Module Code: COC102

Module Title: Advanced AI Systems

Title: ANN Implementation

Student Name: Max Pearson

Student ID: B123103

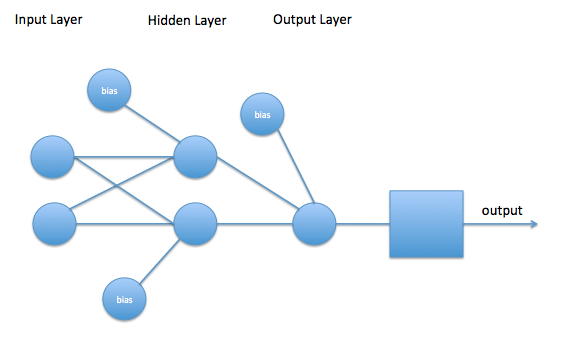
Introduction

The task was to construct a program called a Neural Network Multi-Layer Perceptron using a back propagation algorithm. This algorithm calculates errors and finds classification by using mathematical functions such as: the hyperbolic tangent function, the logistic function and chain rule to derive changes from neuron to neuron.

The program works by setting 3 sets of Neurons called input, hidden and output. These sets or ‘layers’ can communicate with each other through weights, which are initially random.

*Figure1* is an example of a Multi Layer Perceptron with 2 inputs nodes, 2 hidden nodes and 1 output node, including bias for input and hidden.

Figure 1.



1. Implementation

Program Details

Classes

* **NN**: Main Neural Network Class.
* **Neuron**: Contains index, activation, delta, sum and bias for each neuron.
* **Layer:** Holds multiple Neuron objects for each Input, Output and Hidden layers.
* **Data** (Processing data from CWK.xlsx and formatting for use with NN.
* **Layers:** Holds multiple Layer objects, which are the structure of the Network, allowing NN full access to all Neurons with a simple function.

Variables and Objects

* NN
  + netName
  + filename
  + states
  + learning\_rate
  + momentum
  + Network
  + trainingErrors
  + validationErrors
  + learningRates
  + testDataPredictions
  + testDataOutputValues
  + testErrors
  + exitCounter
* Data
  + Document
  + tagLine
  + maxN
  + minN
* Layers
  + Layers
  + NumNeurons
  + NumLayers
* Layer
  + name
  + Neurons
* Neuron
  + activation
  + n
  + delta
  + S
  + bias

Methods

* NN
  + *feed\_forward*
  + *backPropagation*
  + *decayWeight*
  + *train*
  + *test*
  + *getError*
  + *bold\_driver*
  + *captureState*
  + *getStateTypeAt*
  + *saveNetwork*
  + *getWeights*
  + *getW*
  + *getChanges*
  + *getC*
  + *getLayer*
  + *getLayerNeurons*
  + *getLayerSize*
  + *getLayerNeurons*
  + *getPrediction*
  + *runProgram*
  + *initialiseWeights*
  + *getError*
  + *checkCount*
  + *plotErrors*
  + *plotLearning*
  + *plotPrediction*
* Data
* Layers
  + addLayer
  + updateLayerSize
  + \_\_updateWeightSpace
* Layer
  + \_\_createLayer
* Neuron
  + setDelta
  + activate
  + setSum

Functions contained in mlp.py

* plotNN
* saveNN
* loadNN
* loadNetwork
* showNetwork
* getNetworkArrays
* runNetwork

Functions contained in Utility.py

* vector
* plotNetworks
* viaTan
* viaSigma
* normalise
* setWeights
* hyperbolic\_tangent
* hyperbolic\_tangent\_dv
* generateRandFor
* sigmoid
* sigmoid\_dv
* activation\_function
* derivative\_function
* createNormalisedDataSet

Neural Network Functions Implemented

*= activation of layers*

*= delta at layer*

Feed Forward

Back Propagation

Update Weights

Weight Decay

Bold Driver

ANN Configuration

The Neural network created in this report has the following configurations:

3 Layers made up of Input, Hidden and Output, 3 parameter rates including learning rate, momentum and weight decay which are set to a default of 0.5, 0.9 and 0.03. The Data that is accepted by the network has to be in the form shown in Figure 1 and has to be in the ranges of [0,1] or [-1,1], or the network will not be able to function when weights get too large. (logistic function will throw stack overflow error)

The functions contained within the Network are feed forward, back propagation, weight decay and bold driver, unfortunately during the implementation of the algorithm the function annealing was not properly finished and as such may have affected its overall performance.

