Carl Aiau – caiau - 6294888

Most of my research was with the Iris data set, as this was the only data set that we could split into training and test sample sets and test our generalisation ability. My initial explorations were testing the behaviour of various activation functions across all the COSC420 Sample data sets. The functions I choose to explore were Sigmoid, Relu (three variants), and Sine. After this initial investigation, I turned my attention to decreasing my population test error on the Iris sample set by extensive parameter tuning and recording how these changes resulted in different training performance. I will begin with a brief explanation of the program structure and then explore each of the research avenues and adventures.

When you run my program, you will be asked what folder you want to run the data out of. This allows you to store the separate models and their corresponding in.txt, out.txt and param.txt in folders relative to the program, rather than having to move the python script around. Optionally, you can leave this blank which will run the program based on the current folders contents. After you choose the folder (or not) to run from, you choose the activation function for the hidden layer, and the output layer. After this, you also tell the program whether this model needs to learn generalisation (which is applicable only for Iris). When generalisation is enabled, the logic is slightly different for testing, because it will be tested against the test data set only. The logic is also different for bulk learning, as a new training / test split needs to be performed on each iteration. Once the program is initiated up, you are presented with a rudimentary user interface where you can choose several actions. The bulk action is the only action that requires further user input. It will ask you the number of iterations you want to perform, and the number of epochs per each iteration.

My first explorations were around testing out different combinations of activation functions. For each model I used the default parameters that were provided on the COSC420 website and started playing around with various combinations of Sigmoid and Relu and Sine. I tested all combinations of functions, and quickly found that Relu will not work as an output function. In hindsight, the reasons for this are obvious, but I did get some a-ha moments going on my learning journey of turning theory into practise. I recorded the results of this early testing in the tables on the following page. The output function was always either Sigmoid or Sine and models that didn't learn at all have been left out. The first testings were done on Sigmoid which learnt every model quite well considering no changes to the learning parameters. I then began testing with the Relu function, Relu is all the rage in deep learning, but I assumed it wouldn't be very suitable for our shallow network and learning tasks. Relu itself performed quite bad, but I also tried a small alpha value (which I refer to as leakiness) of 0.01, and a higher alpha value of 0.1. These numbers were chosen ad-hoc, 0.01 seems to be an industry standard and was a shoot in the dark. I explored various alpha leakiness values later, which will be discussed further on. I also wanted to try something a bit unconventional, and use a periodic function. Periodic functions in theory should be terrible learners because of their oscillation, but I thought we might be able to get some performance out of it on simple tasks.

Sigmoid Hidden	Sigmoid Output		Sine Output	
	Solved	Epochs	Solved	Epochs
1_random	1000	75	125	23
2_xor	821	717	686	248
3_parity	840	1825	242	391
4_parity	130	2386		
5_encoder	1000	581		
6_iris	28	224		

Sine Hidden	Sigmoid Output		Sine Output	
	Solved	Epochs	Solved	Epochs
1_random	1000	40	162	52
2_xor	691	248	945	140
3_parity	553	293	630	185

Relu Hidden	Sigmoid Output		Sine Output	
	Solved	Epochs	Solved	Epochs
1_random	926	65		
2_xor			82	187
3_parity			10	351
4_parity			924	413
5_encoder				

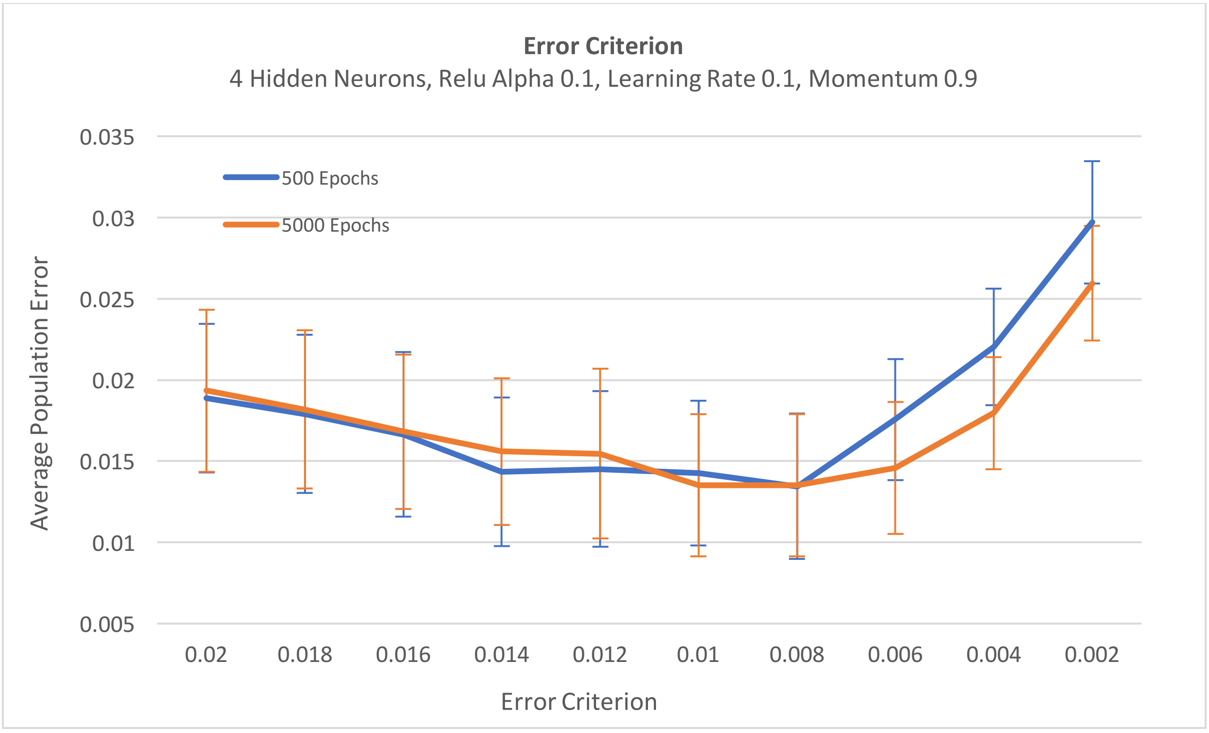
Leaky Hidden	Sigmoid Output		Sine Output	
	Solved	Epochs	Solved	Epochs
1_random	100	50		
2_xor			94	163
3_parity			11	233
4_parity			969	413
6_iris	391	152		

