 'net', self.net)

    def fit(self, X_train: ndarray, y_train: ndarray,
            X_test: ndarray, y_test: ndarray,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32,
            seed: int = 1,
            restart: bool = True) -> None:
        '''
        Fits the neural network on the training data for a certain numb
        epochs. Every "eval_every" epochs, it evaluates the neural netw
        the testing data.
        '''
        np.random.seed(seed)

        if restart:
            for layer in self.net.layers:
                layer.first = True

        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.net.train_batch(X_batch, y_batch)

                self.optim.step()

            if (e+1) % eval_every == 0:

                test_preds = self.net.forward(X_test)

                loss = self.net.loss.forward(test_preds, y_test)

                print(f"Validation loss after {e+1} epochs is {loss:.
```

In the full version of this function in the book's GitHub repository, we also implement *early stopping*, which does the following:

1. It saves the loss value every `eval_every` epochs.
2. It checks whether the validation loss is lower than the last time it was calculated.
3. If the validation loss is *not* lower, it uses the model from `eval_every`

Putting Everything Together

Here is the full code to train our network using all the `Trainer` and `Optimizer` classes and the two models defined before—`linear_regression` and `neural_network`. We'll set the learning rate to `0.01` and the maximum number of epochs to `50` and evaluate our models every `10` epochs:

```
optimizer = SGD(lr=0.01)
trainer = Trainer(linear_regression, optimizer)

trainer.fit(X_train, y_train, X_test, y_test,
       epochs = 50,
       eval_every = 10,
       seed=20190501);
```

```
Validation loss after 10 epochs is 30.295
Validation loss after 20 epochs is 28.462
Validation loss after 30 epochs is 26.299
Validation loss after 40 epochs is 25.548
Validation loss after 50 epochs is 25.092
```

Using the same model-scoring functions from Chapter 2, and wrapping them inside an `eval_regression_model` function, gives us these results:

```
eval_regression_model(linear_regression, X_test, y_test)
```

```
Mean absolute error: 3.52

Root mean squared error 5.01
```

These are similar to the results of the linear regression we ran in the last chapter, confirming that our framework is working.

Running the same code with the `neural_network` model with a hidden layer with 13 neurons, we get the following:

```
Validation loss after 10 epochs is 27.434
Validation loss after 20 epochs is 21.834
Validation loss after 30 epochs is 18.915
Validation loss after 40 epochs is 17.193
Validation loss after 50 epochs is 16.214
```

```
eval_regression_model(neural_network, X_test, y_test)
```

```
Mean absolute error: 2.60

Root mean squared error 4.03
```

Again, these results are similar to what we saw in the prior chapter, and they're significantly better than our straightforward linear regression.

Our First Deep Learning Model (from Scratch)

Now that all of that setup is out of the way, defining our first deep learning model is trivial:

```
deep_neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=LinearAct())],
    loss=MeanSquaredError(),
```

We won't even try to be clever with this (yet). We'll just add a hidden layer with the same dimensionality as the first layer, so that our network now has two hidden layers, each with 13 neurons.

Training this using the same learning rate and evaluation schedule as the prior models yields the following result:

```
Validation loss after 10 epochs is 44.134
Validation loss after 20 epochs is 25.271
Validation loss after 30 epochs is 22.341
Validation loss after 40 epochs is 16.464
Validation loss after 50 epochs is 14.604
```

```
eval_regression_model(deep_neural_network, X_test, y_test)
```

```
Mean absolute error: 2.45

Root mean squared error 3.82
```

We finally worked up to doing deep learning from scratch—and indeed, on this real-world problem, without the use of any tricks (just a bit of learning rate tuning), our deep learning model does perform slightly better than a neural network with just one hidden layer.

More importantly, we did so by building a framework that is easily extensible. We could easily implement other kinds of `Operation`s, wrap them in new `Layer`s, and drop them right in, assuming that they have defined `_output` and `_input_grad` methods and that the dimensions of their inputs, outputs, and parameters match those of their respective gradients. Similarly, we could easily drop different activation functions into our existing layers and see if it decreases our error metrics; I encourage you to clone the book's GitHub repo and try this!

Conclusion and Next Steps

In the next chapter, I'll cover several tricks that will be essential to getting our models to train properly once we get to more challenging problems than this simple one[4] —in particular, defining other `Loss`es and `Optimizer`s. I'll also cover additional tricks for tuning our learning rates and modifying them throughout training, and I'll show how to incorporate this into the `Optimizer` and `Trainer` classes. Finally, we'll see Dropout, a new kind of `Operation` that has proven essential for increasing the training stability of deep learning models. Onward!

1 Among all activation functions, the `sigmoid` function, which maps inputs to between 0 and 1, most closely mimics the actual activation of neurons in the brain, but in general activation functions can be any monotonic, nonlinear function.

2 As we'll see in Chapter 5, this is not true of all layers: in *convolutional* layers, for example, each output feature is a combination of *only a small subset* of the input features.

3 The learning rate of 0.01 isn't special; we simply found it to be optimal in the course of experimenting while writing the prior chapter.

4 Even on this simple problem, changing the hyperparameters slightly can cause the deep learning model to fail to beat the two-layer neural network. Clone the GitHub repo and try it yourself!