COSC 4P80
- Assign 2

Feed Forward Network

Prepared by: Alex Duclos
Instructor: Dave Bockus
Date: March 8, 2023

Abstract

Developing a neural network to figure out if a electric motor is malfunctioning based on behaviour. Using a Fourier transform we create a multi-layer neural network to distinguish data. Using back propagation to learn.

Contents

1 Setup **1**

2 Supporting Code **1**

2.1 Code Explained 2

3 Discussion Part 1 **2**

4 Discussion Part 2 **3**

5 Discussion Part 3 **5**

6 Discussion Part 4 **6**

7 Discussion Part 5 **8**

8 Conclusion **9**

1 Setup

First I created a neural network which can be adapted based on user inputs. Allowing user to specify the hidden layer count and to automatically create a input layer based on the length of the input. I created the initial weights using a random distribution function. The network has two functions for data, one forward which sends the data forward and returns the expected output, which is 0 or 1, as well as a backwards function which calculates the error and modifies the weights to try to correct this error.

We can train this network by using these functions in tandem to identify what output the network gives for an input, calculating the error off of it, and back-propagating for the entire data set for every epoch.

We used a sigmoid activation function as well as the numpy library to handle the math behind the network, as well as speeding up all operations as the underlying code for numpy is written in C. We used matplot lib to form the graphs.

We also normalize out inputs. The network also dynamically creates the input layer size based off of the inputted file.

2 Supporting Code

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

def backward(self, X, y, output, learning_rate):
    # Compute the difference between the output and the true label
    output_error = y - output

    # Compute the derivative of the output layer and update weights
    output_delta = output_error * sigmoid_derivative(output)
    self.weights2 += learning_rate * np.dot(self.hidden.T, output_delta)

    # Compute the derivative of the hidden layer and update weights
    hidden_error = np.dot(output_delta, self.weights2.T)
    hidden_delta = hidden_error * sigmoid_derivative(self.hidden)
    self.weights1 += learning_rate * np.dot(X.T, hidden_delta)

def train(self, X, y, num_epochs, learning_rate):
    for i in range(num_epochs):
        # Compute the output of the network for the current input
        output = self.forward(X)

        # Compute the mean squared error of the output
        error = np.mean((output - y) ** 2)

        # Update the weights based on the output and true labels
        self.backward(X, y, output, learning_rate)
```

```python

def forward(self, X):
    # Compute the dot product of the input with the first set of weights, and apply the sigmoid function
    self.hidden = sigmoid(np.dot(X, self.weights1))