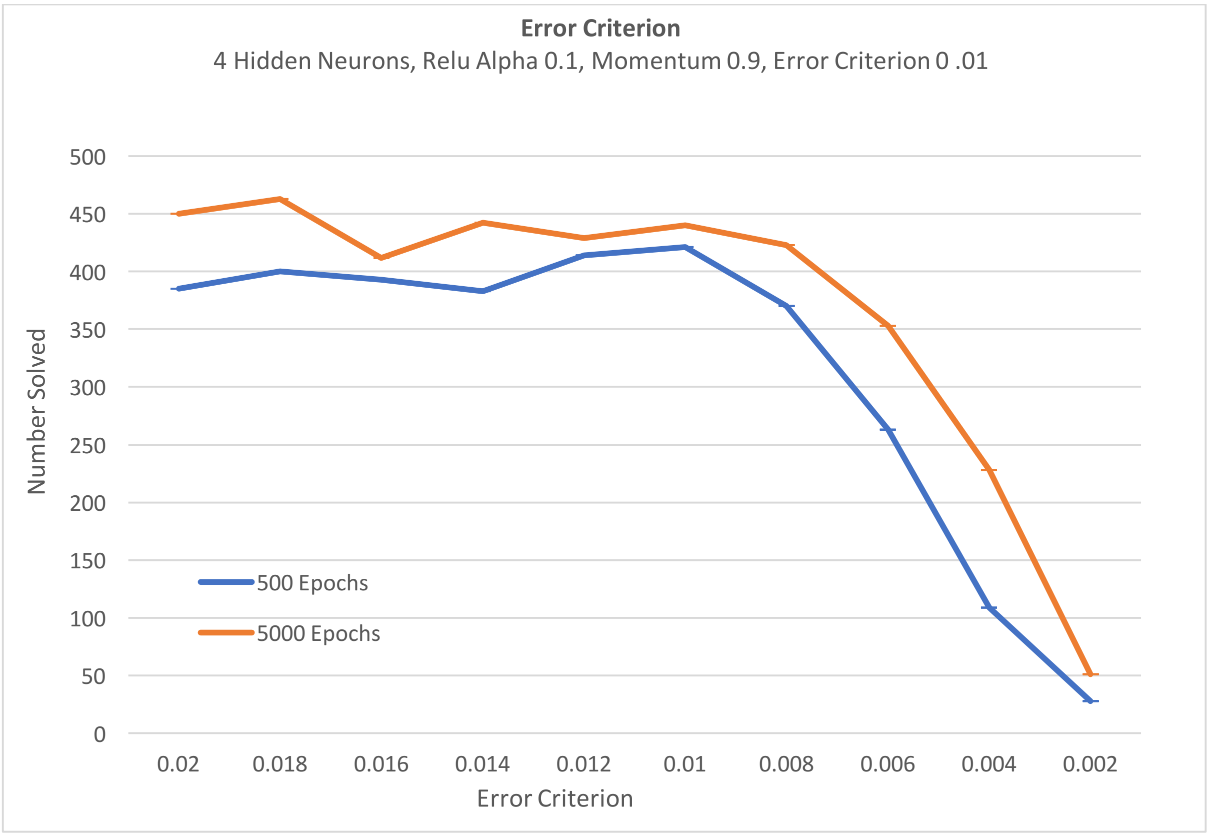
Very Leaky	Sigmoid Output		Sine Output	
Hidden	Solved	Epochs	Solved	Epochs
1_random	1000	35	1000	35
3_parity			17	244
4_parity			994	413
5_encoder	959	364		
6_iris	519	45		

