Artificial Neural Network Study:

Back Propagation Training on Two Data Sets

Seth Kurtenbach

Dr. Keller
Andrew Buck

0.1 Introduction

This report details the construction and implementation of an artificial neural network, with example applications. A neural network is a computational system inspired by the structure of a brain, with neurons connected by intricate webs of synapses. In an artificial neural network, nodes in a directed graph represent the neurons, and the weighted edges represent the synapses. With this structural scheme and a clever algorithm, artificial neural networks can learn from training data the parameters of a function, which can then be used to compute reliable output for novel input data. In its simplest form, a neural network of a single node can learn the line separating two distinct data sets, and accurately classify future data yet unseen. For this experiment, I wrote an artificial neural network module in Python that can create, train, and test three layer networks with any number of nodes. This report details the module's performance on two datasets, both provided from the instructor.

The artificial neuron, also called a perceptron, consists primarily of two functional components. First, it receives weighted input data and aggregates their values, usually through summation. This sum transfers internally to an activation component. The activation component is the analog to a biological neuron's firing in response to stimulus. The degree to which an artificial neuron is activated is a function of the sum of the weighted inputs. Figure 1 on the next page illustrates these ideas.

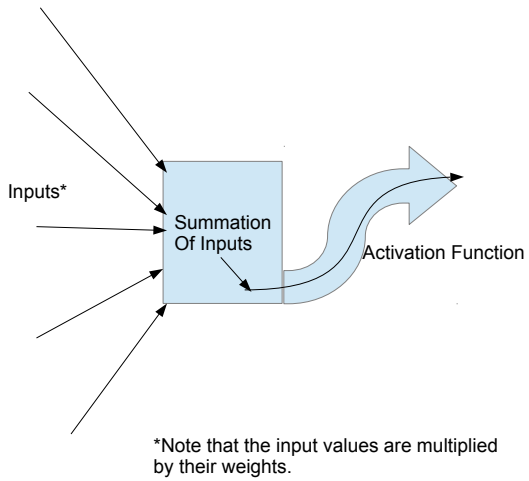

Figure 1: A single artificial neuron, also called a perceptron.

Though a single perceptron is computationally limited to data that can be separated by a straight line, networks of perceptrons can compute a boundary arbitrarily close to the true function, whatever it is. This brings up an important point about neural networks. The function that the neural network computes is only ever an estimate, based on the training data that the network has received. For any given data points, there are infinitely many curves that are consistent with them, and thus that in some sense fit. The goal of training a neural network is to achieve a balance between accuracy and simplicity. Simplicity can be thought of as the number of nodes in the network, which in turn correspond

to the degrees of motion available to the curve. If two curves fit the data equally well, then the one with fewer variables is preferable. Conversely, by increasing the number of nodes in a network, one adds degrees of motion to the curve, thereby allowing it to better fit the data. One might be tempted to introduce as many nodes as one needs in order to fit the training data perfectly. This is ill-advised, because over-fitting a curve to the training data incorporates noise from the data into the function, which diminishes the network's accuracy when faced with novel input data.

In Part A, I put a neural network to work on data composed of two datasets. Each item consists of an x and y coordinate value, and the goal is to identify the best curve dividing the data. Figure 2 on the next page illustrates the neural network to be used in Part A.

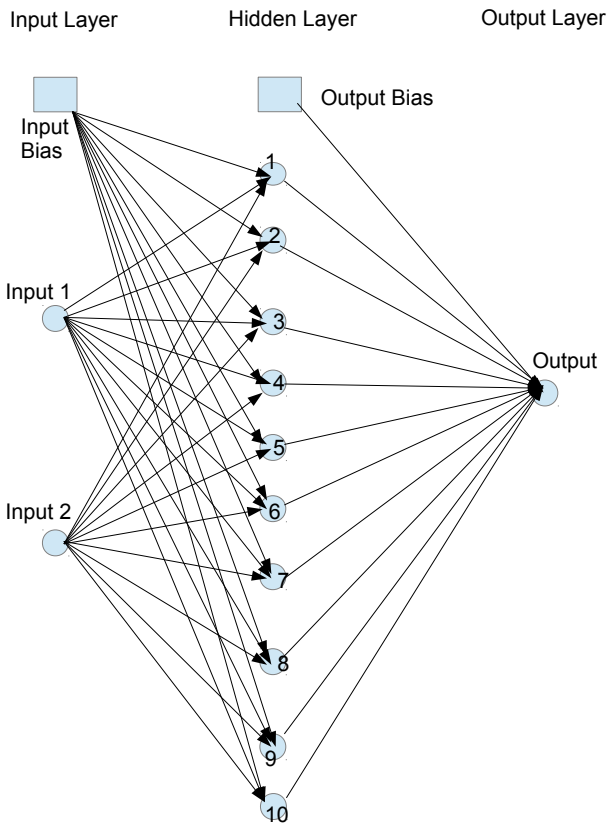

Figure 2: A neural network with 2 inputs, 10 hidden layer nodes, and 1 output node.

I now turn to a technical descriptions of the techniques used in this experiment.

0.2 Technical Description of Techniques

Boiled down to its simplest form, training the neural network consists of five steps. First, initialize the network's size and weights. Second, feed a weighted

input item forward through the network, summing each node's incoming values, then applying the activation function to the sum, and passing this forward to all subsequent nodes. Third, calculate the error of the output. Fourth, update each node's weights according to how much it contributed to the network's error. Fifth, check to see if a stopping condition has been met; if so, return the weights, and if not, continue training. I provide the mathematics of this process, which is illustrated by Figure 3 below.

