NEURAL NETWORK PACKAGE MODULE 5

BUILDING FROM SCRATCH

QMIND EDUCATE

ANDREW FARLEY

TABLE OF CONTENTS

PREREQUISITES

Before we begin, I'd like to outline what I'm assuming you know before you read this:

1. You understand python code and are able to write it:
    a. You have an environment setup on your computer to program python in.
        i. Good guide for the basics here.
        ii. I would recommend an environment called Spyder.
    b. You have knowledge of how the NumPy package works. This can be tricky, and we will be using many aspects of it, so it is imperative you know how to use it. Follow this guide if you have a limited understanding of NumPy.

2. You have read (and at least mostly understood) modules one to three in this package. If you still feel you don't have a good grasp on the material, try re-reading and/or using the resources referenced to understand it better. If it is a certain topic you are having difficulty with, look it up online. There are plenty of resources out there that will give you a fresh perspective which might make tricky material click for you.

3. Have passion for neural networks. This may sound corny, but you are going to need it to complete this properly. It is not an easy task to build a neural network from scratch; there are going to be many challenges that you must persevere through. I encourage you to try everything yourself before checking the answers. Although this may take more time, it is the best way to learn!

Please ensure you have completed everything in the checklist above. With that all being said, let's get into it.

THE SETUP

Where do we even begin when trying to make our own neural network?

The first thing I want you to do is open a blank python file. Name it whatever you like, I would suggest something along the lines of ‘NeuralNetwork.py’. Now let’s build up a checklist of what needs to be done to create a neural net. Looking back at the previous modules we know it needs:

1. A specific structure, outlining how many inputs there will be, how many hidden layers, how many neurons on each hidden layer, and the number of outputs.
2. Layers of neurons which will be able to take an input and produce an output using the forward propagation algorithm.
3. A feed forward function which will take some inputs, compute the entire feed forward algorithm through every layer, and produce some outputs.
4. A train function which will take some inputs, what their expected output should be, and then perform back propagation on the network to step the weights to reduce error.

This list consists of the very basics we need to get a working neural network.

The very first line of code you should write is your import statement. We are using numpy as our only dependency, so you must import the numpy library. To do this, make line 1 of your program:

```
import numpy as np
```

I am importing numpy as np because it makes our code much more concise, this is common practice in Python.

THE NEURAL NETWORK CLASS: SETUP

Both the layer and the neural network have different properties and variables associated with them. Let's define some classes for each. A class is just a defined type of variable that will contain different variables and functions inside of it. It allows for reusable, modular code. Let's begin with the definition of a Neural Network class. We know that every Neural Network needs a structure which includes:

- The number of inputs
- The number of hidden layers
- The number of neurons on each hidden layer
- The number of outputs

In the constructor of our class let's ensure we have variables for these values.

```python
class NeuralNetwork:
    def __init__(self, inputNum, hiddenNeurons, outputNum):
        self.inputNum = inputNum
        self.hiddenNum = len(hiddenNeurons)
        self.outputNum = outputNum
```

As you can see, we only have three, true inputs to the constructor (the method called when the class is created). The inputNum and outputNum parameters are integers that will define exactly what they are called. The hiddenNeurons parameter is a list of the number of neurons at every hidden layer. This kills two birds with one stone, as we can just get the length of this list to know how many hidden layers there are!

This is how everything will be built: we are going to break it down to the principles, ideas, and values we need, and then build our code around that.

We now have a basic Neural Network object, but how will each layer be stored? Let's make another class.

THE LAYER CLASS

First, we begin with the definition of the layer. We know we need two inputs in the constructor, how many inputs the layer receives and how many neurons it has.

```
class Layer:
    def __init__(self, inputNum, neurons):
```

This is the definition for our Layer class. Let's now recall that every layer has two matrices associated with it, the weights matrix and the bias matrix. The weights matrix will be of size *inputNum x neurons* and the bias matrix will be *1 x neurons*. We initialize them with random values:

```
class Layer:
    def __init__(self, inputNum, neurons):
        self.weights = 2 * np.random.rand(inputNum, neurons) - 1
        self.bias = 2 * np.random.rand(1, neurons) - 1
```

Notice how we are manipulating the random matrices with multiplication and subtraction. This is because the np.random.rand function gives you a matrix with random values in the range 0 to 1 (with the shape specified by the inputs). We want our weights and biases to be in the range -1 to 1. We can manipulate the space accordingly.