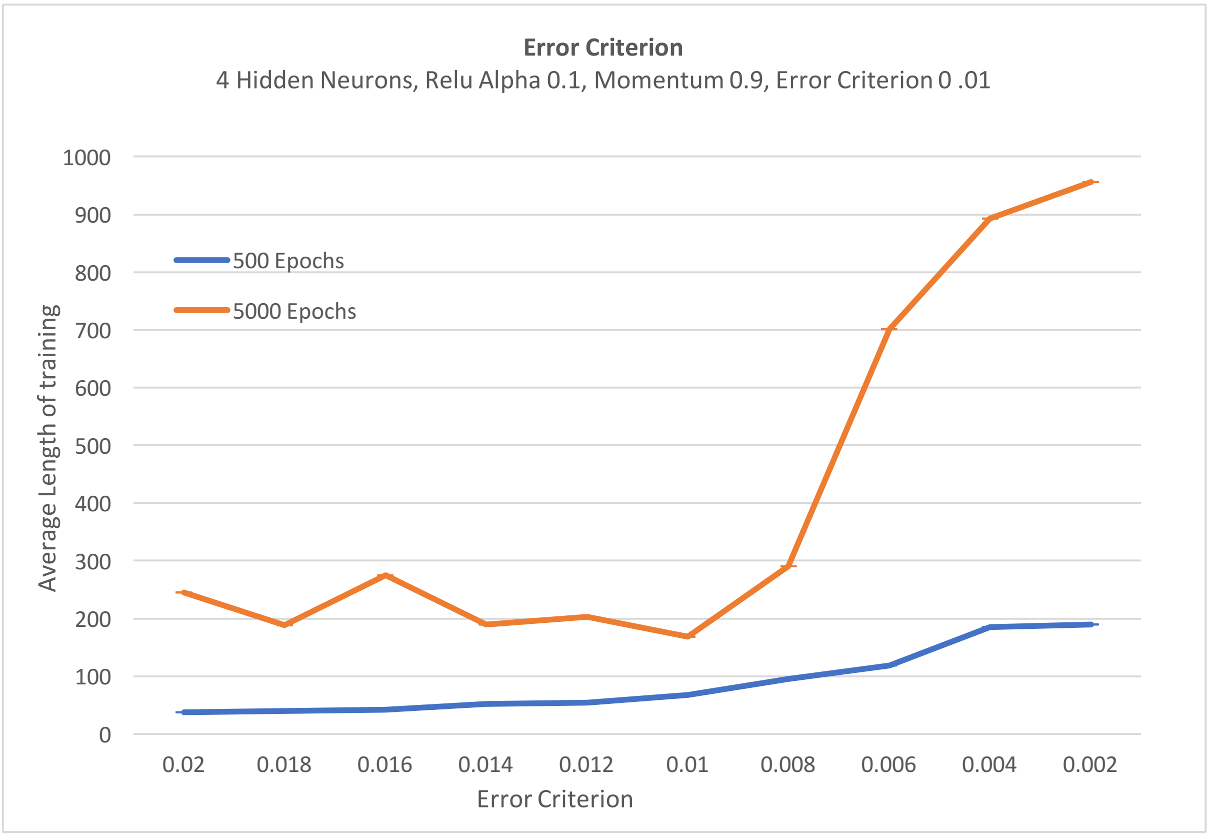
These results were created from a one layer neural network as specified in the param.txt files. They were run for 1000 iterations of 5000 epochs each, with the default learning rate, momentum rate, and error criterion.

Sigmoid Sigmoid was the only combination that solved everything, it's Iris success rate was only around 3% but the naïve implementation with no parameter tuning still impressed me. Sine didn't perform very well with sigmoid, apart from the Sine Sine combination which was good at solving XOR with 95% success and much quicker than sigmoid sigmoid. Relu Sine performed well at solving one form of parity. I haven't figured out why it would perform good on this parity, but worse on the the simpler form. I'm guessing the combinations performance is more related to what is getting spat out of the Relu function, and the Sine function can handle the wider range in a more trainable way. The fact that the Relu Sine output for 4_parity was almost identical regardless of the Relu alpha lead me to this conclusion. The most interesting takeaway from this first testing, was the 52% success rate (non-generalised) and ridiculously fast learning that the Relu (0.1) and Sigmoid produced. This would be the model that I based a lot of my further explorations on. Although my research was concentrated on the generalisation capabilities of the Relu Sigmoid combination, I would be interested to come back to see if periodic functions have any place in neural networks whether that is for better performance or been able to handle some forms of learning better.

After these initial function explorations, I turned my attention to the parameters. My main goal was to see how low we could get the average test population error down, and to see if it was possible to reduce the standard deviation of the sample. The small sample size of 150 did cause a significant amount of variability in the results but there are still obvious trends that I will discuss. All the parameter testing was done for 1000 iterations, and a run was done with 500 epochs and another with 5000 epochs. I was testing both numbers of epochs to see the effects that over training would have, and whether the increased practical cost (time / computation) had a benefit or a negative effect on the generalisation capabilities of the resulting models. Error bars on all the average population error graphs are 1 standard deviation. The initial error criterion was set to 0.02. I thought the Iris model should be able to perform much better than this and we were cutting the learning process short.