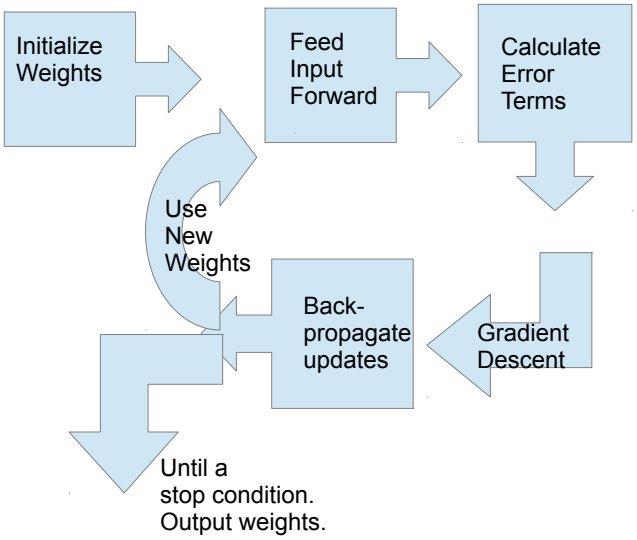

Figure 3: A flowchart of the backpropagation training algorithm.

Let each input x_i relate to each hidden node n_j via weight $w_{i,j}$. Let v_j be the result of the aggregation function. Then,

$$v_j = \sum_{j=1}^{k}(x_i w_{i,j}) + bias_j,$$

where k is the number of nodes in the hidden layer, and $bias_j$ is a threshold.

Each node receives input from a bias node, but the bias nodes themselves receive no input. Typically we think of the bias node as having a value of 1, and the threshold it contributes to v_j is equivalent to the weight of the connecting edge.

The activation function is a design decision made by the researcher, but the core idea is that it should model an appropriate notion of *on* and *off*. A simple step function models this exactly, with either a zero value or a one, depending on whether the threshold has been met. However, for many purposes, this is not desirable. In order to compute the error of the hidden layer nodes, we must be able to calculate the derivative of the activation function. The step function's derivative is either 0 or undefined, so it does not serve us. The sigmoid function is infinitely differentiable, and appropriately models something like the degree to which a node is *on* or *off*, or better yet, the intensity of its activation. The sigmoid function $\varphi()$, applied to the aggregate v_j, is,

$$\varphi(v_j) = y_j = \frac{1}{1+e^{-v_j}} \ .$$

As v_j increases, y_j approaches 1. As v_j decreases, y_j approaches 0.

Each node composes these two functions during its feed forward stage. The result is a y value for each output node.

The y value is next used to calculate the network's error, one node at a time. We begin with the output layer, because the target values for those nodes are given in the training data. There error, e_j, of an output node is,

$$e_j = (d_j - y_j),$$

where d_j is the target, or desired, output for the corresponding input. The error energy of the output node o_j is half its error squared:

$$E_j = \tfrac{1}{2} e_j^2.$$

With this, we can calculate the total error energy of the network, which we'll use to update its weights. The total error energy is simply the sum of each output node's error energy.

Since the training data does not provide desired outputs for the hidden layer nodes, the contribution of each to the output's error is calculated based on its weighted activation. The idea is that more heavily weighted hidden layer nodes contribute more to the output, so if the output is too high, then the hidden nodes' weights should be lowered proportionally to its intensity. If the output is too low, then the more activated hidden nodes need to contribute more. Formally, we seek the partial derivative of E with respect to each weight $w_{i,j}$. After some calculus we arrive at the following equation,

$\frac{\partial E}{\partial w_{i,j}} = -e_i \varphi'(v_i) y_j = \delta_i y_j,$

where $\varphi'(v_i)$ is the derivative of $\varphi(v_i)$. We call everything prior to y_j the *local gradient* of the weight, denoted by δ_i for each v_i.

The delta term of a hidden layer node j, which denotes the extent to which j contributes to E, is defined as,

$$\delta_j = \varphi'(v_j) \sum \delta_o w_{j,o} \text{ for each output node } o \text{ connected to } j.$$

Each weight is then updated according to its delta term and the neural network's learning rate, α, which has some value from the unit interval $(0, 1)$, determined by the network's user. The learning rate corresponds to how reactive a node is to is error contribution. The weight correction, or weight update, for a weight $w_{i,j}$, is,

$$\Delta w_{i,j} = \alpha \delta_i y_j,$$

which we add to $w_{i,j}$ to generate its weight for the next iteration.

I implement these steps by representing each input datum as a list [inputs, desired], and each layer's weights as a $j \otimes i$ matrix, for j receiving nodes and i incoming nodes. I apply the calculations through a sequence of matrix operations, storing each important value in a corresponding list, so that it can easily be accessed based on indexing. Algorithm 1 shows an example of this.

Algorithm 1 Python code for the main training loop up to the delta terms.

```
# 1. feed forward
# from inputs to hidden layer
for h in range(hids):
.....v = aggregate(invs, inweights[h], inbias[h])
.....hidvs.append(v)
.....y = phi(v)
.....hidys.append(y)
# from hidden layer to outputs
for o in range(outs):
.....v = aggregate(hidys, outweights[o], outbias[o])
.....outvs.append(v)
.....y = phi(v)
.....outys.append(y)
.....err = ds[o] - y
# 2. calculate output node deltas
outdeltas = [] # list of delta values for each output node
hiddendeltas = [] # list of delta values for each hidden node
for o in range(outs):
.....outdeltas[o] = (outys[o]*(1-outys[o])*(ds[o]-outys[o]))
# 3. calculate hidden node deltas
for h in range(hids):
.....for o in range(outs):
.....hiddendeltas[h][o] = (hidys[h]*(1-hidys[h])*
sum(outdeltas,outweights,h))
```

Note that *(outys[o]*(1-outys[o])*(ds[o]-outys[o]))* multiplies output o_i by *(1 - o_i)*, which is equivalent to $\varphi'(v_i)$. The deltas were then used to calculate the weight updates.

