Final Write-up for a Simple Multi-layered Neural Network

Joe Brenner

- Course ID: CMPLXSYS 530,
- Course Title: Computer Modeling of Complex Systems
- Term: Winter, 2018

Problem Overview and Literature Review

For the past few years I have been very interested in Machine Learning and intelligent systems. One area that I especially wanted to learn more about was neural networks,I was curious about how they are implemented and what affects their performance. To study this, I created an agent-based implementation of a simple multi layered neural network using python's NetworkX library.

A properly implemented neural network can be "taught" how to recognize patterns and or classify data. This is done by training the network with specific training data using alternating cycles of forward propagation, in which data is fed into the network and is classified, and back propagation, in which the weights of neuron connections are updated based on the error between the expected output values and the actual outputs of the network from the forward propagation. After the training is done the network can be used to classify data in an identical format to the training data, even if it is presented with an unfamiliar combination of values. The accuracy and efficiency of this learning is the process that I was most interested in studying.

My main goals were to examine how the structure of the network and certain parameters, such as learning rate or the number of training passes, will affect this learning. Example of these effects could be undertraining, in which the network does not learn how to classify the training data let alone the testing data, or overtraining, which limits the networks in classifying any data that deviates slightly from the training data.

For this model, Agent Based modeling was very useful as it was just a very intuitive way of implementing the neural network. At its core, the neural network is just a dynamic network in which the values of each node depend on its neighbors' values and the weights of these connections are constantly changing. Each node follows the same pattern, the only difference being for determining the value of the input neurons and the error values of the output neurons. This meant that I could just treat each node as an agent and implement the necessary calculations at the simplest level instead of trying to write calculations to propagate the values at a higher level, which would have required a better understanding of the material than I had at the beginning of the project. There are more efficient and less computationally expensive ways of implementing a neural network than as an ABM, but they would not have been more difficult to implement given me the understanding that this model did.

For testing purposes, this model was applied to a classification task. The model was trained to classify the expected knowledge level of a a user in one of four categories - "Very Low" , "Low", "High", and "Very High" based on 5 percentage quantities for the degree of study time for the goal material, degree of repetition number of user for goal object materials, the degree of study time of user for related objects with goal object, exam performance of user for related objects with goal object, and the exam performance of user for goal objects. These were continuous values between 0 and 1. The data set contained 249 instance of training data and 146 instances of testing data and was downloaded from https://archive.ics.uci.edu/ml/datasets/User+Knowledge+Modeling# . In theory, this model would be able to be trained on any machine learning data set, and this one was chosen fairly arbitrarily.

While the inspiration for neural networks, actual neurons in a brain learning to recognize patterns, is a deeply researched and abstract field, the science of neural networks is actually fairly concrete. Because of this, it did not take too much research to translate a neural network into an agent-based model. Unlike the social dynamics that are often modeled by ABMs, this model required very little generalization and few assumptions in order to be viable. Due to this rather straightforward mapping from the real to the model, the literature I referred to for this model was mainly focused on the underlying mathematics and the interactions between the neurons. All the resources I used can be found in the References section.

Model Description

1. Agents

The agents for this model are the neurons or the nodes of the network. These neurons are arranged into equally sized groups called layers, and every neuron in layer *i* is connected to every neuron in layers *i* – 1 and *i* + 1. The layer that a neuron is in determines the type and behavior of the neuron.

First, input neurons are the neurons in the first layer of the network. The values of these neurons are used as the input to the network and are set by a data set specific function that parses the input data into a usable format. This function will be different for any data set. The last layer contains the output neurons. As the name implies, the output of these neurons is considered to be the output of the network. All the layers in between are considered to be "hidden" layers and contain the hidden neurons.

The first step to calculating the output value of each hidden and output node is finding the summed input to this neuron. This is done by taking the output of every neuron in the preceding layer and multiplying it by the weight of the connection between it and this neuron. These weighted outputs are summed together and are put through a sigmoid function, which is any bounded function that always has a positive derivative. The one chosen for this implementation is the logistic function, defined by the equation $S(x) = \frac{1}{1+e^{-x}}$. This maps all possible inputs within the range of 0 to 1, with negative inputs leading to values < .5, positive inputs giving values > .5, and an input of 0 gives a value = .5. One of the key features of this function is that its derivative is easily calculated for any of its output values with the formula d(output) = output*(1 – output). This value will be needed for the back propagation.