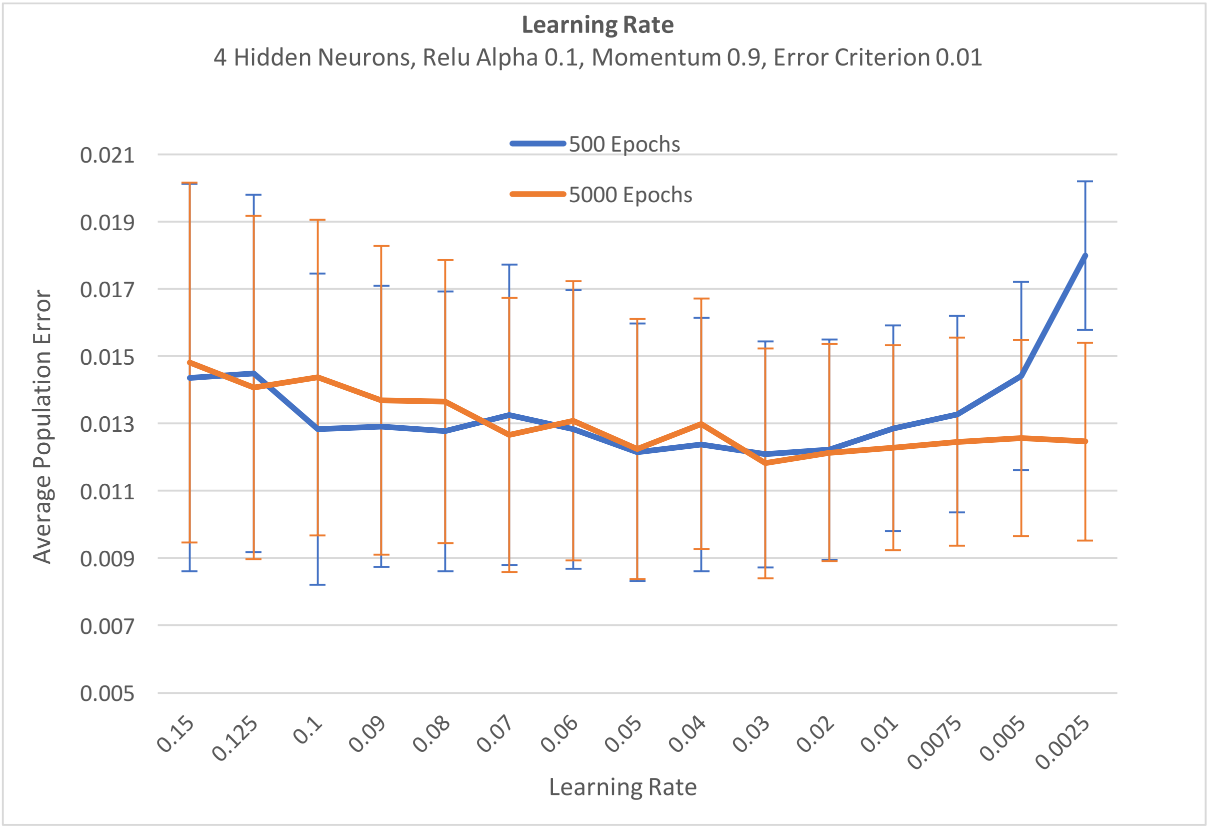


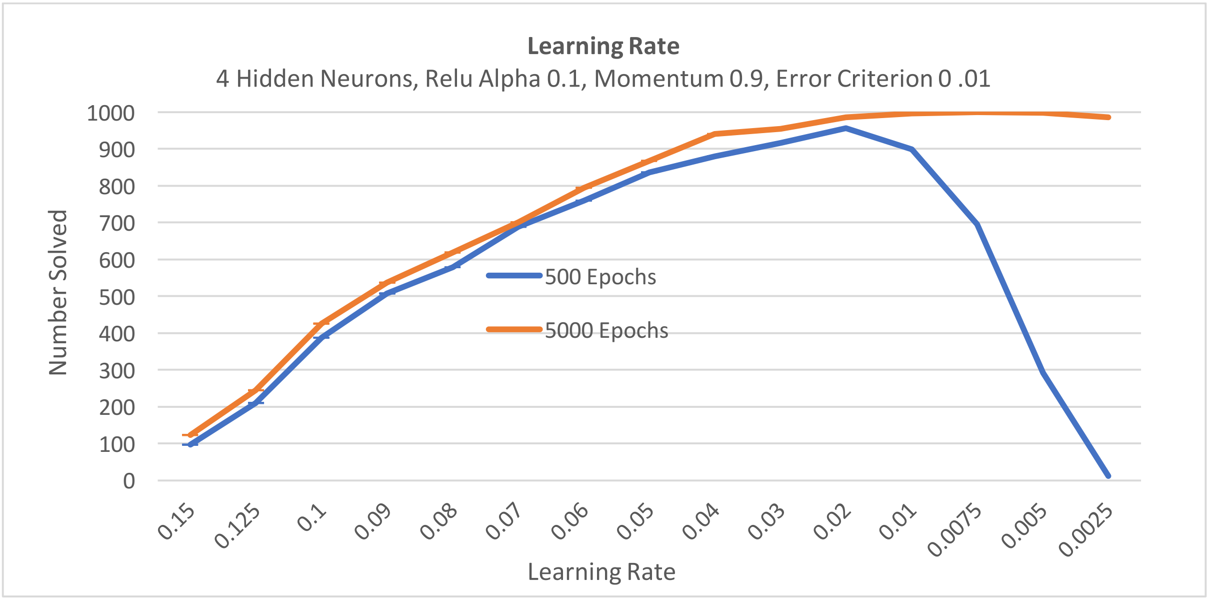


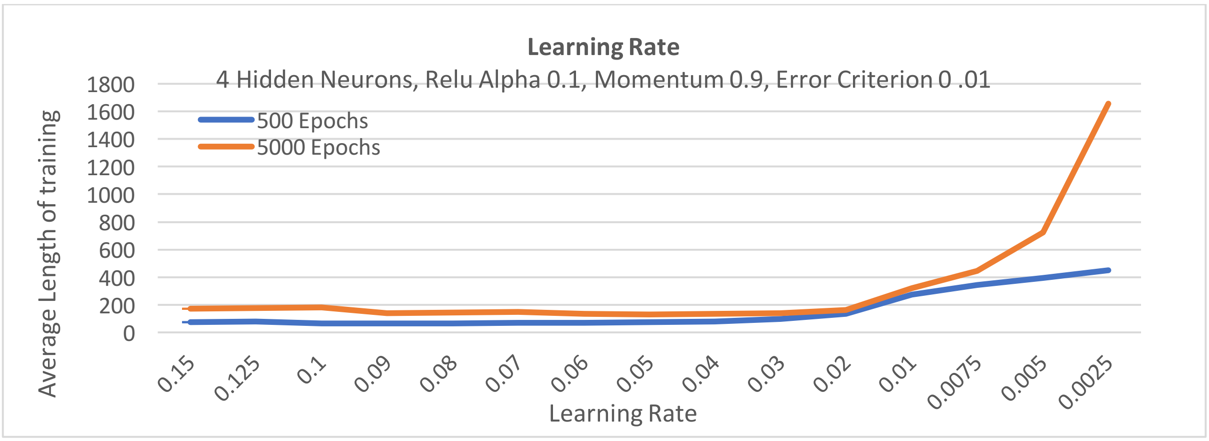


The optimal average test error was produced at 0.008 error criterion. There was no benefit in overall average test error from training for longer epochs. 5000 epochs resulted in 423 models reaching solution, but this increased the population test error from 0.01345 to 0.01352. As you extend the epochs your length of successful training increases, because there is more time given to models that started badly to descend and catch up with the early successes. The standard deviation is high at 0.00892, so we can't be too confident in the number, but 0.08 – 0.12 resulted in much better results than 0.2 or 0.04. We see substantial overtraining once the error criterion goes below 0.08, and there is a rapid rise in test error, and decrease in successful models.

The next parameter to tune was the learning rate, which also had an impact on model success.