Algorithm 2 Python code for updating the weights.

```
# 4. update weights and biases
for h in range(hids):
.....for o in range(outs):
.............change = N*outdeltas[o]*hidys[h] - M*(lastoutweights[o][h] - outweights[o][h])
.............temp = outweights[o][h]
.............outweights[o][h] = outweights[o][h] + change
.............lastoutweights[o][h] = temp
for o in range(outs):
......tempbias = outbias[o]
......changebias = N*outdeltas[o] - M*(lastoutbias[o]-outbias[o])
......outbias[o] = outbias[o] + changebias
......lastoutbias[o] = tempbias
for h in range(hids):
......for i in range(ins):
............change = N*hiddendeltas[h]*invs[i] - M*(lastinweights[h][i] - inweights[h][i])
............temp = inweights[h][i] inweights[h][i] = inweights[h][i] + change
............lastinweights[h][i] = temp
for h in range(hids):
......tempbias = inbias[h]
......changebias = N*hiddendeltas[h] - M*(lastinbias[h] - inbias[h])
......inbias[h] = inbias[h] + changebias
......lastinbias[h] = tempbias
```

The N term in the above algorithm is the learning rate, and the M term is for momentum, which adds to or detracts from the reaction depending on continuity of the update's direction through the statespace. The longer it updates in a direction, the faster it moves in that direction. I now turn to the results of Part A.

0.3 PART A: results

0.3.1 After one Epoch

An epoch is an entire cycle of through the training data, training on each input once per epoch. For this experiment, we were presented with a training data

set of 314 *(x,y)* inputs, each with a desired output of either 1 or 0. After one epoch, performed on the data in its given order, the following weights resulted:

	X1	X2
1	0.666	-1.4624
2	0.2029	1.3468
3	1.0125	-0.061
4	0.0268	-0.0247
5	0.2687	0.4567
6	0.0549	1.6977
7	1.6947	-0.3572
8	-1.0114	0.5102
9	-1.2785	1.3501
10	1.9632	-0.0968

Table 1: Input weights. From input X1,X2, to hidden 1..10.

-0.7503	1.3531	0.6005	-0.4242	-1.1033	-1.1052	-1.3527	-1.5798	-0.6223	1.9566

Table 2: Input bias.

0.6559	-0.5854	-0.0559	-0.0757	0.3242	1.0607	1.4526	0.3952	-0.1688	-1.3284

Table 3: Outweights.

-0.0917

Table 4: Bias from hidden layer to outnode.

Finally, after the first epoch, the average network error energy $E = 0.1231$. The goal for this dataset is to train the network until it achieves less than 0.001 average error energy.

0.3.2 Continued Training

Following the first epoch, I randomized the order in which the inputs were presented with the following code.

Algorithm 3 The name of the dataset is Cross_Data, so references to 'cross' modify this in the obvious ways. The object d is a dictionary, which associates a value with a key.

```
def randCross():
......newcross = []
......d = {}
......i = 0
......while i < 314:
............n = rand(0,313)
............d[n]=cross[i]
............i = i + 1
......for i in d.keys():
............newcross.append(d[i])
......return newcross
```

The network took 33 epochs to dip below 0.001, reaching 0.0008 for its average error energy, as illustrated in the graph below.

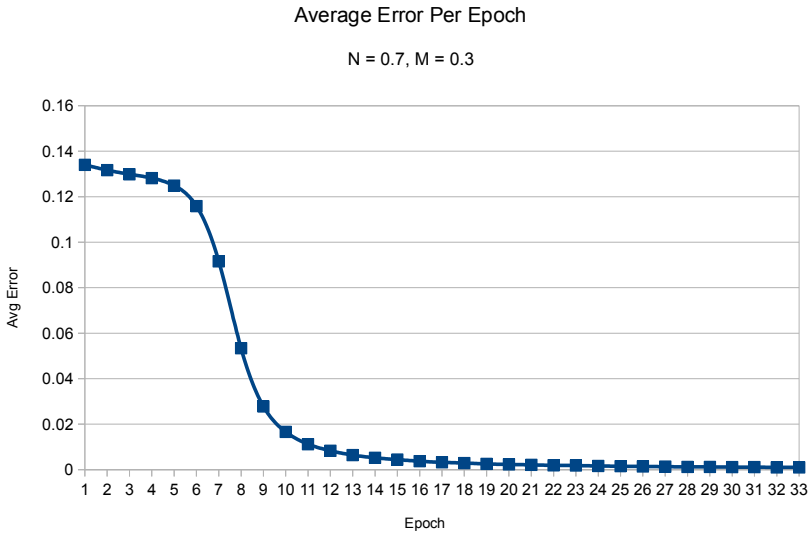

Figure 4: Average Error per Epoch, with learning rate N = 0.7, and momentum term M = 0.3

The datasets, when graphed, form a cross shape on the axes corresponding

to values with output 1, and values with output 0 are isolated to the outer corners of each quadrant, as seen below.

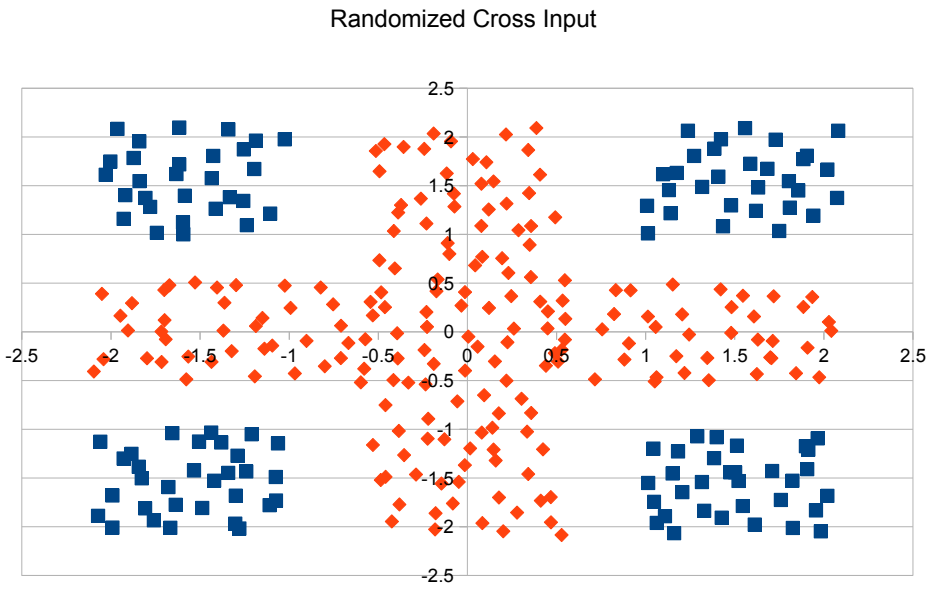

Figure 5: The network's classification of the data.