Additionally, each neuron has an error and a delta value used for adjusting connection weights. The error value is the total error for the network that the neuron contributes to. The delta value is how much the neuron contributes to that error. For the output neurons, the error and delta calculations are fairly simple in implementation, if not in theory. The error of an output node is equal to the difference of the actual output minus the ideal output. To find the nodes delta value, this error is then multiplied by the derivative of the sigmoid function at the neuron's output value.

The error for the hidden neurons is a bit more difficult. To find a hidden neuron's total contribution to the error, its contribution to every neuron in following layer must be summed up. The delta value of each subsequent neuron it contributes to is multiplied by this neurons output value and by the weight of the connection between these neurons. The sum of all of these calculations is the error value for that node. Then it is the same as for the output nodes, this error value is multiplied by the derivative of the output of this neuron.

Once all of the delta values have been calculated, they are used to update the weight of the connections between the neurons. This is done by the equation $weight_{ij} = weight_{ij} - learning_{rate} * delta_i * output_{neuron\ J}$ for the edge connecting neurons *i* and *j*. This equation is essentially calculating the contribution that this edge gives to the total error and adjusting the weight to minimize that error.

Agent Attributes

- *type* – the type of neuron as described above – input, hidden, or output
- *output* – the output value of the neuron, a number between 0 and 1
- *summed_input* – the total sum of all the weighted inputs to this neuron
- *error* – The total error of the network that this neuron contributed to
- *delta* – The measure of this neuron's contribution to that total error
- *connections* – maintained as edges in NetworkX undirected graph, each neuron has a list of the connections it shares with other neurons
- *connection weight* – The weight of the given connection. Tracked as the "*weight*" value for each edge. can be any positive or negative number

*Input neurons only keep track of their *output* value and *neuron* type

Agent Functions and Pseudo Code implementations

- *activate* – calculate this neuron's output

```
activate(neuron)
        neuron[summed_input] = sum_input(neuron)
        neuron[output] = sigmoid_function(summed_input)
```

- *sum_inputs* – sum the weighted inputs to this neuron
    - *neuron$_{i\,j}$ = neuron number i in layer j*
    - *edge$_{i\,k}$ connects neuron$_{i\,j}$ and neuron$_{k\,j-1}$*

```
sum_inputs(neuron$_{i\,j}$):
    for each neuron$_k$ in layer$_{j-1}$
        sum = sum + neuron$_{k\,j-1}$[output] * edge$_{i\,k}$[weight]
```

- *calc_error* – calculate this neuron's total contribution to the error

```
calc_error(neuron$_{i\,j}$):
    neuron$_{i\,j}$.[error] = 0
    if neuron$_{i\,j}$[type] is output
        neuron$_{i\,j}$[error] = neuron$_{i\,j}$[output] - idealOutput[node]
        neuron$_{i\,j}$[delta] = neuron$_{i\,j}$[error] * calc_deriv(neuron$_{i\,j}$[output])

    else if neuron$_{i\,j}$[type] is hidden
        for each neuron$_k$ in layer$_{j+1}$
            neuron$_{i\,j}$[error] += edge$_{i\,k}$[weight] * neuron$_{k\,j+1}$[delta]
            neuron$_{i\,j}$[delta] = neuron$_{i\,j}$[error]] * calc_deriv(neuron$_{i\,j}$[error])
```

- *update_weights* – update the weights of the input connections to this neuron

```
update_weights(neuron$_{i\,j}$):
    for each neuron$_k$ in layer$_{j-1}$
        edge$_{i\,k}$[weight] = edge$_{j\,k}$[weight] - learning_rate * neuron[delta] * neuron$_{k\,j}$[output]
```

2. Global Variable and Initialization

Global Parameters

- *training_data* – A dataFrame containing all instance of the training data
- *testing_data* – A dataFrame containing all instance of the testing data
- *num_layers* – The number of hidden layers in the model
- *num_in_layer* – The number of neurons in each hidden layer
- *numInputs* – The number of input neurons
- *numOutputs* – The number of output neurons
- *idealOutput* – A list used to hold the ideal output for the given input
- *learning_rate* – a scalar used to modify how drastically the weight of a connection can change in one turn

It is important to note what determines the layout of the network. Each hidden layer is identical in size and is determined by the user. On the other hand, the numbers of input neurons and output neurons are defined by the data set. The number of input neurons matches the number of input values that are being plugged into the network for the classification. The number of output neurons is also determined by the data set. So overall, the number and size of the hidden layers are variable within classification tasks, while the size of the input and output layers are chosen to correspond with the properties of the data set and would need to be adjusted with any change in data set.