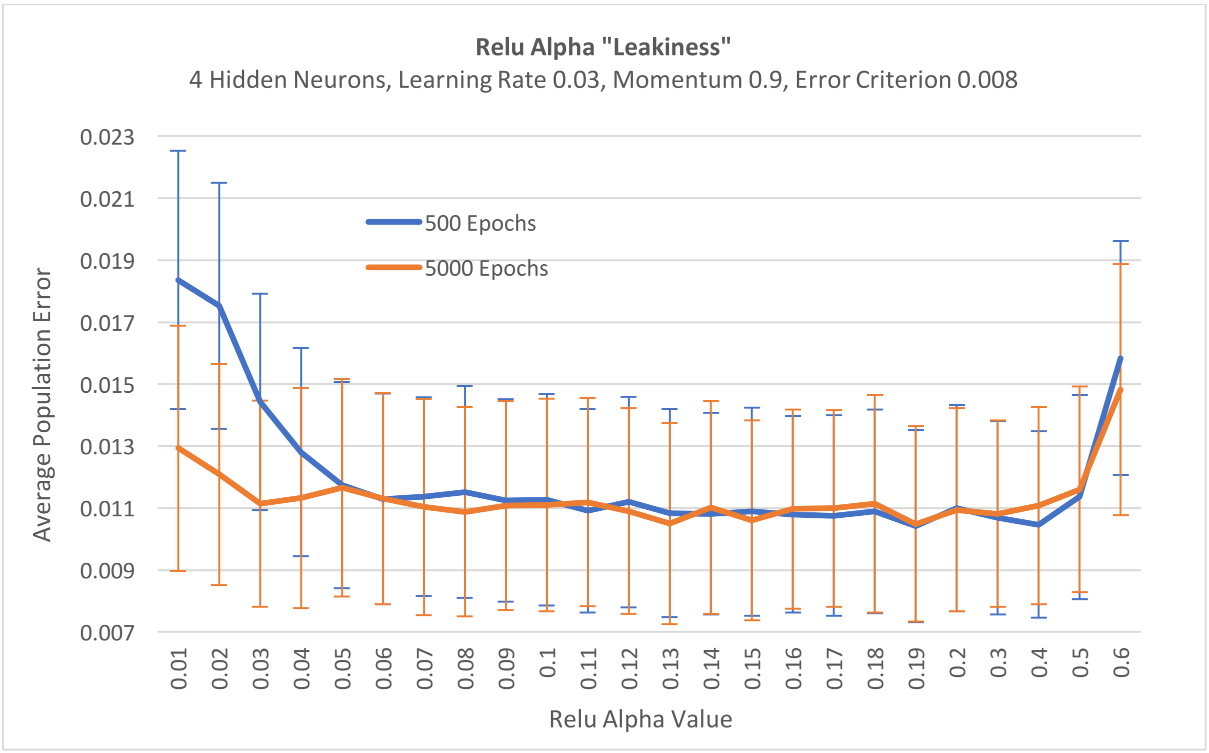


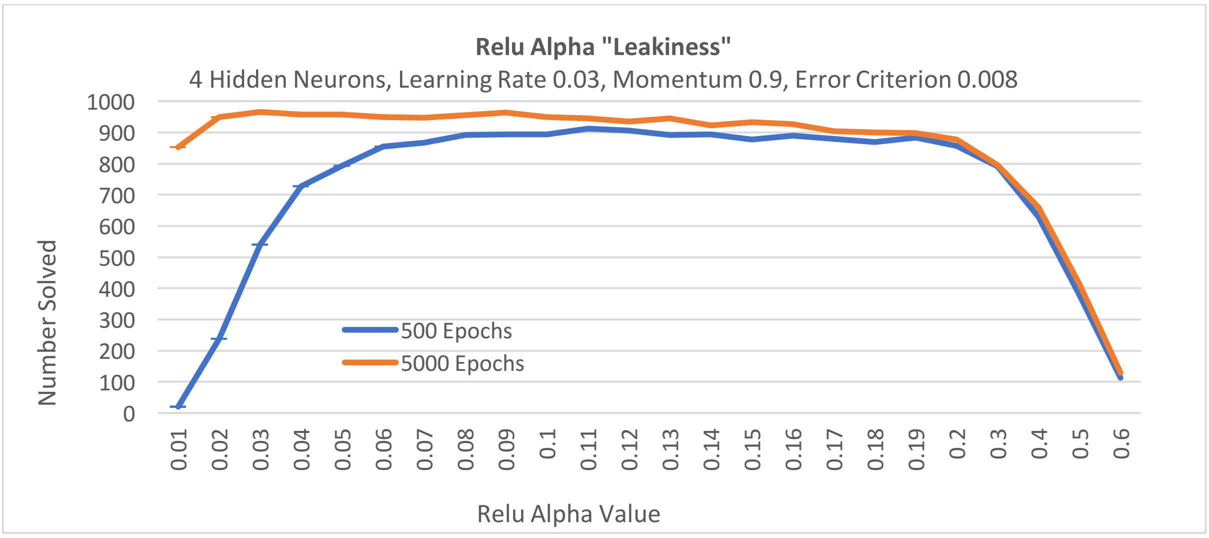


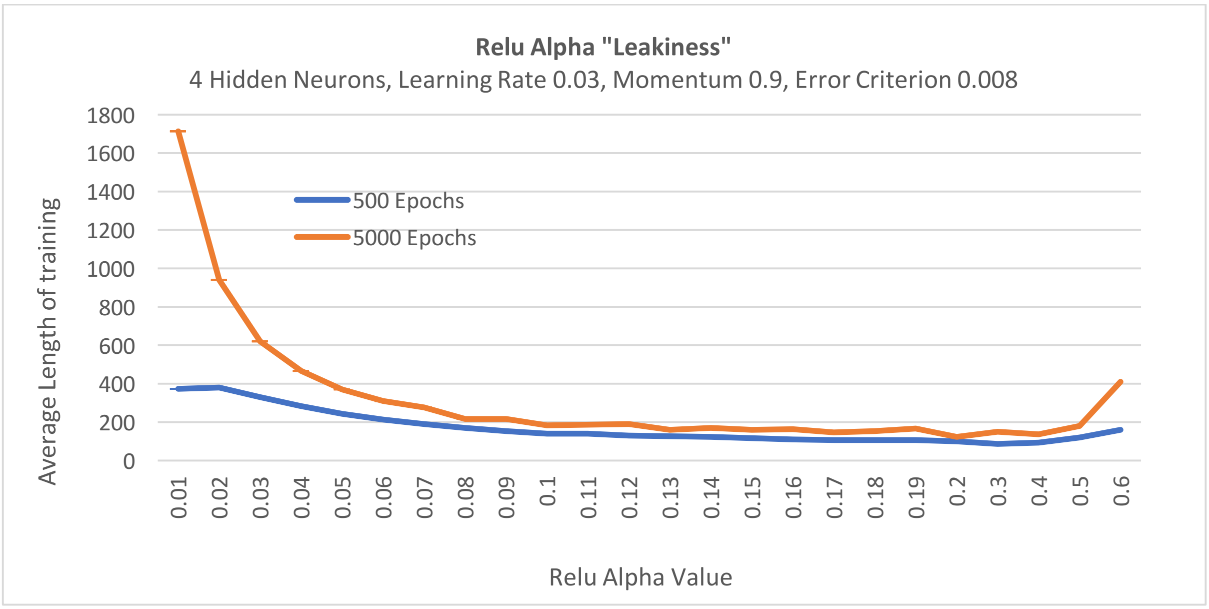


The best learning rate from this sampling is around 0.03, which is reasonably slow, but it results in a test population error of 0.011814, and 955 models successfully reached the error criterion of 0.01 (I had not switched the error criterion to 0.008 yet). Training the model with 5000 epochs versus 500 epochs did result in more models finding their way to global minima, and reduced the average test error. We can see the higher test error at 0.15 and 0.25 due to under and over training the model. Please note this graph isn't a linear scale. Between 0.1 and 0.01 we have incremental 0.01 steps, but outside of this range had to increase and decrease more quickly for the sake of time. After both tests had been completed I began using the new learning rate of 0.03 and error criterion of 0.008 for future testing.