0.3.3 Retrain with Random Initial Weights; Alter N and M

The network proved to be extremely robust with respect to randomized input order. However, altering the learning rate N and the momentum term M led to some interesting variation. In what follows, I present the graphs corresponding to various N and M combinations.

The initial N value was 0.7, and the initial M was 0.3, and the graph for this combination can be seen above in Figure 4. To begin this tour, we will start with both values at 0.1. A small learning rate means that the update weight is only contributing a very small fraction of its value to the weight in the next iteration. Thus, we should expect to see a graph consistent with slow movement through the state space. Similarly, a small momentum term indicates that there is little benefit gained from continuity of motion through the state space.

In Figure 6 we see the network slowly improve from >0.12 for approximately 60 epochs, at which point it turns downward. However, its rapid decrease between epochs 60 and 90 declines, and it takes a rather long time to finally breach 0.001 after 270 epochs. Other than the puzzling improvement over those middle 30 epochs, the graph is entirely as we expected it to be. We can explain the rapid improvement by considering the nature of the state space. There is one global minimum, and initially the weights are not very near it. The network slowly searches through the space until it finds the slope, and the improvement yielded from traveling down the slope is great. However, the relative steepness of the decline, compared to the remainder of the curve, conceals what turns out to be a very slow approach to the minimum. Because the network took so long to find the target value, its distorted representation makes it appear to have a sharp improvement in the middle. However, compared to other values of N and M, this improvement is very slow, as we shall see.

Holding fixed N = 0.1, if we increase the momentum to M = 0.5, we see

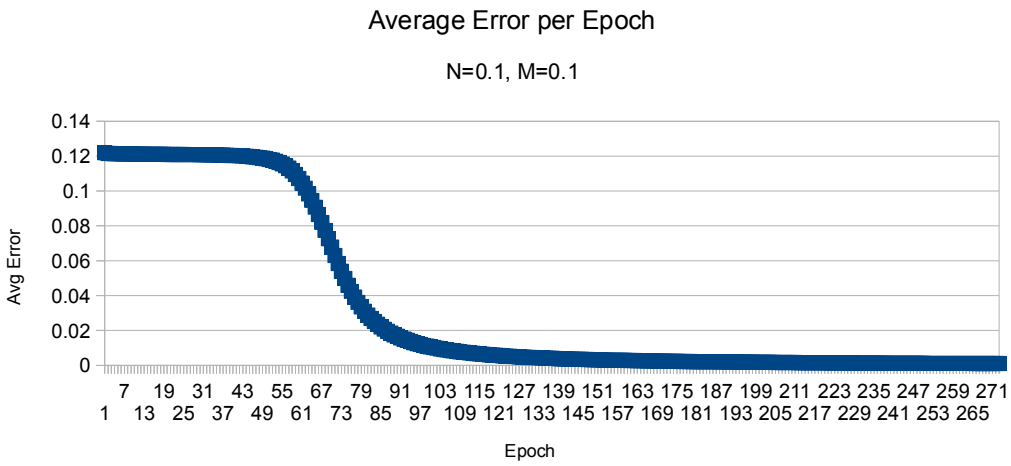

Figure 6: Average Error per Epoch, N=0.1, M=0.1.

considerable improvement. The network may still be a slow learner, but it gets faster as it maintains forward progress. Compare the length of its dramatic improvement in Figure 7 with that of Figure 6 above.

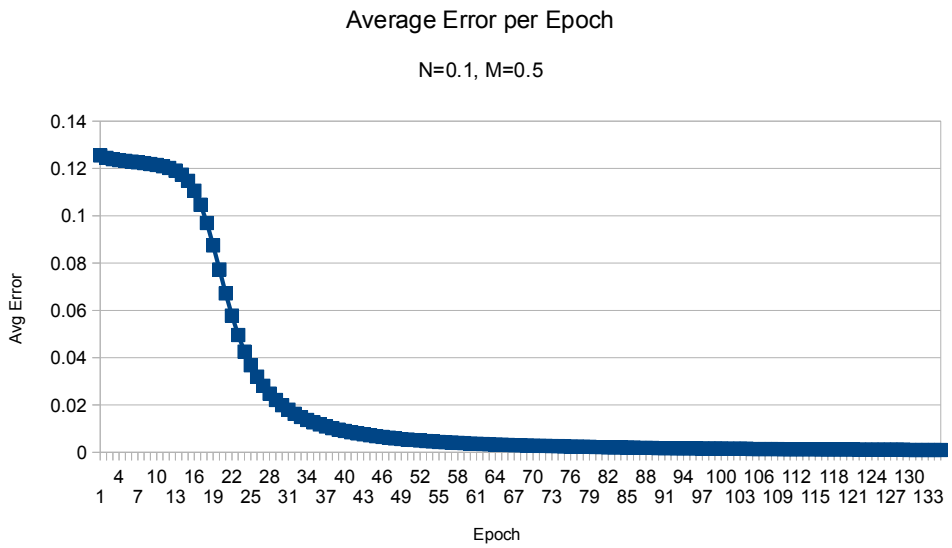

Figure 7: Average Error per Epoch with N=0.1, and M=0.5.

Interestingly, by increasing the momentum all the way up to M=0.9, we see tremendous improvement, but the same general shape of the curve. Specifically, we see a rather early dramatic improvement, followed by a leveling off, after which it breaks through the barrier and for a rather long time it slowly achieves its goal. The state space seems to have a shoulder surrounding the global minimum, which itself appears to be very thin. It seems to be thin because of the long time it takes to zero in on the goal. The network is probably bouncing over and around it like a coin at the end of its spin.