    # Compute the dot product of the hidden layer with the second set of weights, and apply the sigmoid function
    output = sigmoid(np.dot(self.hidden, self.weights2))
    return output
```

Listing 1: Python example

2.1 Code Explained

These are the primary functions in the neural network. Sigmoid and Sigmoid derivative are used in backpropegation and backward and train are used to initalize the matrix.

Calling nn.foward() will allow you to predict an output, which is used both for testing, training, and actually getting usable results out of the network.

3 Discussion Part 1

I used a static learning rate of 0.1 The MSE over 10000 epoch is as follow:

Table 1: MSE per Epoch

Epoch	**MSE**
0	0.28015124592765495
1000	0.015238976324625628
2000	0.0053447927813529954
3000	0.002537586348674161
4000	0.0015096883581622543
5000	0.00102516931339903
6000	0.0007560355485122358
7000	0.0005892362708036518
8000	0.00047758474800363253
9000	0.00039849267690033376

We can see that it starts with a .28 and quickly goes down to .02 after only 1000 epochs. More epochs gets a more specific network. This was on the 1000 sample size training data with 12 hidden nodes.

Epoch	MSE
0	0.2973518896461779
1000	0.08006745135601322
2000	0.05881557965133051
3000	0.044334162834307696
4000	0.03531914796699035
5000	0.029991267475922973
6000	0.02678223126552964
7000	0.024762332351237735
8000	0.023430424622077842
9000	0.02251373317143947

Using 6 hidden nodes with 150 sized training data, we get

We can see here that the optimal amount of MSE is 0.02. So the ideal and most specified input for our network is dependent both on input size as well as hidden layer size.

Furthermore, we implement a test network function, which allows us to test an input based off a neural network as an input.

4 Discussion Part 2

Using the 150 sized file, and 10,000 epochs we generate this graph [Figure 1].

We can see that all inputs trend together.

When using 25 as a hidden size for the same input, we can generate [Figure 2]. We can see in image 2 a much quicker conversion, taking less than half as many epochs for all the training sets to converge.

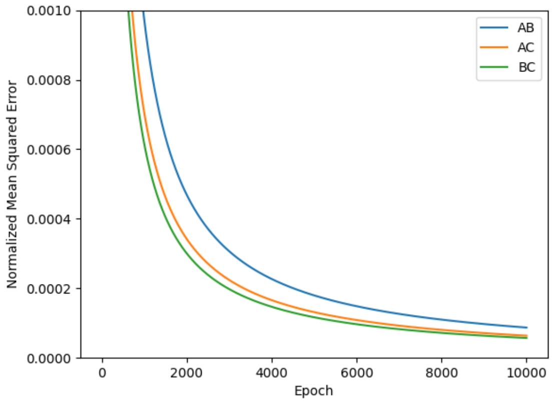

Figure 4.1: 150 input, 10000 epochs, 8

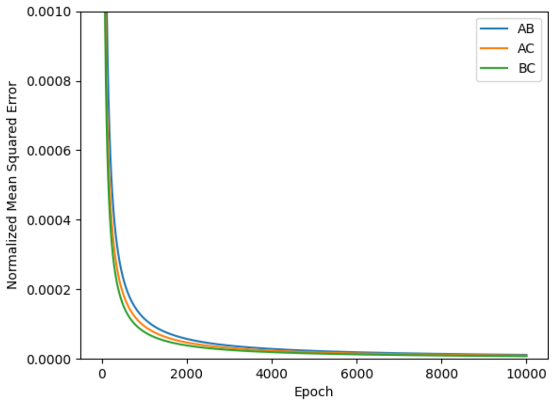

Figure 4.2: 150 input, 10000 epochs, 25 hidden

5 Discussion Part 3

Implementing momentum very quickly allows us to converge. I used an initial value of .9 and multiplied it by .9 for every epoch as long as it remained above 0.5. Continuing to use the 150 input training set with 8 hidden nodes, we create this data-set using the same splitting as in Part C.

Epoch	**MSE**
0	0.6738298155661789
5	0.03979317883930451
10	0.014718938398791081
15	0.009017340839524442
20	0.Epoch 20, MSE: 0.006453993235001123

Table 2: MSE values for the neural network with a learning rate of 0.9.

Epoch	**MSE**
0	0.7131921298303292
5	0.01641538707289942
10	0.007956890088508728
15	0.005290668342611634
20	0.003953661542173633

Table 3: MSE values for the neural network with a learning rate of 0.9.

As we can see from these tables, it only takes fewer epochs to generate a MSE which is low compared to the previous parts. A graph is [Figure 3]

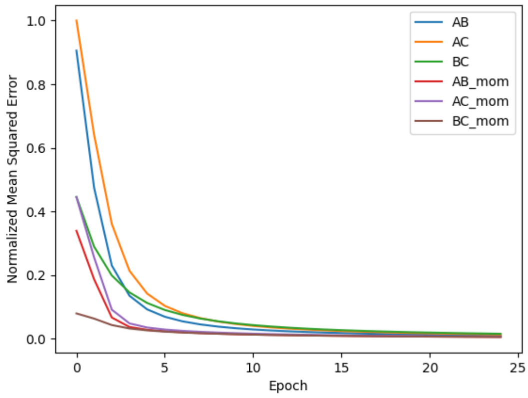

Figure 5.1: Momentum Race.

6 Discussion Part 4

For this part we used 150 input size again, with 8 hidden inputs. We'll use 100 epochs for this portion.

Looking at this table, we can see some very obvious things. Training set A and B always cause a high MSE. This is due to the uneven nature of the data set. Furthermore, training set BC generated a terrible result and even a MSE near 1.

The best results occur when using AX and C as our test set.

Table 4: MSE for different training and test sets

Training Set	Test Set	MSE
AB	B	0.5101357447050856
AB	A	0.5101704637535027
AB	C	0.0003136482533078072
AC	B	0.5113393051410082
AC	A	0.5114976716671992
AC	C	0.0005603099963618362
BC	B	0.45984055367859344
BC	A	0.45525311189230727
BC	C	0.9687893598238666
AB	B	0.5006573308965951
AB	A	0.5006207892715613
AB	C	0.0005676994714459097
AC	B	0.5060711660774762
AC	A	0.5063464172999042
AC	C	0.0008245561459402124
BC	B	0.45193490601748487
BC	A	0.4502200094671267
BC	C	0.9547093783465248

7 Discussion Part 5

In this section we expand the neural network to have 3 hidden layers. This remains dynamic like the previous parts, where the user can select the number of nodes for each layer. The fundamentals of the neural network don't change here either, back propagating from layer N to layer N-1, and forward feeding from layer N to N+1.

In this section we will use 6 hidden nodes on each hidden layer.

When using only 10 epochs, a number which was completely fine in the previous part

Training Set	Test Set	MSE
AB	B	0.481025264706865
AB	A	0.48100277044854295
AB	C	0.0021487104532940562
AC	B	0.44933642333026874
AC	A	0.4493429071394288
AC	C	0.006501194178445878
BC	B	0.4340856940162583
BC	A	0.43387600626338024
BC	C	0.9170176233876534
AB	B	0.448339305168056
AB	A	0.44825906440435737
AB	C	0.006548612993577
AC	B	0.4450806607045077
AC	A	0.44518982002687735
AC	C	0.007314290323871856
BC	B	0.43868399271401676
BC	A	0.4382040096951598
BC	C	0.9269480701668228

The MSE of these charts are at best equal, but in many cases worse. If we expand to 1000 epochs, however, we get a table such as this.

Training Set	Test Set	MSE
AB	B	0.5258119509902481
AB	A	0.5258119371619241
AB	C	3.4692064770355837e-06
AC	B	0.5260299458357827
AC	A	0.5260299669051591
AC	C	2.7425902893942473e-06
BC	B	0.470684726866821
BC	A	0.470636053854254
BC	C	0.9966896264127715
AB	B	0.5253575575532294
AB	A	0.5253575200755032
AB	C	5.257848448062897e-06
AC	B	0.5255183242405246
AC	A	0.5255189749693983
AC	C	5.651470562569696e-06
BC	B	0.47035317543555066
BC	A	0.4703469572559509
BC	C	0.9960309729556156

In this table we can see that worst case scenario remains the same, but the MSE for our best case scenario rapidly drops, and since we aren't using our test set for training in these questions, we can be reasonably certain that we're not memorizing but instead doing a fantastic job of predicting.

8 Conclusion

Overall, this project required a fundamental understanding of FF neural networks. Implementing back propagation, and creating a flexible network were both required to succeed. Using sigmoid and numpy, we generated a quick and efficient network to generate consistent outputs.

Our results also show an importance in test set data. Multiple layers added to the network allow for a well generalized network, Splitting the data up gave a drastic difference in the effectiveness of the training.