Next up I wanted to test the leakiness of our Relu function, and what effects this had on both learning performance and time. I found the results from the leakiness experimentation interesting. The main



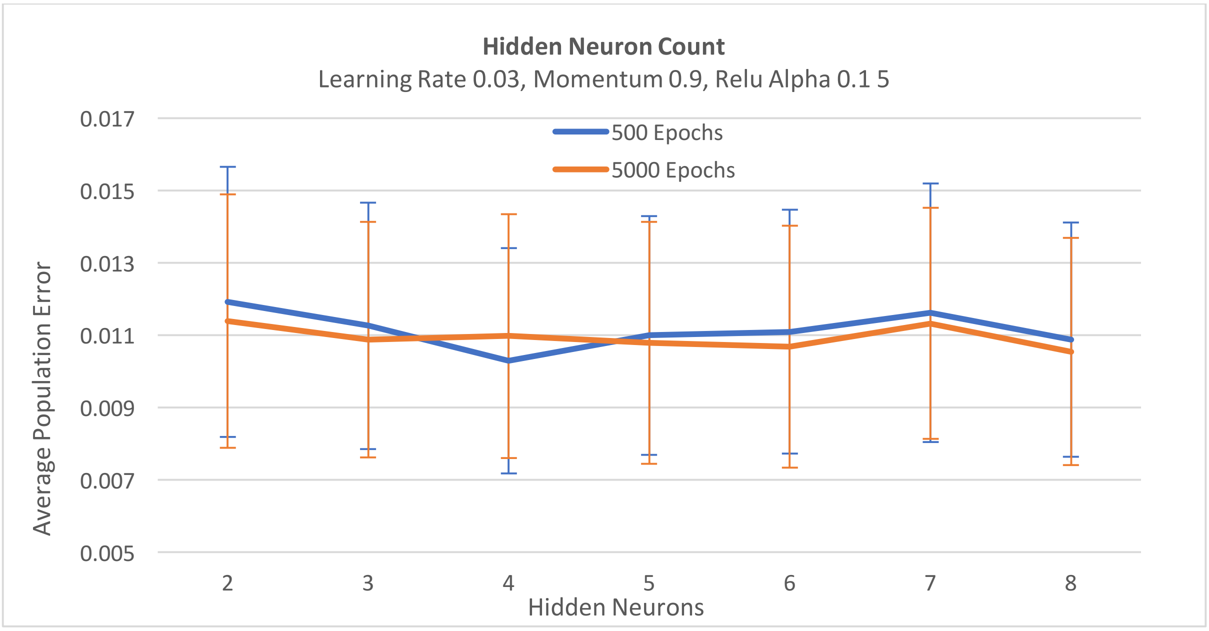


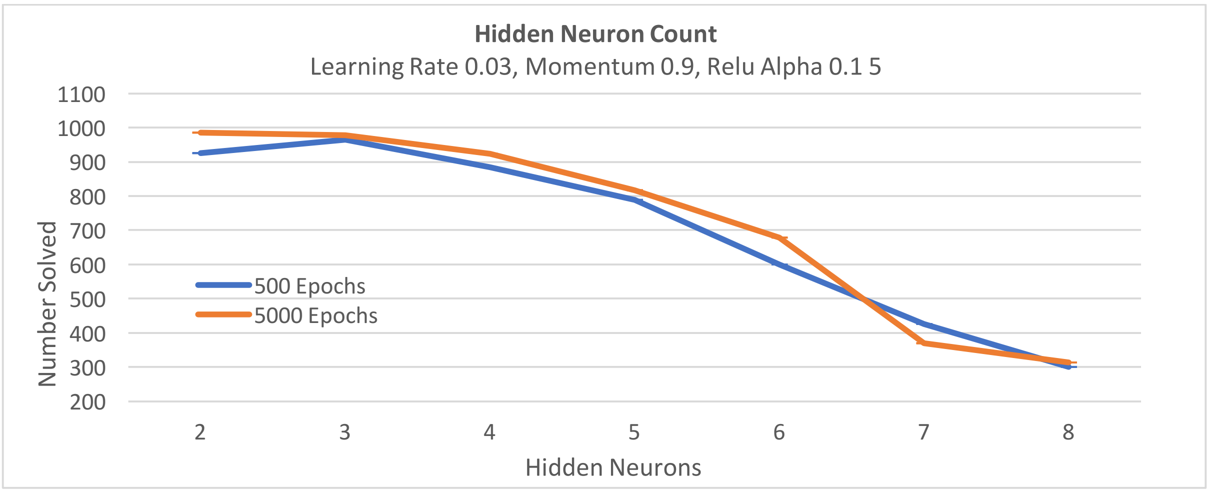


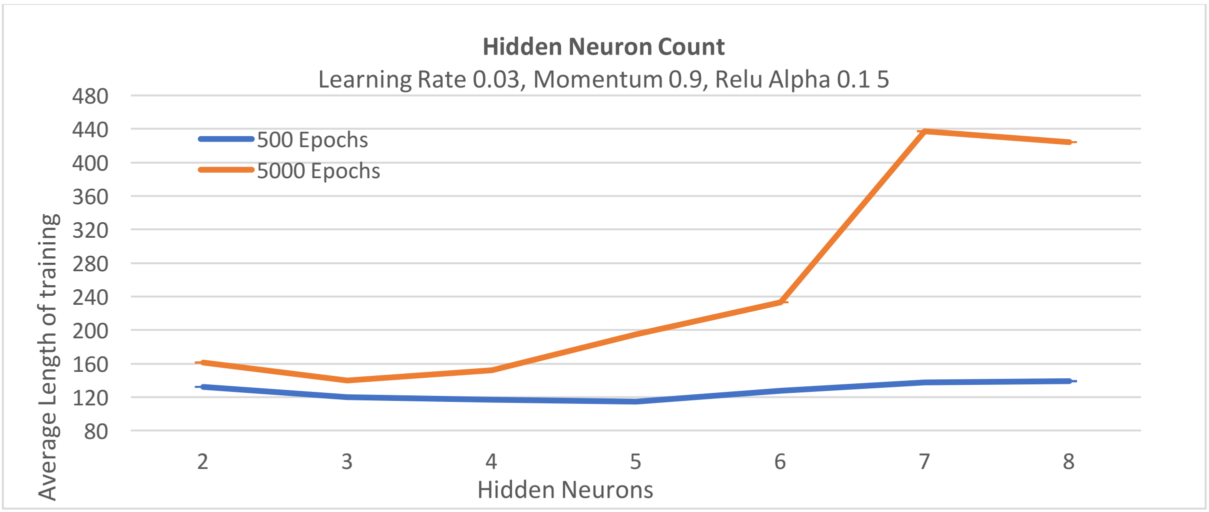
observation was that test population error didn't change much once the Relu function had an adequate leak level. This is more evident in the 500 epoch series. The error doesn't decrease until the leakiness reaches 0.05. But then regardless of whether you're using 500 or 5000 epochs, the error stays approximately the same until 0.5. Looking at average length of training, it's demonstrated that the higher alpha values of 0.1 compared to 0.01 result in much quicker learning. In fact, there were only 2% solved at 0.01 within 500 epochs. The Relu function without an alpha could not reach criterion at all (not shown). What I found surprising was that I got much better results using higher leak values than the industry standard. But that standard is prescribed for deep networks architectures where learning will be more difficulty and over many more epochs. I choose to use the Relu alpha value of 0.15 at a standard error of 0.010602 across 5000 epochs.

After all the learning parameters were "optimised", I looked at adjusting the number of hidden neurons.

Unfortunately, due to the computational costs and increased run time I was only able to investigate a hidden layer range of 2 - 6.







I was surprised with the effects on the error rate, I was expecting more variability with changing the number of neurons. The other two graphs were as expected, the network would become harder to train, take longer, and have less models reaching solution before they ran out of epochs. What these graphs don't do justice is the actual amount of time (in minutes, rather than epochs) that the training took. I was originally planning on going up to 12 neurons but my hardware limitations quickly told me otherwise.