However, while the input neurons are pretty much set in stone, the number of output neurons could still be changed within limits for certain data sets. For this model, there are 4 output neurons, one for each possible category. This makes it so that the ideal output is in the form of 3 outputs being 0 while one of them, whichever corresponds to the correct category, is 1 and the classification reported by the network is chosen to be whichever output neuron has the greatest value. This could have been done a few different ways, such as with only one neuron giving ranges of values for different categories (0 - .25 for “very low”) or with 2 neurons that act as binary indicators for category 1 – 4. I chose 4 as it seemed to be the most intuitive to interpret.

Initialization

The model is initialized based on the num_layers, numInputs, numOutputs, and num_in_layer parameters. It creates the specified number of nodes then iterates through and separates them into their respective layers by adding in the connections between the neurons. The weights of these connections are initially set to be random numbers chosen from a uniform distribution between -1 and 1. All other agent variables are initialized to 0 and are updated in the first tick. The training and testing data is loaded from an excel file, and the idealOutput is left empty and is updated at the beginning of each forward propagation tick.

Initialization Pseudo Code

```
init_network(net, num_layers, num_in_layer, numInputs, num_outputs):
    add nodes from 0 to numInputs with attributes…
        output = 0
        type = "input”
    add nodes from numInputs to num_layers * num_in_layer + numInputs with attributes…
        output = 0
        type = “hidden
        error = 0
        delta = 0
        summed_input = 0
    add nodes from num_layers * num_in_layer + numInputs to num_layers * num_in_layer + numInputs
                                                             + num_outputs with attributes…
        output = 0
        type = “output”
        error = 0
        delta = 0
        summed_input = 0

    add edges between neurons in adjacent layers with attributes…
        weight = random number in range (-1, 1)
```

3. Interaction Topology

As mentioned before, the neurons are grouped into layers. During each tick every neuron interacts directly with every neuron in adjacent rows. Neurons never interact with other neurons inside of their row, or with any neurons two or more layers away. We will consider layers closer to the input neurons to be preceding layers and those closer to the output neurons to be the subsequent layers.

During the forward propagation ticks, the neurons in each layer will receive the input from each neuron in the preceding layer, then send their value to the neurons in the subsequent layer. In this way, the information propagates forward in the network, from the input neurons to the output. During the backpropagation ticks, the neurons in each layer will receive data (the delta values) from the neurons in subsequent layers, then send their own data to the neurons in the preceding layer. For these ticks, the information propagates backwards from the output to the input.

Back propagation will not always occur. When the network is being tested, either on the training data or testing data, this step can be skipped to avoid any changes in the network that would affect the results. If back propagation does not occur, then information only flows forward, and the connection weights are never updated.

4. Model Schedule

Schedule per Epoch

1. Load first data instance
    - I. Set input Neurons
    - II. Parse expected category and set idealOutput
2. Forward Propagate
    - I. Iterate through the neurons in layer order starting at hidden layer one
        - i. Iterate through previous layer to calculate summed input
        - ii. Put summed input through sigmoid to find output
        - iii. Update asynchronously
    - II. Finish when last output neuron is updated
3. Calculate and record total error
    - I. Add to the total Epoch Error
4. Backwards Propagate (skip if testing)
    - I. Calculate error and delta values of output neurons
    - II. Iterate through hidden neurons in reverse order starting at the last hidden layer
        - i. Iterate through subsequent layers neurons to calculate error
        - ii. Use error to calculate delta
    - III. Update Connection weights
        - i. Iterate through all neurons
            1. Iterate through all neurons in the preceding layer and update the connection weight between them and the current neuron.

ii. This is effectively updating the connection weights synchronously

5. repeat steps 1 – 4 for every data instance in the epoch
6. Divide the sum of the errors by the number of epochs to get the average error per epoch

