HW3 WriteUp

Thomas McDonald
University of Colorado – Colorado Springs
tmcdonal@uccs.edu

Abstract

This paper entails my implementation of creating a multilayer nueral net that trains using backpropogation from scratch using python.

Approach

This assignment was to create a functional nueral net that can identify letters correctly. To do this we had to create a multilayered nueral net that trains with backpropogation. The architecture of my net has 62 input nodes, an initial value of 30 hidden nodes, and 7 output nodes. I chose 62 input nodes to represent each pixel in our 7x9 letter like the one shown below.

```
..##...
...#...
...#...
..#.#..
..#.#..
.#####.
.#...#.
.#...#.
###.###
```

An input node is on (1) if the pixel inputted is anything other than "." and off (-1) otherwise. I used 1 and -1 to represent an on and off state respectively because output values my activation function, a bipolar sigmoid, range from -1 to 1. The output layer of the net is used to identify which letter the net thinks is most probable. I have 7 nodes to identify which of the 7 letters we are identifying (A,B,C,D,E,J,K). For an input A, the ideal signals of the output nodes should be set to (1,-1,-1,-1,-1,-1,-1). For an input B the nodes should have signals (-1,1,-1,-1,-1,-1,-1) and so on. I implemented this project from scratch using python. The only libraries I'm using is random for the weights of each node, copy to deepcopy objects and math for the activation function. To begin, I made a node object that holds the signal value (-1 to 1), an array of weights for the next layers nodes, a variable for the amount of error, the delta weight to change each weight, the y_in which is the nodes bias plus the sum of the previous layers signal times the weight associated to the node, and the delta bias.

```
##node object for net
class Node():
    def __init__ (self):
        self.signal = 0
        self.weight = []
        self.error = 0
        self.deltaWeight = []
        self.y_in = 0
        self.bias1 = bias1
        self.bias2 = bias2
        self.deltabias1 = 0
        self.deltabias2 = 0
```

bias 1 is the bias for the hidden layer and bias 2 is for the output layer. To create my Nueral Net, I created a net object that holds an array of arrays, where each array holds nodes. The 0th entry is the array of input nodes, 1st entry for the hidden layer, and 2nd entry for the output nodes. The net object also has a buildNet method to build the net, feed to feed the net, feedForward which feeds the rest of the net from the input nodes, backProp to back propogate the net, and finally the update method which corrects the weights and biases based off the error. To calculate the signal, I used the bipolar sigmoid function to give me a signal (shown below).

```
def bipolarSigmoid(x):
    return (2 / (1 + math.exp(-1*x ))) - 1

def bipolarSigmoidx(x):
    return 0.5*((1+bipolarSigmoid(x))*(1-\
        bipolarSigmoid(x)) )
```

I made the values below consts to easily adapt my net and expirement with it. The variable names are self explanitory except for "trainHard", trainHard is the number of epochs I'm using to train the net.

```
inner_nodes = 30
output_nodes = 7
signal_const = 1
bias1 = 0.5
bias2 = 0.75
alpha = 0.2
trainHard = 100
```

The overall process of my program goes as followed:
Train

1. Collect the pixels from a letter and feed it to the input layer of the net
2. The net continues to feed through each layer giving a node a signal till it gets to the output layer

3. The net is given a target vector that represents the letter the net should have guessed and each node in the output layer calculates its error

4. Nodes in the hidden layer calculate their deltaweights and deltabias's from the output node's error

5. Nodes in the hidden layer now calculate their error

6. Using the error in the hidden layer, the input layer can now set their deltaweights and deltabias's

7. all weights and biases are adjusted based off the deltaweights and deltabiases

8. repeat 1-7 till there is no more letters to train from

Test

1. Collect the pixels from a letter and feed it to the input layer of the net

2. The net continues to feed through each layer giving a node a signal till it gets to the output layer

3. Retreive the output layer array and sort the signals in an descending order

4. The largest value represents the nueral net's best guess

5. Repeat 1-4 till there is no more letters to test

Experiments

As a base test, I used the following constants:

```
inner_nodes = 30
output_nodes = 7
signal_const = 1
bias1 = 0.5
bias2 = 0.75
alpha = 0.2
trainHard = 100
```

There's nothing special about these values other than this was the setup when my net started showing promising results. To calculate the average accuracy, I use the testing dataset on my net and test it 10 times while summing the reported accuracy of my net and resseting the net after a test.

Results

Running the test with the initial constants I set, I was given an average accuracy of 87.14285714285712 which Isnt bad for disregarding the optimization of my net. With the tables below, you can see how the accuracy changes when changing the constants.

Changing the Number of Hidden Nodes

# of Hidden Nodes	Avg. Accuracy
10	68.57142857142856
20	89.04761904761904
30	87.14285714285712
40	84.76190476190476
50	84.28571428571426
100	72.38095238095238
200	50.476190476190474

you can see the lower range number of hidden nodes around 20 has the best accuracy and anything above or below that the accuracy of the net starts to decline.

Changing the Initial Bias of the Hidden Layer

Bias Value	Avg. Accuracy
-3	95.23809523809523
-2	85.23809523809523
-1	86.19047619047618
0.2	76.19047619047657
0.5	87.14285714285712
0.8	80.9523809523812
1	90.47619047619071
2	80.9523809523812
3	85.7142857142859

Seems like the bias changing doesn't have that much of an effect of the net's accuracy

Changing the Initial Bias of the Output Layer

Bias Value	Avg. Accuracy
-3	95.23809523809537
-2	84.28571428571426
-1	96.18654268465448
0.2	71.42857142857115
0.5	85.7142857142859
0.75	87.14285714285712
1	71.42857142857115
2	80.9523809523812
3	71.42857142857115

Changing the bias here doesn't show anything that really correlates here as well.

Changing the Epochs

# of Epochs	Avg. Accuracy
50	84.28571428571429
100	87.14285714285712
200	92.85714285714285
400	92.38095238095238
1000	90.95238095238093

You can see the more runs I do the accuracy gets worse from over fitting.