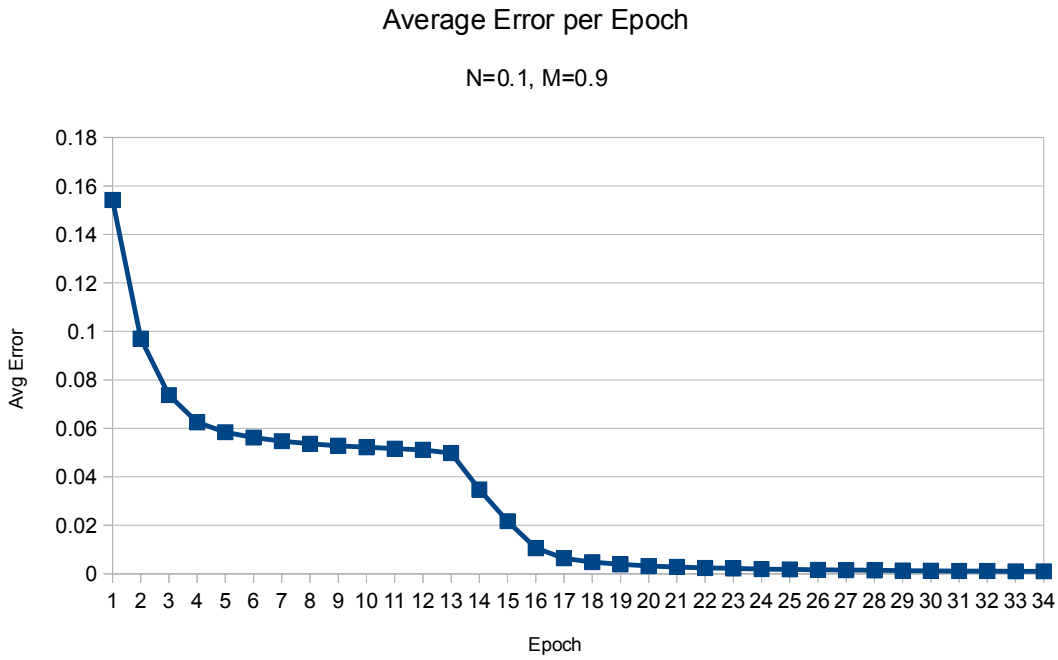

Figure 8: Average Error per Epoch with N=0.1, and M=0.9.

Something really remarkable happens when we increase N to N=0.3, and leave M at M=0.9. Things are balanced in just the right way as to swish smoothly into the minimum after only 10 epochs. We still see the pronounced tail, further supporting my hypothesis that the minimum itself is a very deep and thin sliver in the state space. However, representing this sort of improvement on a graph of a different dimension, like that of Figure 6, would cause it to appear virtually vertical, with no tail at all. However, we do still see a hint of a shoulder immediately prior to the tail.

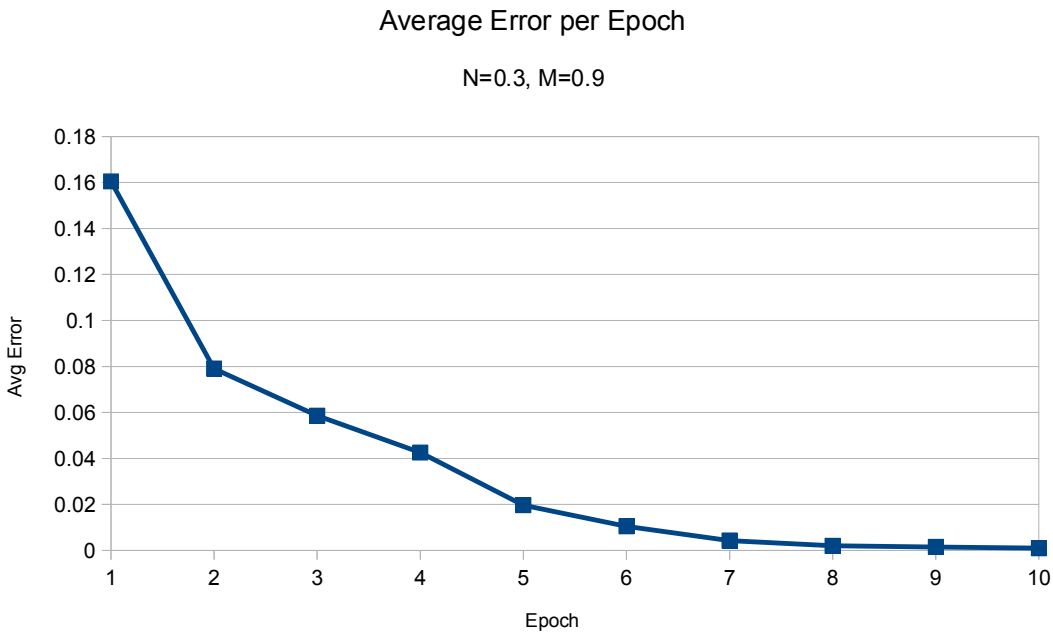

Figure 9: Average Error per Epoch with N=0.3 and M=0.9

We pause in the middle on our way to opposite N/M poles at N=0.4 and M=0.8, roughly the converse of the original assignment. We see that this yields an improved result over the N=0.7, M=0.3 result, though it is slightly worse than N=0.3, M=0.9. We see no shoulder in this curve, but rather a sharp initial improvement and a relatively long tail.