Model Schedule Pseudo Code

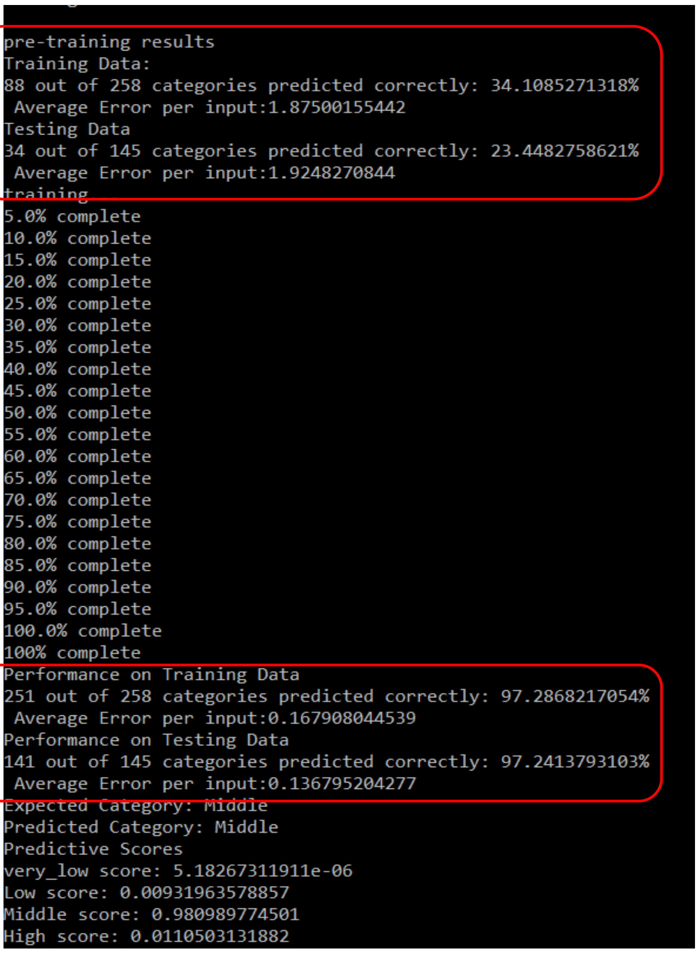
```
train_epoch:
    epoch_error = 0
    instances = 0
    for each data instance
        set_data_in_out
        forward_prop()
        back_prop()
        instances += 1
        epoch_error += get_output_error()
    return epoch_error/instances

set_data_in_out
    parse input values
    set input neurons output values

    parse expected output
    set corresponding node to 1, all others to 0

forward_prop
    for each neuron in order
        activate(neuron)

back_prop:
    for each neuron in reversed order
        calc_error(neuron)

    for each neuron in order
        update_weights(neuron)
```

Model Results and Analysis

To analyze the model, I ran parameter sweeps and a limited number of monte carlo simulations. I ran through the following set of parameters:

- Number of Hidden layers: 1 – 5 layers
- Number of neurons per hidden layer: 1 – 10 neurons in intervals of 2
- Learning rate: from: .1, .25, .5, .75, 1

I ran the model with each of these parameter sets for 1000 epochs each, recording the average error per training epoch after every epoch, and the average error per testing epoch every 10 epochs. This error was calculated as the sum of the absolute values of the differences between the ideal output and the actual. Therefore, maximum possible error was 4, and the minimum would be 0.



```
pre-training results
Training Data:
88 out of 258 categories predicted correctly: 34.1085271318%
 Average Error per input:1.87500155442
Testing Data
34 out of 145 categories predicted correctly: 23.4482758621%
 Average Error per input:1.9248270844
training
5.0% complete
10.0% complete
15.0% complete
20.0% complete
25.0% complete
30.0% complete
35.0% complete
40.0% complete
45.0% complete
50.0% complete
55.0% complete
60.0% complete
65.0% complete
70.0% complete
75.0% complete
80.0% complete
85.0% complete
90.0% complete
95.0% complete
100.0% complete
100% complete
Performance on Training Data
251 out of 258 categories predicted correctly: 97.2868217054%
 Average Error per input:0.167908044539
Performance on Testing Data
141 out of 145 categories predicted correctly: 97.2413793103%
 Average Error per input:0.136795204277
Expected Category: Middle
Predicted Category: Middle
Predictive Scores
very_low score: 5.18267311911e-06
Low score: 0.00931963578857
Middle score: 0.980989774501
High score: 0.0110503131882
```

Fig 1. Example model run.

Every network began with around an average error per epoch of approximately 2 and would predict anywhere between 8% to 40% correctly. This makes sense as it is classifying randomly before training. The best networks were able to get down to below a .2 error per epoch, leading to success rates of over 97% accurate predictions. This can be seen in the example model run in fig 1.

I used the average error per epoch to create time graphs to examine the behavior of the model over the its training cycle, mainly looking for the following 4 qualities:

- Minimum amount of error – What was the smallest average error the network achieved? An average error of .2 gives about a 90% prediction success rate, .1 yielded 97% correct. On the other end, an average error of 1.5 to 2 gives anywhere between 1% to 40% success, basically random

- Number of Epochs to reach minimum error – How many epochs did it take for the error to start dropping? Once it began to drop, how steep was that slope? When did it reach the minimum?
- Volatility of this error – Was the error line smooth, indicating a steady decrease, or volatile with many drastic increases and decreases, indicating the network and its performance is unstable?
- Training Error vs Testing Error– Ts the trend in error change similar for both data sets? Is the minimum error significantly different for the two?