$[0,1) * 2 = [0,2)$ Multiply by 2 because our new range has a 'length' of 2 (1 – (-1) = 2).

$[0,2) - 1 = [-1,1)$ Our desired range can be obtained by shifting all values down by 1!

Great, we now have the definition and constructor of our Layer class, but what functions need to be associated with it? We know that every layer needs to be able to feed forward an input matrix to produce and output matrix. The process for this, as we should know, is quite simple. We simply multiply the input by the weights, add the bias, and apply the activation function to every output.

```
def feed(self, inputData):
    #Multiply input by weights
    out = inputData.dot(self.weights)

    #Add the bias
    out = out = self.bias

    #Apply the activation function
    out = 1 / (1 + np.exp(-out)) #Sigmoid activation function

    return out
```

Perfect! That is going to be our Layer class, you will see why it isn't more complex when we do our training. There are many ways to create a neural network library so don't be discouraged if you were thinking it would be different. Something that may be ringing a few alarm bells in your head is that it doesn't seem that this layer can be trained. We will handle training through our NeuralNetwork class however you could integrate that as a feature/function of the Layer object.

Your Layer class should now look something along the lines of:

```
class Layer:
    def __init__(self, inputNum, neurons):
        self.weights = 2 * np.random.rand(inputNum, neurons) - 1
        self.bias = 2 * np.random.rand(1, neurons) - 1

    def feed(self, inputData):
        #Multiply input by weights
        out = inputData.dot(self.weights)

        #Add the bias
        out = out + self.bias

        #Apply the activation function
        out = 1 / (1 + np.exp(-out)) #Sigmoid activation function

        return out
```

THE NEURAL NETWORK CLASS: FORWARD PROPAGATION

The next logical step in adding to this library is enabling forward propagation in our NeuralNetwork class. Before we create a function that will do this, we need to set ourselves up a bit. Our NeuralNetwork class has no layers in it yet, just the information for how they should be structured. Let's setup our layers in the constructor. We know we will need access to every layer, so it would be wise to store them in a list.

```
class NeuralNetwork:
    def __init__(self, inputNum, hiddenNeurons, outputNum):
        #Storing the structure
        self.inputNum = inputNum
        self.hiddenNum = len(hiddenNeurons)
        self.outputNum = outputNum

        #Creating and storing the layers

        #First, get the possible input and output sizes (needed for layer
construction)
        possibleInputSizes = [inputNum] + hiddenNeurons
        possibleNeuronSizes = hiddenNeurons + [outputNum]

        #Create, initialize, and store every layer in the self.layers list
        self.layers = [Layer(possibleInputSizes[i], possibleNeuronSizes[i])
for i in range(self.hiddenNum+1)]
```

Now that we have every layer initialized and store, we can start setting up a feed forwards function that will take an input array and return the network's output. To accomplish this, we know we will need to propagate the input sequentially through each layer. We already have a function at the layer level to do this, so we just need to loop through and compute the output.

```
def feedForward(self, inputData):
    #need a temporary variable to store our current layer's output
    curData = inputData

    #loop through every layer, feeding the last output in as the new input
    for layer in self.layers:
        curData = layer.feed(curData)

    return curData #the last layer's output is the networks output
```

This should be able to take any 1D numpy array as an input (as long as it is the right size) and produce an output. This output will obviously be random as all the weights and biases are random. We are now done with forward propagation.

CODE CHECK-IN

Please test out the different aspects of the code and ensure everything is working right. Is the feed forward function producing an output? If so, is the output the correct shape? These are good things to test before you move on to the next part.