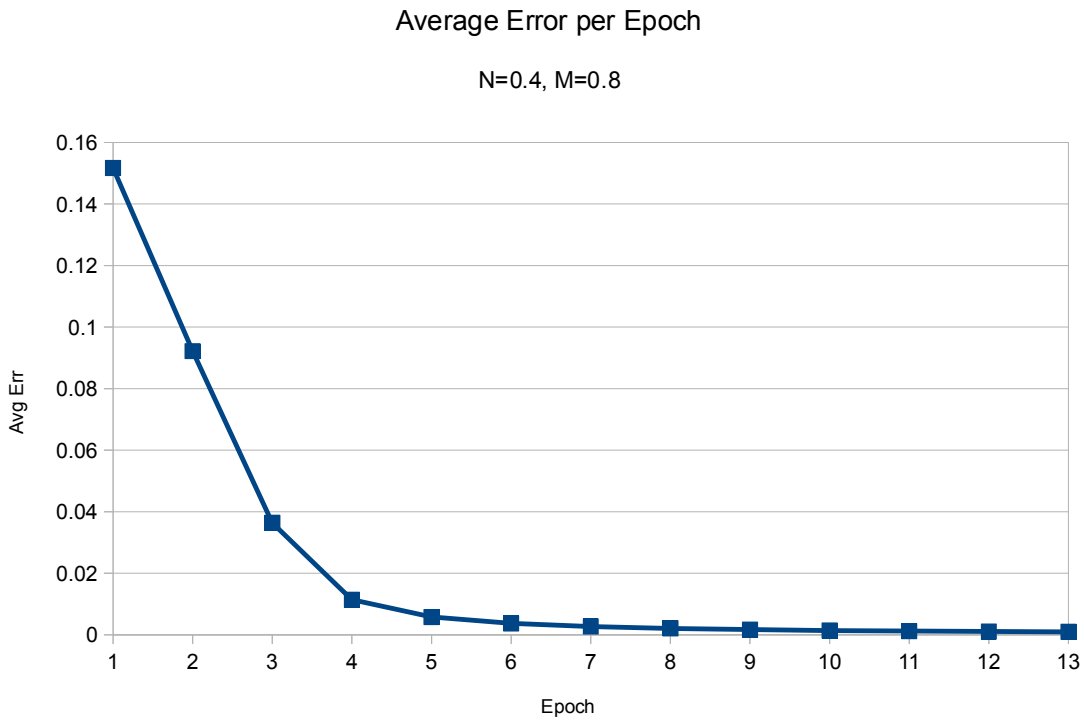

Figure 10: Average Error per Epoch with N=0.4, M=0.8.

The curve begins to regain its shoulder as it passes through the original 0.7/0.3 combination, and we examine it briefly at N=0.8, M=0.3, for confirmation.

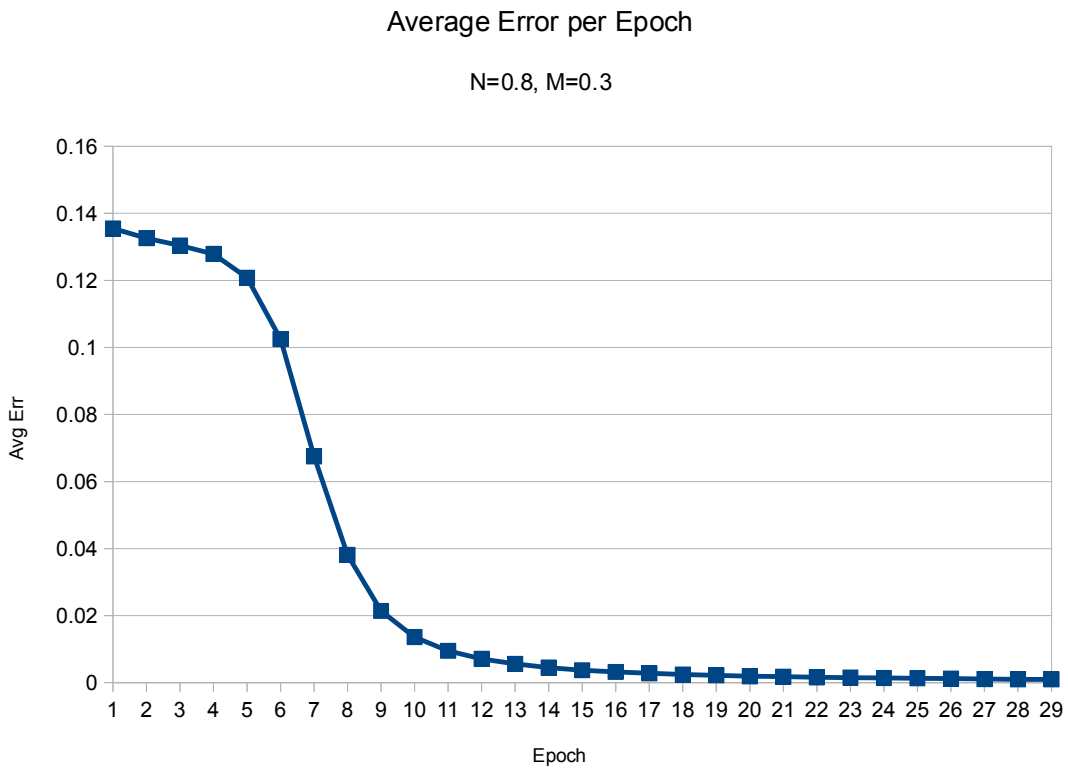

Figure 11: Average Error, N=0.8, M=0.3.

Not much changes as we hold N fixed and increase M. However, we reach the fastest success time at N=0.8, M=0.8, with a mere 7 epochs elapsing, followed immediately by the strangest thing ever at N=0.8, M=0.9.