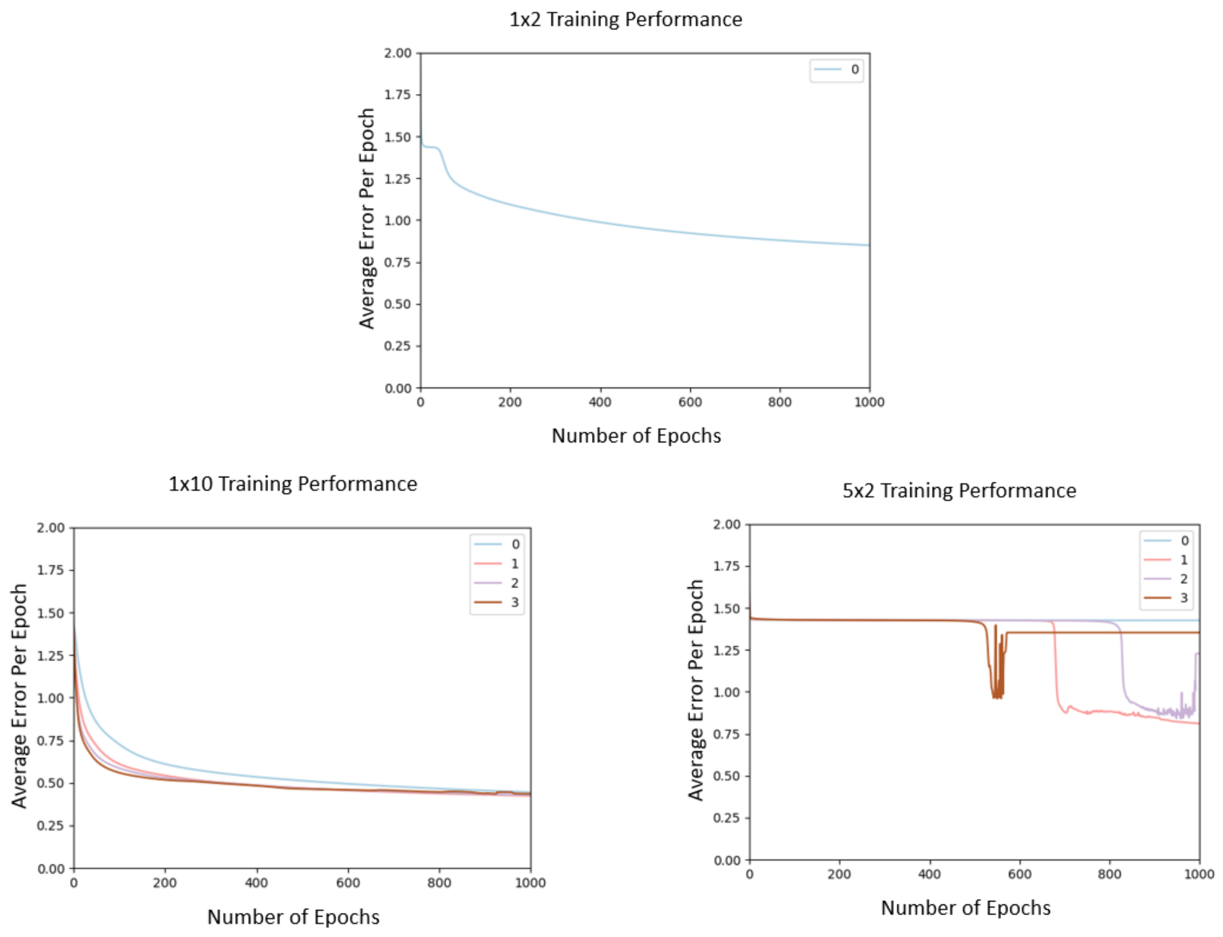


Fig 2. Comparison of three model configurations[1][2]

Overall, performance improved as the number of layers and the number of neurons per layer increased, Although the size of the layers seemed to be a more significant factor than the layers themselves. This is displayed very well in Fig 2., which shows the performance of three

[1] All of the graphs are labeled as MxN, where M is the number of layers and N is the number of neurons per layer.

[2] Graphs with the key of 0, 1, 2, 3 include the performance of that network size with learning rates of .25, .5, .75, and 1.

different size networks. With the minimum number of layers and maximum nodes, the 1x10 network, the final average error is approximately half that of the 1x1 network. When the number of layers is maximized and number of nodes minimized in the 5x2 network, the average error is only slightly lower than the baseline 1x2 network. This seems to suggest that it may be more effective in terms of performance costs and rewards to widen the network instead of deepening it.

