Introduction

The goal of this paper is to discuss my implementation of a Multi-Layer-Perceptron network, and its' performance in three, pre-defined scenarios. A Multi-Layer-Perceptron is a feedforward artificial neural network containing at least three layers of nodes, which is trained using backpropagation and the gradient descent algorithm. The first scenario the neural network needs to be trained to solve is the classic XOR problem. Secondly, the network must be trained to predict the sin value of a combination of randomly generated numbers. Finally, the network will be trained to predict a letter value (from A to Z) given an existing set of numerical feature vectors and target labels. To surmise, the network must be capable of handling a binary problem, a linear problem, and a multi-class classification problem.

Method

My work is coded in Python 3.8.5. I decided to take an Object-Oriented approach and encapsulate my code into a class. I wanted to make the MLP object as modular as possible both to avoid writing repetitive code, and also to be able to easily modify hyper-parameters when conducting training experiments. However, given the scope of the assignment, my MLP class does have a fixed single hidden layer, and additional hidden layers can not be instantiated, although the number of hidden units may be modified. The MLP class has two obligatory parameters (the number of inputs and number of outputs), and a variety of default parameters that can be modified upon initialisation. The details of these parameters are documented in the provided Jupyter notebook. The methods of the MLP class follow most Python ML conventions; there is a fit method, and a predict method, and a number of internal methods that facilitate the implementation of the user's hyper-parameter choices, and the workings of the gradient descent algorithm.

For each of the three datasets my experiments took the same approach. After having conducted basic data exploration, I trained a baseline model with my "best guess" hyper-parameter choices. Once it had been established that the network was learning correctly, I then explored how varying the choice of activation function, the number of hidden units, the learning rate, and the weight initialisation strategy could effect the performance of the network. I used the loss curve, and accuracy as evaluation metrics. "Accuracy" in the case of XOR or the letter recognition dataset was simply considered to be the proportion of correct predictions. With regards to the sin experiment, the output of the model was continuous and hence I choose to use the sum of the squared error as an accuracy metric. Given that all the data was relatively balanced, I deemed it unnecessary to look at F1 scores, precision or recall metrics. When considering the effect on loss, I was looking to see how the slope and lower bound of the loss curve changed, as this is indicative of the speed of convergence and the lowest loss fit of the model during training. The loss value is saved internally after each epoch of training. A separate error metric is calculated and saved after each epoch of training and can be accessed at MLP.training_error. For the XOR training and sin training the error metric used is the sum of the squared error, and for the letter recognition data, the error metric used is the misclassification rate. The calculated error and loss for each experiment performed have been provided in a folder named "evaluation-results".

Two loss functions have been provided; Cross-entropy loss and Mean Squared Error. Four activation function options are given; Sigmoid, Tanh, Softmax, and linear. The linear "activation function" is essentially the absence of an activation function, as the goal of an activation function is to add non-linearity to the training of a network. Another quirk to be noted is that the "derivative" of my Softmax activation returns an array of ones, and the "derivative" of my Cross-entropy loss returns the difference between the predictions and the target value. Neither of these are the real derivatives per-say but, in the context of my project the Softmax activation function is only ever used in conjunction with the Cross-entropy loss function, and vice-versa, and when the derivatives of these two functions are multiplied together to fulfil the chain rule, they can be simplified to the difference between the output and the

targets. The network has four possible weight initialisation strategies; He[1], Xavier[2], random, and zero initialisation. Although it is generally not advisable to initialise the network's weights using entirely random numbers or zeros, I considered the inclusion of these two strategies interesting for the purposes of benchmarking.

I also allowed the user to make use of learning rate decay. Learning rate can be decreased or increased it by X every N epochs, where X is the decay parameter and N is the interval parameter.

All of the graphs and tables provided in this paper can also be viewed in the provided Jupyter notebook.