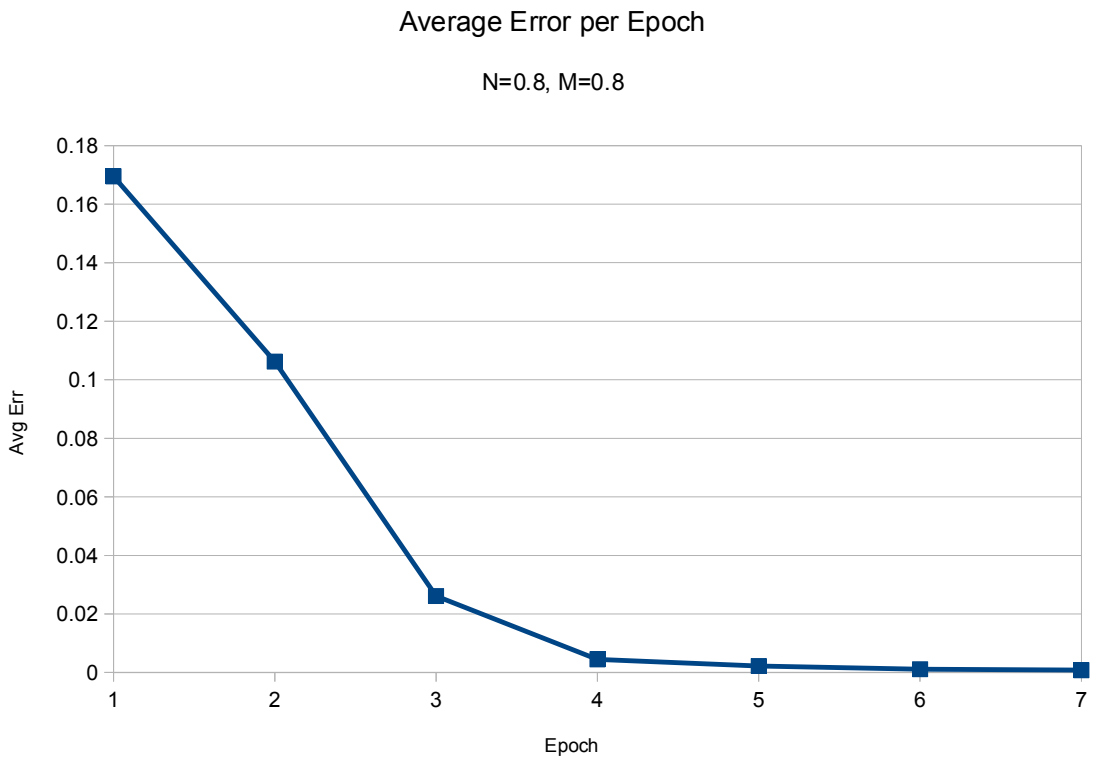

Figure 12: N=M=0.8

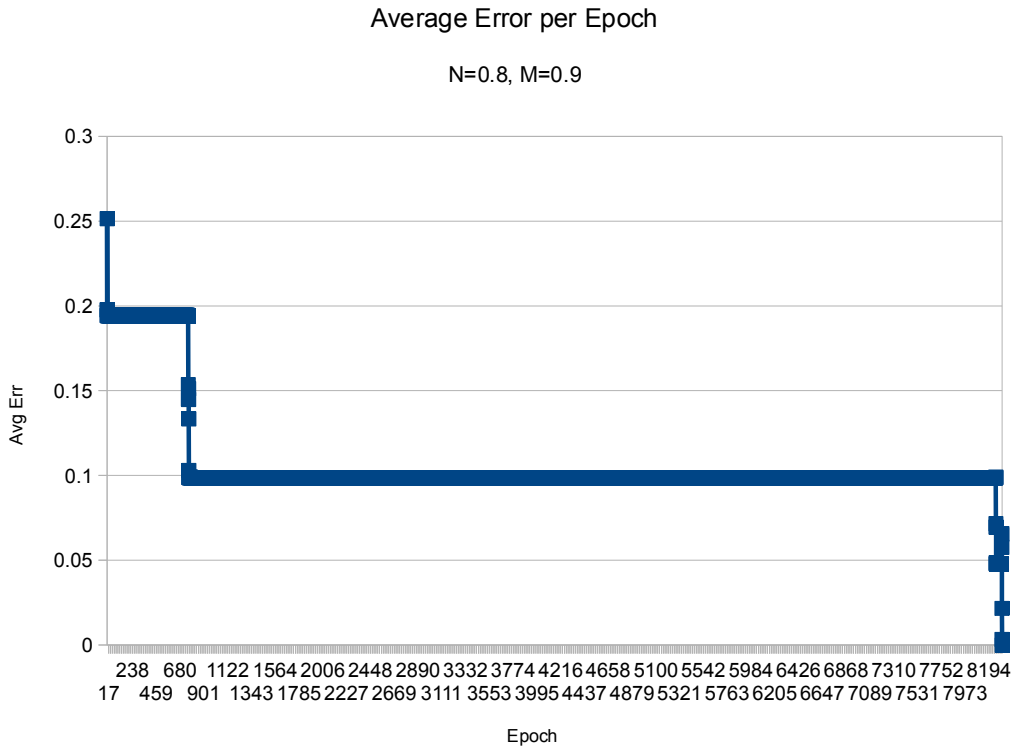

Figure 13: N=0.8, M=0.9.

By increasing M to 0.9, the network transitions from the optimal solution time to the worst, taking over 8000 epochs. I would like to further examine this graph in something closer to its full extent, because it appears to be a singularity or tipping point in the dataset and network complexity. It will likely grant good insight into the landscape of the state space. Even from the distance presented in Figure 13, we can see long periods of stagnation punctuated by relatively rapid periods of improvement. Although, it is hard to say how fast the improvement actually is, given the distortion. It is truly bizarre that such a difference can result from a mere 0.1 increase in momentum. I am tempted to infer that the difference lies solely in the momentum term, as the learning rate is constant, but I think it is likely that a more complex interaction is causing this.

This concludes my discussion of the network's application to the cross dataset. We established that the neural network does what it is supposed to do with competence, and that dramatic differences are possible with the slightest alteration of learning rate and momentum combination.

0.4 PART B: Geister Data

In this section we apply the neural network to a two player imperfect information board game. It is a useful way of testing the neural network's ability to process and adapt to rather complex input. In this experiment, we solve a decision problem involving the imperfect information component of the game. I will briefly take a moment to explain the relevant aspects of the game.

In Geister (German for "ghosts"), each player controls 8 ghosts on a 6 by 6 board. Each ghost has a type, either Good, or Evil, determined by a marking seen only by the controlling player. Two of the games objectives depend on this type distinction. One objective is to capture all four of the opponent's

Good ghosts. Another objective is to trick the other player into capturing all four of your Evil ghosts. If either objective is achieved, that player is the winner. Therefore, a good deal of strategy goes into trying to infer, based on its behavior, what type an opponent's ghost is. The data set we're presented with contains 17 input variables, corresponding to a variety of data points relevant to a ghost's type. For example, one of the inputs is the number of times the ghost has responded to a threat by capturing the threatening ghost, another is the number of times it's responded by fleeing, and another by remaning still. Each of these situations gives some information about the ghost's type.