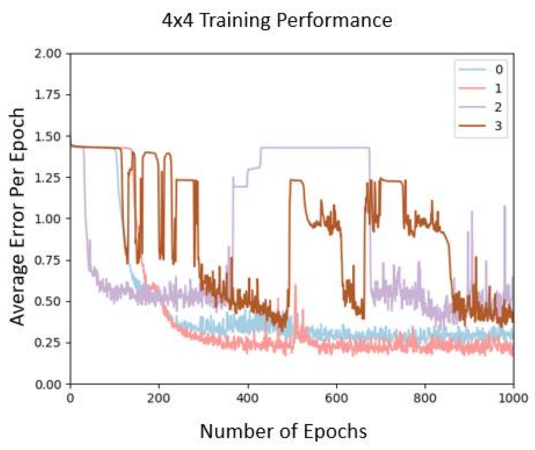


Fig 4. larger networks show more variability with increased learning rate

The learning rate offered some interesting insights as well. Higher learning rates did not necessarily lead to faster learning but almost always led to a higher variability in the results. The best performance seemed to consistently come from those networks trained at a .1 learning rate as they were the most consistent. This effect of learning rate became much more pronounced as the networks became larger. This initially surprised me but makes sense as small changes would naturally become more pronounced as they affect more and more connections.

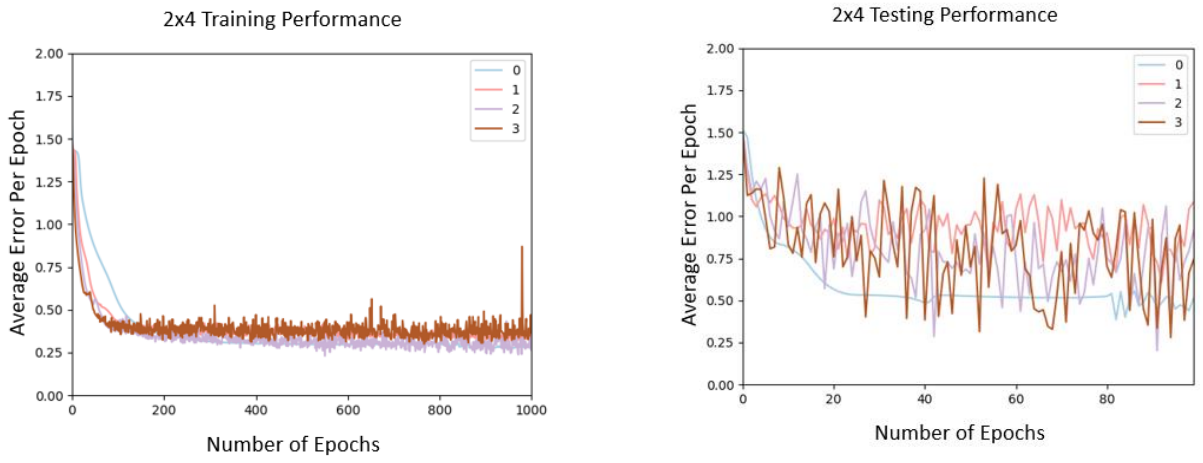


Fig 5. A large disparity between performance on training data and performance on testing data

Several configurations had significant different success between classifying the training data and the testing data. I was unable to pinpoint what qualities led to this, and they may just be anomaly runs, so monte carlo simulations would be needed to study this further.