XOR experiments

	T1	T2	T3	T4
Target	0	1	1	0
Sigmoid	[0.13514246692871365]	[0.8444193737675844]	[0.8675596468277179]	[0.16950249909755702]
Tanh	[0.00017245096451317792]	[0.9894092369967675]	[0.9860464105713841]	[0.0004333373902748717]
Linear	[0.4999999999999997]	[0.4999999999999999]	[0.5]	[0.5000000000000001]

Figure 1, XOR test 1 model outputs

The choice of the XOR problem has its' roots in the history of Connectionism. Minsky and Papert's 1958 book contained a series of proofs which illustrated the limitations of Rosenblatt's much-lauded Perceptron, and in the aftermath of its' publication it became a benchmark against which many researchers tested networks.[3] The inability of the Perceptron to learn XOR was one of the more popular proofs from the work.
The XOR problem, having only four possible outputs, leaves no room for a train/test split. Hence, our training data is our test data, and notions of "over fitting" do not apply here. For this problem, the neural network requires two input units, N hidden units, and a single output. Mean Squared Error was used as the loss function for all experiments in this section. The Softmax activation function was not tested for this problem, as it is most suitable for multi-class classification problems. As an initial benchmark, I trained three different networks using Sigmoidal, Tanh, and linear activations consecutively. The number of hidden units, and the learning rate were fixed arbitrarily at 4 and 0.1. I allowed the network to train for 10,000 epochs and used a "He" weight initialisation strategy. As seen in the table in figure 1 on the right, when using Tanh or Sigmoidal activations the network learned successfully, but the linear activation failed to learn the problem. This is the expected result, as the XOR problem requires non-linearity to be introduced to the network.

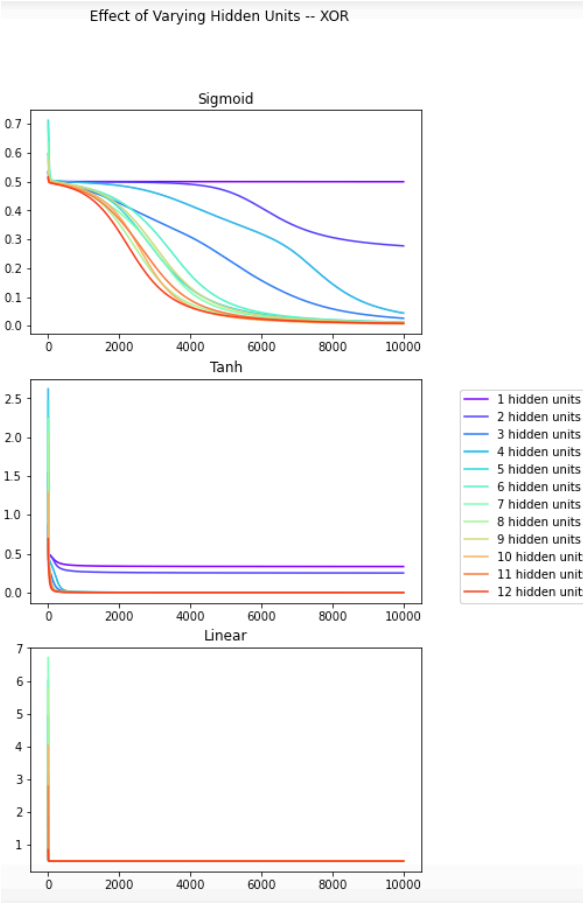

Figure 2, XOR loss for varying hidden units

I next varied the number of hidden units and allowed the network again to train for 10,000 epochs. The resulting loss curves can be seen in figure 2, on the right. The network using the Sigmoid function proved the most reactive to the number of hidden units, with the loss globally converging more quickly as the number of hidden units increased. Tanh required at least 3 hidden units for loss to converge to zero within 10000 epochs, but beyond 3 hidden units it behaved in a very stable manor. This does not necessarily mean that a network with two hidden units and Sigmoid or Tanh activations is incapable of

[1] Ji He, Man Lan, CHew-Lim Tan, Sam-Yuan Sung and Hwee-Boon Low, "Initialization of cluster refinement algorithms: a review and comparative study," *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, 2004, pp. 297-302

[2] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010.