The experiment proceeded in three parts. First, I created a Python script that imported and called my neural network function, using 17 inputs, 10 hidden, and 2 outputs, corresponding to the confidence that the ghost is Good and Bad, respectively. In total we had 800 inputs. I devoted 70% of this, 560, to the set of training data. In the second part, I collected 15%, 120, and devoted it to a validation set, to be fed forward on in parallel to the training. Thus, as the neural network trained, the validation data provides real time feedback for its performance on non-training data. Finally, part three tested the network weights on the remaining pristine 15% of the data set.

For an initial learning rate and momentum term, I chose 0.4 and 0.7 respectively. I made this decision based on the following. With such a large data set, I wanted the network to accelerate somewhat rapidly as it improved. However, being unsure of the landscape's terrain, I wanted to decrease the chances that it would bounce over the top of a thin global minimum. So, I gave it a somewhat small learning rate. I knew prior to the experiment that a consistent function would be very difficult to compute. My goal was to reach an average error energy of .10 in the training data.

The result of parts 1 and 2 are depicted below. We see the training error

sits slightly below the validation error, as we should expect, although the network did not improve very much. It did not appear to suffer from over-fitting, however, because the validation error did not show signs of increasing. On the first attempt, I achieved the goal of sub-10%.

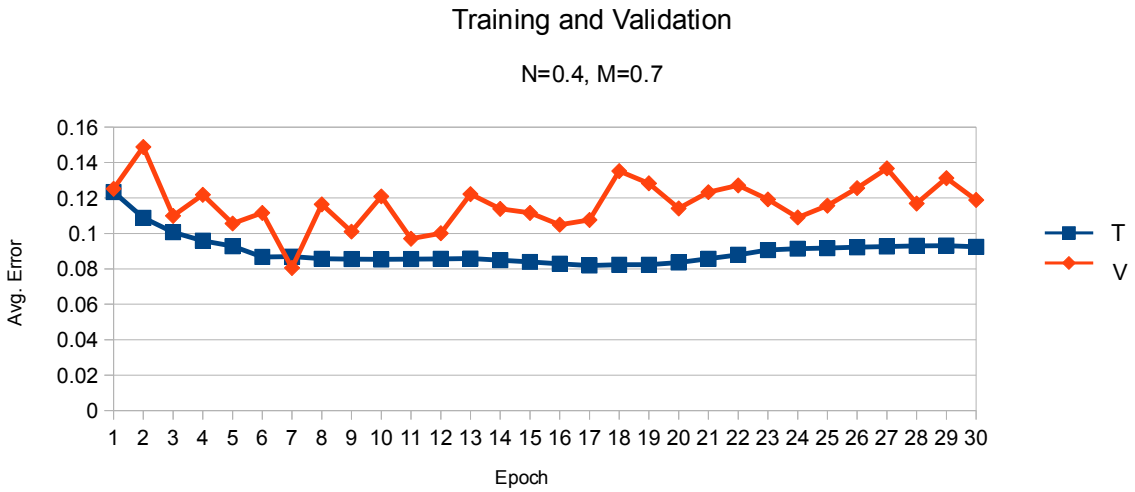

Figure 14: Training Error vs Validation Error, per Epoch.

Part 3 of the experiment ran entirely new data through the network, as well as the remainder of the data (without updating any weights), and the network achieved an average error energy of approximately 0.07. I was quite satisfied with this result, but attempted to improve it further by adding hidden nodes and adjusting the learning rate and momentum parameters. I was not able to achieve a better result, though I did not systematically explore all of the combinations.

As a final assessment of the network's ability to learn and infer, I present a confusion matrix, which tallies the network's true positive, true negative, false positive, and false negative rates. These correspond to correctly identifying a Good ghost, correctly identifying an Evil ghost, and incorrectly identifying each respectively. Below is the optimal confusion matrix I achieved, with 10 hidden nodes and N=0.4, M=0.7.

	True	False
Positive	247	78
Negative	317	114

Table 5: Confusion Matrix, N=0.4, M=0.7. A total of 613 correct, roughly 77%.

I am suspicious of these results. It puzzles me how the average error energy could be .07, and yet the network could infer correctly only 77% of the time. I suspect that I have made a coding error when tallying the results. For reference, here is the relevant code:

Algorithm 4 The Confusion Matrix calculation, perhaps with errors.

```
trueGood = []
trueBad = []
falseGood = []
falseBad = []
unsure = []
desiredouts = geisterdesired()
for k in range(len(geisterdata)):
.....if desiredouts[k][0] == 1: # column 18, if the ghost actually is good
.............if fulltest[0][k][0] >= 0.5 and fulltest[0][k][1] < 0.5: # Network id'd it
as good
.....................trueGood.append(fulltest[0][k])
.............elif fulltest[0][k][0] < 0.5 and fulltest[0][k][1] >= 0.5:
.....................falseBad.append(fulltest[0][k])
.............else: unsure.append(fulltest[0][k])
.....elif desiredouts[k][1] == 1: # column 19, if the ghost actually is bad
.............if (fulltest[0][k][1]) >= 0.5 and fulltest[0][k][0] < 0.5:
.....................trueBad.append(fulltest[0][k])
.............elif fulltest[0][k][1] < 0.5 and fulltest[0][k][0] >= 0.5:
.....................falseGood.append(fulltest[0][k])
.............else: unsure.append(fulltest[0][k])
```

0.5 Conclusion

In hindsight, my decision to implement the neural network and backpropagation algorithm in Python from scratch, rather than in a perhaps friendlier language like Matlab, was a mistake. However, the process of working through every minute detail certainly helped me learn more about both neural networks and computer programming, so in that respect the experiment was a success. As usual, time constraints prevented me from fully exploring all of the things I would have liked to explore in the data, and the network's capability. Should I find time in the future, I would like to further examine the cross data processed at N=0.8 and M=0.9. Though I did not include it in this report, I did re-randomize the input order for those network parameters, to see if perhaps it was a fluke in the data. The result was another very long solution, though not quite as long as the one included in this report. To me, this suggests that there

is a very interesting relationship between those parameters and that data set.