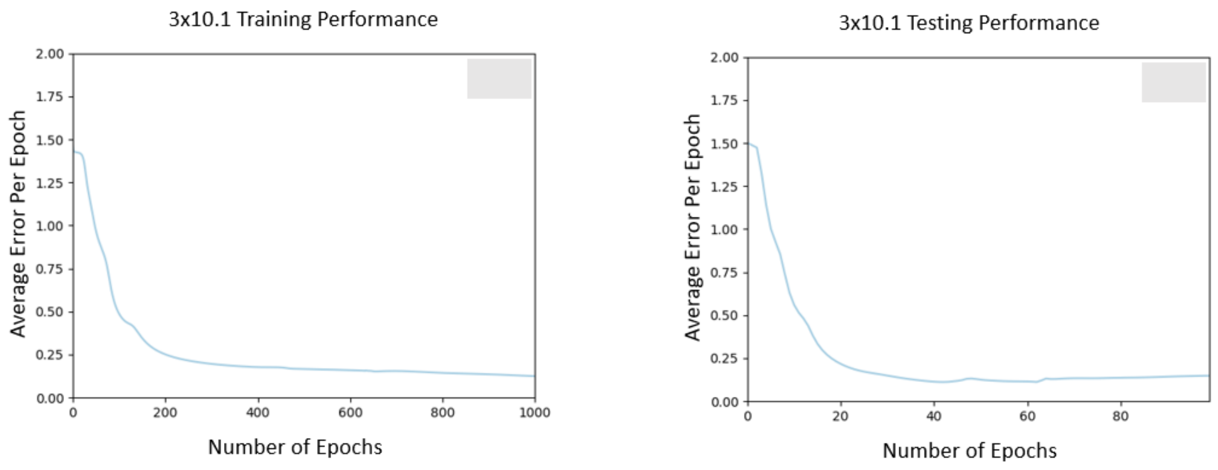


Fig 6. Single run of a 3x10 network with a .1 learning rate[3]

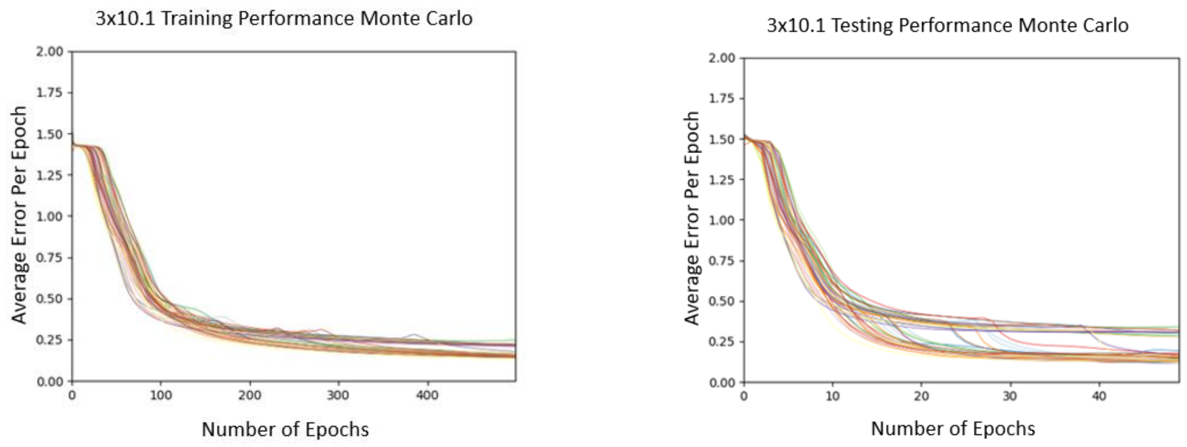


Fig 7. Monte Carlo simulation of a 3x10 network with a .1 learning rate

It seemed as though the network hit a sweet spot in performance, when considering the low error point and the speed of getting there. This came with 3 layers, 10 neurons per layer, and a learning rate of .1. Networks larger than this appeared to be too unstable and displayed too much variability to consistently achieve this level of performance. The performance of this network was confirmed with a monte carlo simulation checking both the training and testing performance, although in many simulations it seems that the testing performance did reach the same level of quality.

[3] The title 3x10.1 indicates a network with 3 layers, 10 neurons per layer, and a learning rate of .1

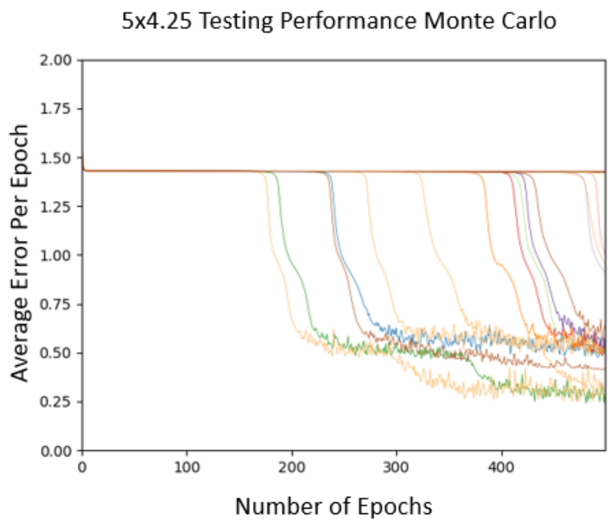


Fig 8. Monte Carlo Simulation of a 5x4 neural network with a .25 learning rate

One phenomena I had not expected was apparent performance plateaus of certain networks before they begin to learn, which can be seen if Fig 8. Especially in the larger networks, some would keep a steady average error of 2 for 100 – 300 epochs before dropping suddenly. I would guess that this is due to the random initialization of the network connections. If the initial weights are exceptionally far away from their ideal values, they might need to pass some threshold value to begin seeing performance benefits. This issue would be exacerbated in larger networks as the number of connections needed to pass this threshold increases with the number of nodes.