Here is what your code should look like right now if you have been explicitly following this module:

```python
import numpy as np

class Layer:
    def __init__(self, inputNum, neurons):
        self.weights = 2 * np.random.rand(inputNum, neurons) - 1
        self.bias = 2 * np.random.rand(1, neurons) - 1

    def feed(self, inputData):
        #Multiply input by weights
        out = inputData.dot(self.weights)

        #Add the bias
        out = out + self.bias

        #Apply the activation function
        out = 1 / (1 + np.exp(-out)) #Sigmoid activation function

        return out


class NeuralNetwork:
    def __init__(self, inputNum, hiddenNeurons, outputNum):
        #Storing the structure
        self.inputNum = inputNum
        self.hiddenNum = len(hiddenNeurons)
        self.outputNum = outputNum

        #Creating and storing the layers

        #First, get the possible input and output sizes (needed for layer construction)
        possibleInputSizes = [inputNum] + hiddenNeurons
        possibleNeuronSizes = hiddenNeurons + [outputNum]

        #Create, initialize, and store every layer in the self.layers list
        self.layers = [Layer(possibleInputSizes[i], possibleNeuronSizes[i]) for i in range(self.hiddenNum+1)]

    def feedForward(self, inputData):
        #need a temporary variable to store our current layer's output
        curData = inputData

        #loop through every layer, feeding the last output in as the new input
        for layer in self.layers:
            curData = layer.feed(curData)

        return curData #the last layer's output is the networks output
```

THE NEURAL NETWORK CLASS: BACKPROPAGATION

Alright, let's back up for a second and look at what we've coded.

We have:

1. A Layer class, capable of taking an input array and producing an output array. It has weights and biases associated with it.
2. A Neural Network class which contains the structure of our network, a list of its layers, and a function to forward propagate an input.

We need:

1. A back-propagation algorithm which will adjust the weights and biases of every layer to step closer to the network always providing the desired output.

Our back-propagation (or 'train') function is going to need a few things too. As inputs, we will need an input array with it's expected output. This will allow us to compute our cost function, which as we know is the backbone for backpropagation. Let's define it now:

```
def train(self, inputData, expectedOutput):
    #Get current output
    curOutput = self.feedForward(inputData)
    #Compute the cost (we will just use expected - actual)
    cost = expectedOutput - curOutput
```

In addition, we will need the output at every layer to compute the gradient of the activation function. We could do some fancy math and propagate the input a certain number of times, or we could just store the output of every layer when doing forward propagation. To do this, we simply add a new variable to the class in the constructor:

```
#Store the layer outputs
self.layerOutputs = [None for i in range(len(self.layers))]
#initialize the layer outputs to be None
```

Now we modify our feed forward function and simply save every layer output to this variable:

```
def feedForward(self, inputData):
    #need a temporary variable to store our current layer's output
    curData = inputData

    #loop through every layer, feeding the last output in as the new input
    index = 0
    for layer in self.layers:
        curData = layer.feed(curData)
        self.layerOutputs[index] = curData
        index = index + 1

    return curData #the last layer's output is the networks output
```

We still need one more thing before we complete our train function, the learning rate, which we will define in the constructor and give a default value of 0.05:

```
#Defining the learning rate
self.learningRate = 0.05
```

And then we can write some getter and setter functions for this learning rate:

```
def setLearningRate(self, newLR):
    self.learningRate = newLR
def getLearningRate(self):
    return self.learningRate
```

Alright, time to move into the train function.

As usual, let's build up a checklist of what this function needs to do:

- Loop backwards through every layer
- On each iteration, calculate the gradient of the cost function at that point
    - Therefore, we need the value of the cost function at each iteration
- We need the value of the derivative of the activation function at each layer
    - Which means we need the output of every layer

Let's begin with our loop, like the one in the feed forward function except we are looping backwards:

```
def train(self, inputData, expectedOutput):
    #Get current output
    curOutput = self.feedForward(inputData)
    #Compute the cost (we will just use expected - actual)
    cost = expectedOutput - curOutput

    #Get the list of layer outputs
    layerOutputs = [inputData] + self.layerOutputs

    #Loop through every layer backwards
    for i in range(len(self.layerOutputs)):
        #Do a little math to get the correct index
        #(as we need to loop backwards)
        index = len(layerOutputs) - 1 - i
```

We now have the index of the layer to modify at every loop iteration. Next, we need to compute the gradient of the cost. As we can recall, the formula for it is:

$$Gradient\ of\ Cost = Cost * Derivative\ of\ Activation\ Function$$

This should be easy enough to compute. We already have the cost and we can calculate the derivative of our activation function. In this case it's just:

$$f'(x) = f(x) * (1 - f(x))$$

```
#Compute the gradient
gradient = cost * (layerOutputs[index] * (1 - layerOutputs[index]))
```

