NEAT ACW

# Question 1

When we apply perceptron learning to the dataset of question 1, we can observe that the network was unable to learn the data set. The graph below shows us that the total error did not descend after 1000 epochs. If any learning was taking place it would be expected that the error term would slowly descend. Additionally the network output graph below can explain why. As the expected outputs fall across a range of values, and the perceptron is able to output either a 1 or 0, it is unable to generate suitable outputs to generalise the expected outputs. Even if the outputs of the perceptron were scaled to match the range of the network outputs it still would be unable to produce the intermediate values.

If we replace the perceptron’s activation function with the logistic sigmoid activation function, we can observe that the network begins to learn. The error graph below shows that the error gradually reduced over a number of epochs of training. Additionally the output graph produces a reasonably close generalization of the data set. We can conclude that as the neuron is able to output a range of values, it is able to learn the data set, where the perceptron could not.

The following diagrams describe the perceptron (left) and the neuron (right) as implemented. The source code of the implementation can be found in appendix A.

# Question 2

For question 2 several networks were trained using MATLAB. The first two networks for comparison are a network with 2 hidden layers with 3 neurons per layer, and 2 hidden layers with 10 neurons per layer.

The graph suggests that the networks with more neurons are able to reduce the error the furthest. And thus produce a better generalisation.

The following graph describes the training of the same networks, this time with a momentum term introduced.

In both cases the networks trained with momentum are able to produce a slightly better result after the first 1000 epochs. However it can be noted that the gradient of the networks with the momentum term are steeper after the first 1000 epochs, so should the networks be trained for longer, they may produce better results. In the case of the 10-10 network we can see that the network trained with momentum begins with a lower gradient than the network trained without and also ends with a higher gradient than the network trained without momentum. This suggests that the momentum term has the effect of restricting higher gradients and increasing lower gradients, to stabilise the rate of training. The graphs of the 3-3 network with and without momentum are too similar to draw the same conclusion from.

The following graph introduces another 3 networks, a network with a single hidden layer (5 neurons) and two networks with 3 hidden layers (10 neurons per layer in the first network 3 neurons per layer in the second). Each network in the graph is trained with momentum.

As is to be expected the network with a single hidden layer does not perform well. The graph above shows that the mean squared error achieved was much higher than the other networks. Contrary to expectations the 10-10-10 network is unable to achieve a lower mean squared error than the 10-10 network. It is possible that at this point the network is over-parameterised as 30 neurons in total is far larger than the other networks. Interestingly the 3 layer network with 4 neurons per layer performs the best of all networks evaluated, achieving the lowest mean squared error. The 4-4-4 network has 12 neurons in total whereas the 10-10 network has 20 in total, so it is possible that the 10-10 network is over parameterised or that the 3rd hidden layer is accountable for the improved error.

The following graphs show the outputs of the worst network (single hidden layer with 5 nodes) and the best network (3 hidden layers 4 nodes per layer). We can see that the worst network does not achieve a good classification as there are many outputs which are some way between the two expected output values, 0 and 1. The second graph shows that by contrast the best network achieves a good classification, the majority of values being very close to 0 and 1, with only a few that are some distance from the expected values.

The following graph shows that when using a decision boundary of 0.2 and 0.8 there are very few values which are unclassified, and all of them lie on the boundary of the original dataset. If we compare this graph with that of the original data set (as seen below), we can conclude that the 4-4-4 network provides a good generalisation of the original data.

# Appendix A – Program listing for Question 1

The source code of the C++ program developed for question 1 follows:

#include <fstream>

#include <vector>

#include <cmath>

#include <ctime>

class Perceptron

{

public:

Perceptron(int numInputs) :

\_inputs(numInputs, 0),

\_weights(numInputs, 0),

\_bias(0)

{

}

void RandomiseWeights()

{

srand((unsigned int)time(0));

for (unsigned int i = 0; i < \_weights.size(); ++i)

{

\_weights[i] = (rand() % 10000) / 10000.0;

}

}

double Sum()

{

double val = 0;

for (unsigned int i = 0; i < \_inputs.size(); ++ i)

{

val += \_inputs[i] \* \_weights[i];

}

return val;

}

double ActivateLogistic()

{

return 1.0 / (1 + exp(-(Sum() + \_bias)));

}

double ActivateStep()

{

if (Sum() + \_bias < 0)

return 0;

else

return 1;

}

double \_bias;

std::vector<double> \_inputs;

std::vector<double> \_weights;

};

bool ReadData(const std::string& filename, std::vector<double>& data)

{

std::ifstream file(filename.c\_str());

if (!file.is\_open())

return false;

data.clear();

double value;

while (file.good())

{

file >> value;

if (!file.fail())

data.push\_back(value);

}

file.close();

return true;

}

int main()

{

std::vector<double> inputData;

ReadData("data.csv", inputData);

std::ofstream errorlog ("error.csv");

const unsigned int numInputs = 3;

Perceptron p(numInputs);

p.RandomiseWeights();

unsigned int numIterations = inputData.size() - numInputs;

unsigned int numEpocs = 1000;

double learningrate = 0.05;

double bestError = -1;

for (unsigned int e = 0; e < numEpocs; ++e)

{

double error = 0;

for (unsigned int i = 0; i < numIterations; ++i)

{

for (int w = 0; w < numInputs; ++w)

p.\_inputs[w] = inputData[i+w];

double delta = inputData[i+numInputs] - p.ActivateLogistic();

error += delta \* delta;

for (unsigned int w = 0; w < numInputs; ++w)

p.\_weights[w] += delta \* learningrate \* inputData[i+w];

p.\_bias += delta \* learningrate;

}

errorlog << error << "\n";

}

errorlog.close();

std::ofstream outputlog ("output.csv");

for (unsigned int i = 0; i < numIterations; ++i)

{

for (int w = 0; w < numInputs; ++w)

p.\_inputs[w] = inputData[i+w];

double output = p.ActivateLogistic();

outputlog << output << "\n";

}

outputlog.close();

return 0;

}

Using the ActivateLogistic function for activation allows the neuron training to take place and replacing this with ActivateStep allows the training of a perceptron to be observed.