The mathematical formulation of each functions use in the Neural Network are detailed above and are fully integrated and fully functional within the overall system. However in order to show the data going in and out of the Neural Network i.e processing and outputting, it had to be able to use external functions to plot graphs and format data streams and as such functions and classes such as Data, Layers, Layer and Neurons were used to simplify the program in an object sense.

Figure 2.

Network Details

*LR*: Learning rate parameter

*Mom*: Momentum parameter

*WDF*: Weight Decay rate parameter

*Hidden*: Hidden nodes

*Inputs*: Input nodes

*Training Iterations*: Run training set N times before validation set.

Figure 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Network** | **LR** | **Mom** | **WDF** | **Hidden** | **Inputs** | **Training Iterations** |
| **1** | 0.5 | 0.9 | 0.03 | 8 | 8 | 10 |
| **2** | 0.5 | 0.8 | 0.025 | 7 | 8 | 20 |
| **3** | 0.4 | 0.9 | 0.025 | 6 | 8 | 10 |
| **4** | 0.4 | 0.8 | 0.020 | 5 | 8 | 20 |
| **5** | 0.3 | 0.9 | 0.020 | 4 | 8 | 10 |
| **6** | 0.3 | 0.8 | 0.015 | 3 | 8 | 20 |

*Performance Rating*: Performance rating based on average time to execute.

*Best Execute*: Minimum Epoch Termination of Test 1, 2 and 3.

*Training Set*: % of Data Set for Training Network

*Validation Set:* % of Data Set for Validation of Network

*Test Set*: % of Data Set for Testing Network

Figure 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Network** | **Performance Rating** | **Execution (epochs)** | **Training Set** | **Validation Set** | **Test Set** |
| **1** | 5/10 | 36 | 70% | 15% | 15% |
| **2** | 3/10 | 48 | 70% | 20% | 10% |
| **3** | 6/10 | 32 | 70% | 15% | 15% |
| **4** | 8/10 | 18 | 70% | 20% | 10% |
| **5** | 4/10 | 38 | 70% | 15% | 15% |
| **6** | 7/10 | 20 | 70% | 20% | 10% |

Proposed Network Configuration

Hidden Nodes = [ 3 – 5 ]

Momentum = 0.8

Weight Decay factor = [0.3 - 0.4]

Training Iterations = 20

Network Predictions

Figure 5







Network Errors

Figure 6







Network Learning Rate

Figure 7







Figure 8

Network Analysis

In order to accurately measure the differences between the networks the data has been graphed to illustrate changes in error, learning rate and predictions. Overall the 6 networks performed very well obtaining a minimum point in less than 50 epochs, however in deciding these networks problems were encountered, if the number of iterations were set less than 10 with 8 input nodes and any amount of hidden nodes between 2 and 10 the network didn’t find a minimum until epoch 200+ and sometimes would not terminate in the validation set even after 400 epochs, however when setting the training iterations between 10 and 20 the epoch terminations became fairly consistently terminating after 10-100 epochs.

In the graphs produced it is fairly obvious to see that the higher the validation set % the faster the network is not only finding a low error but also converging to a specific point faster. Network 2 did encounter problems as it peaks too quickly but this may be because of the termination value currently set at 10 as setting it at say 6 would of allowed termination around the 15-20 epoch mark. Overall it’s easy to see that the best performing networks were networks that started on a low learning rate had a lower momentum, a high weight decay and a middle amount of hidden nodes. Learning too quickly may be a problem if the hidden nodes are too high and the learning rate should be low if it wants to build up a steady increase in weight changes, obviously epochs are going to counter balance the learning rate through weight decay but both have to complement each other.

Ideally a Network should only spend a small amount of time at its peak learning rate and decrease rapidly from there and as such Networks 4 and 6 display properties that make it more desirable for faster learning and for lower error rates.