The gradient is used to compute the weight changes. From module three we know this is equal to:

$$Weight\ Changes = Learning\ Rate * \ Gradient * \ Inputs\ from\ Previous\ Layer$$

But that is the formula for the weight change of one connection. To compute the weight change for every weight we will use the dot product. If we make the gradient a column vector and keep the inputs as a row vector, then gradient $*_{dot}$ prev. inputs will give us this! Try the matrix math out for yourself to see why.

```
#Compute the weight change
if (layerOutputs[index-1].ndim == 1):
    layerOutputs[index-1] = layerOutputs[index-1].reshape(1,
len(layerOutputs[index-1]))
weightChange = self.learningRate * (layerOutputs[index-1].trans-
pose().dot(gradient))

#Compute the bias change
biasChange = self.learningRate * gradient
```

The transpose function will simply flip the layer output on its side, turning it from a row vector to our needed column vector. The if statement exists because transpose operates differently for 1D and 2D matrices. We must ensure that layerOutputs is 2D, so the transpose works correctly. This is a quirk of numpy. Because the learning rate is a scalar, it will be applied to every value in the created matrix.

Now we can simply add this change to our current weights to produce our new ones!

```
#Get the new weights at layer[index-1] (this is just a consequence of how we
setup our indexing, these are still the weights for the current layer)
self.layers[index-1].weights = self.layers[index-1].weights + weightChange
```

The change in the bias is just the gradient as it is unaffected by the state of the current weights:

```
#Get the new biases
self.layers[index-1].bias = self.layers[index-1].bias + biasChange
```

Perfect, there is only one more thing we have left to do in each iteration. We need to calculate the cost function for the layer before us, but how do we do that? We can simply propagate the cost values backwards by doing the reverse of forward propagation. This is why this algorithm is sometimes called 'back-propagation', we are literally propagating our error backwards through the network.

```
#Propagate the cost backwards one layer
cost = cost.dot(self.layers[index-1].weights.transpose())
```

Because we have our for-loop setup properly, this should run through every single layer, nudging the weights and biases so the output gets closer to what we want. We have now completed our train function! It should look something like this:

```python
def train(self, inputData, expectedOutput):
    #Get current output
    curOutput = self.feedForward(inputData)
    #Compute the cost (we will just use expected - actual)
    cost = expectedOutput - curOutput

    #Get the list of layer outputs
    layerOutputs = [inputData] + self.layerOutputs

    #Loop through every layer backwards
    for i in range(len(self.layerOutputs)):
        #Do a little math to get the correct index
        #(as we need to loop backwards)
        index = len(layerOutputs) - 1 - i

        #Compute the gradient
        gradient = cost * (layerOutputs[index] * (1 - layerOutputs[index]))

        #Compute the weight change
        if (layerOutputs[index-1].ndim == 1):
            layerOutputs[index-1] = layerOutputs[index-1].reshape(1, len(layerOutputs[index-1]))
        weightChange = self.learningRate * (layerOutputs[index-1].transpose().dot(gradient))

        #Compute the bias change
        biasChange = self.learningRate * gradient

        #Get the new weights at layer[index-1] (this is just a consequence of how we setup our indexing,
        #these are still the weights for the current layer)
        self.layers[index-1].weights = self.layers[index-1].weights + weightChange

        #Get the new biases
        self.layers[index-1].bias = self.layers[index-1].bias + biasChange

        #Propagate the cost backwards one layer
        cost = cost.dot(self.layers[index-1].weights.transpose())
```

USING THE NEURAL NETWORK: THE XOR TEST

We have completed everything from our original checklist, we are effectively done our neural network! Now let's test it out with a simple example. We are going to train it to understand the bitwise, XOR function. This is a good example to test our network with because it is **not linearly separable**. This means that a line cannot be drawn in the output space to separate the two possible answers. If our network can figure this out it means it has more complexity than a linear regression, which is needed to do anything useful.

First, we initialize our neural network and get some training data ready:

```
#The XOR Test

# Initialize the network
nn = NeuralNetwork(2, [5], 1)

# Setting up the training data
data = []
data.append(np.array([0, 1]))
data.append(np.array([1, 0]))
data.append(np.array([1, 1]))
data.append(np.array([0, 0]))
# The expected output for every input added above
out = []
out.append(np.array([1]))
out.append(np.array([1]))
out.append(np.array([0]))
out.append(np.array([0]))
```

Above, we have defined a NeuralNetwork object (called nn) that has 2 inputs, 1 hidden layer with 5 neurons, and 1 output. We then created all our training data (that we will randomly sample from) including the expected output for each of the inputs.