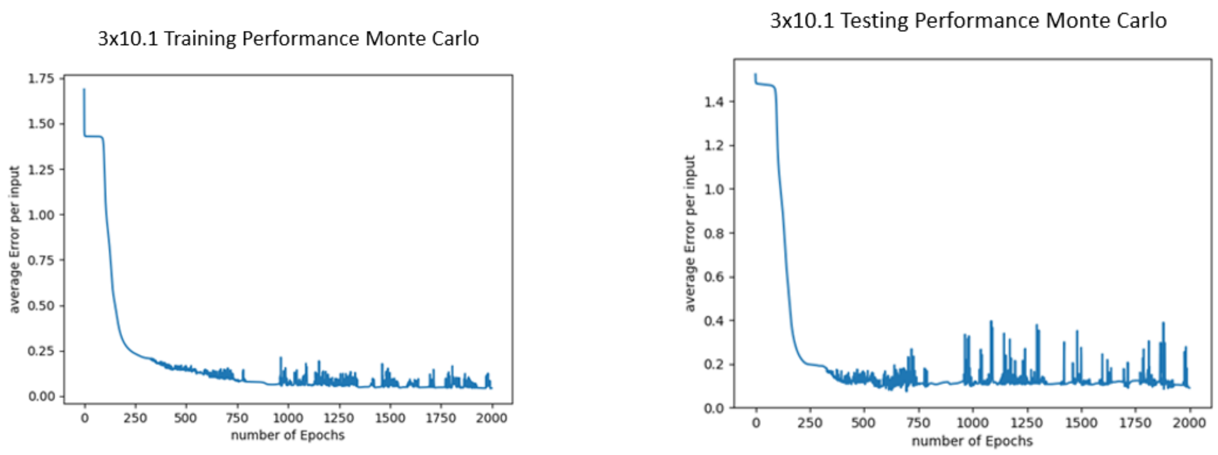


Fig 8. Comparison of Training and testing performance over 2000 Epochs

I saw slight evidence of overtraining in the 3x10 network, in which there were slight decreases in testing performance while training performance still seemed to be increasing. I extended this run to 2000 epochs to see if the pattern continued on, and it does seem to, though not very significantly.

Discussion and Future Work

Running the parameter sweeps was very enlightening as to what affects the performance of the network. Overall, I found that the networks with smaller learning rates typically performed best in most aspects. Furthermore, it appears that for this data set, the performance peaked in terms of stability, average error, number of epochs needed to learn, and in testing performance at a 3x10 network. This would indicate that there should be a peak performance configuration for any data set. I plan to do some further research to find out if there have been any studies on what determines this configuration for various tasks and data sets.

There are a few things that I would like to further analyze this model. First, I would like to run more Monte Carlo simulations to analyze certain behaviors of the network that were described above and see what the average behavior of different network configurations is. Also, I would like to gather more numerical data, including average error for testing and training sets and the number of epochs need to reach this error, and use it to analyze the performance of the model statistically.

The results of my current analysis have also shown a couple more avenues I would like to explore for this model. First, I want to test it on different data sets and classification tasks to see how its performance changes. I would also like to increase my parameter sweeps to include more nodes per layer, to see where the returns start to diminish with growing layer size. Finally, I want to try initializing the connections in different ways, either all to a certain value or following different random distributions, to see if this affects the initial plateau behavior of the larger networks.

This project has been very enlightening to me. From the implementation to the testing, I have learned a lot more about how Neural Networks work and what may affect their performance, accomplishing my original goal.

References

H. T. Kahraman, Sagiroglu, S., Colak, I., Developing intuitive knowledge classifier and modeling of users' domain dependent data in web, Knowledge Based Systems, vol. 37, pp. 283-295, 2013.

Mazur, “A Step by Step Backpropagation Example,” Matt Mazur, 21-Nov-2017. [Online]. Available: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. [Accessed: 24-Apr-2018].

“How to Implement the Backpropagation Algorithm From Scratch In Python,” Machine Learning Mastery, 22-Apr-2018. [Online]. Available: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/. [Accessed: 24-Apr-2018].

Nielsen and M. A., “Neural Networks and Deep Learning,” Neural networks and deep learning, 01-Jan-1970. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap2.html. [Accessed: 24-Apr-2018].