Overall the performance increases that were gained were mainly due to decreasing the error criterion. Second most important would be the learning rate followed by the Relu alpha value. The alpha value doesn't have a direct correlation with learning performance, in this shallow network architecture it had a wide sweet spot, anything between 0.05 and 0.2 got the same level of training performance but with differing duration and number of successful models. In hindsight my testing methodology was quite flawed, and getting the mean and variance in the test population error didn't provide as much of a meaningful result as I wanted. I should have been recording every single result, so that I could have looked at upper and lower quartile concentrations and seen some other statistics, but I became time limited as I neared the conclusion of my research. I also believe that the sample size of the Iris set wasn't large enough to extrapolate too much from my conclusions. You can clearly see trends, and obvious better choices in some of the parameter selection, but the fine tuning is lost to noise. My plan was to generate a random Iris set of data (approximately 750) from adding random variability to the current set, and use that as the training set, and Iris itself as the test set, but then I ran out of time. Would this have increased the performance? It's still an interesting experiment I want to try. I also want to get around to implementing stochastic gradient descent, or mini batch from scratch. I can't overstate how much more understanding I have garnered from coding all of this from scratch.

Where to from here? Now that I have a bit more confidence on the inner workings of neural networks, I'll feel more comfortable using and tuning Tensorflow, and playing around with some multi-layer networks. Reinforcement learning really interests me, and I did really like working with genetic algorithms. I think mutating different parameters or network structures could be an interesting experiment, although in theory it'll be computational expensive.