We can now use a for-loop and train our network a certain number of times:

```
#Train the network
import random as rand
for i in range(50000):
    index = rand.randint(0, 3)
    nn.train(data[index], out[index])
```

In this example above, we are training it fifty-thousand times, each time randomly selecting a data point to train with on each iteration. There is lots of theory and ideas on how to properly train a neural network however that is a topic for another time.

Once the network has been trained, we should be able to test it out:

```
for input in data:
    print(nn.feedForward(input))
```

If all went according to plan, something like this should have outputted:

```
[[0.96463616]]
[[0.97456754]]
[[0.02862707]]
[[0.02903726]]
```

As you can see our outputs are converging to our desired values! If you had run the inputs through the network before training, you would have gotten random outputs.

CLOSING REMARKS

If you were able to follow along with this module without simply copy and pasting code, give yourself a pat on the back, that is not an easy task. That means you have the understanding and ability to program a basic neural network which sets you well above most in terms of skill. In the future I will be showing you how to program exactly what we did above, but with tensor flow. You will notice there will be even less code. This is how most people begin with neural networks, however now that you have made one from scratch, I hope you will be able to understand how it works better. Tensorflow, and many other AI and machine learning libraries, are incredibly useful and used widely in industry. It is vital you learn them however I have always believed its better to create something from scratch to begin with, to ensure you truly understand it.

I took the same process as we did to create my own neural network package called NeuralNetworkPY, which you can install on your computer with "pip install NeuralNetworkPY". You can view the source code and everything else here. It is stable, with many more features added to it than what we did. If you are stuck when building your own, I suggest you look at this package as it may make something click in your brain. Play around with it as it is quite user friendly. There are a couple more examples of tests you can run with your own neural network (such as training it on the MNIST data set).

Something else to consider is adding different features to your network library. There is so much more one can do, such as adding support for different activation functions, being able to save the network and load it later and adding support for accepting different kinds of input. All of these are stretch goals once you have polished the basic version outlined in this module.

I'd like to thank you for sticking around to the end, this is not easy material and it takes a strong mind to continually learn and problem solve on the fly. In a future module, as mentioned above, I will show you the basics of tensorflow and give you many ideas for what you can do with your neural network. There are many data sets and uses for AI out there, my hope is you will take what you have learned here and do something great with it.

REFERENCES

[1] D. Shiffman, "YouTube," The Coding Train, 18 January 2018. [Online]. Available: https://www.youtube.com/watch?v=qWK7yW8oS0I&index=12&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[2] D. Shiffman, "YouTube," The Coding Train, 22 January 2018. [Online]. Available: https://www.youtube.com/watch?v=MPmLWsHzPlU&index=13&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[3] D. Shiffman, "YouTube," The Coding Train, 23 January 2018. [Online]. Available: https://www.youtube.com/watch?v=QJoa0JYaX1I&index=14&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[4] D. Shiffman, "YouTube," The Coding Train, 24 January 2018. [Online]. Available: https://www.youtube.com/watch?v=r2-P1Fi1g60&index=15&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[5] D. Shiffman, "YouTube," The Coding Train, 5 February 2018. [Online]. Available: https://www.youtube.com/watch?v=8H2ODPNxEgA&index=16&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[6] D. Shiffman, "YouTube," The Coding Train, 6 February 2018. [Online]. Available: https://www.youtube.com/watch?v=qB2nwJxNVxM&index=17&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[7] D. Shiffman, "YouTube," The Coding Train, 7 February 2018. [Online]. Available: https://www.youtube.com/watch?v=tlqinMNM4xs&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh&index=18. [Accessed 2018].

[8] D. Shiffman, "YouTube," The Coding Train, 12 February 2018. [Online]. Available: https://www.youtube.com/watch?v=188B6k_F9jU&index=19&list=PLRqwX-V7Uu6aCibgK1PTWWu9by6XFdCfh. [Accessed 2018].

[9] A. Farley, "PyPi," 28 December 2018. [Online]. Available: https://pypi.org/project/NeuralNetworkPY/. [Accessed 2018].