[3] Minsky, Marvin, and Seymour A. Papert. Perceptrons: An introduction to computational geometry. MIT press, 2017.

learning the XOR problem, it could very well be related to the learning rate, the number of epochs trained, or the initialisation strategy. The linear network was unaffected by variations in the hidden units and did not learn.

I next experimented with the learning rate, training the networks on a range of learning rates between 1 and 0.0001. Larger learning rates of 1 and 0.5 caused issues in the linear network, with the loss exploding to infinity. These loss values were too large to be plotted in the figure on the right. The network with Sigmoidal activations was quite sensitive to changes in the learning rate and performed best with a larger learning rate. We can see as the learning rate grows smaller, the network fails to converge in 10,000 epochs. The network using Tanh seemed significantly more stable, although it too failed to converge when the learning rate became very small. We can see that loss oscillates quite a bit when the learning rate is one, as the purple loss line is quite thick. This is indicative of the network "bouncing" across a local minima due to the overly-large learning rate. The linear model was effected by learning rate, but the outcome is the same nonetheless, it does not learn the XOR problem.

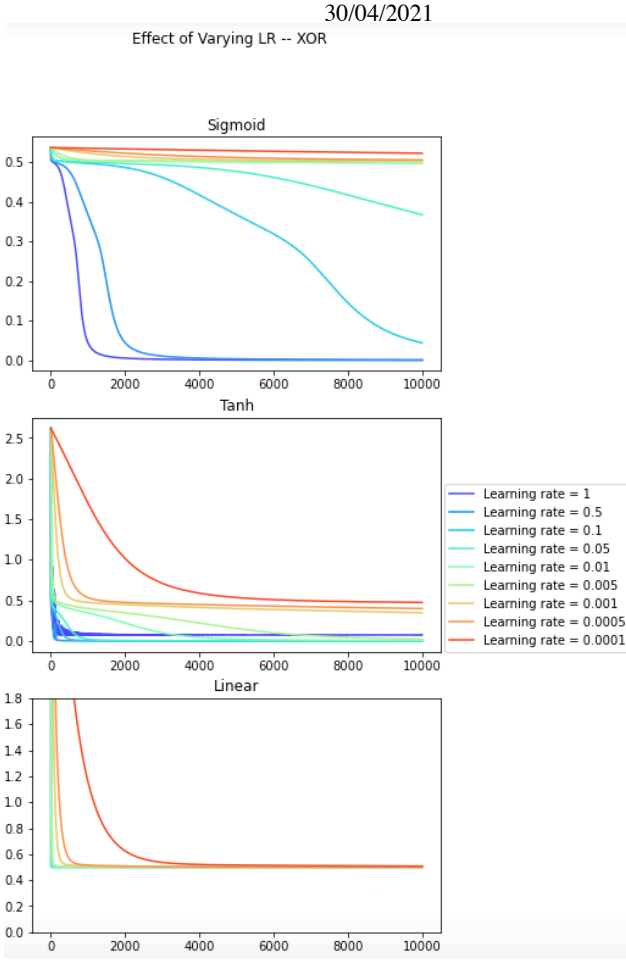

Figure 3, XOR loss for varying learning rates

I then tested the effect of varying the weight initialisation strategy. I kept the learning rate fixed at .1, and the number of hidden units set to 4 and again allowed the network to train for 10,000 epochs. Overall both the He and Xavier initialisation strategies led to very similar results at a fairly similar rate for Tanh. The Xavier initialisation did result in a much quicker drop in loss for the Sigmoid function, with the He initialisation performing much on a similar level as the random initialisation. Initialising the weights to zero globally did not work, and prevented the network from training correctly.

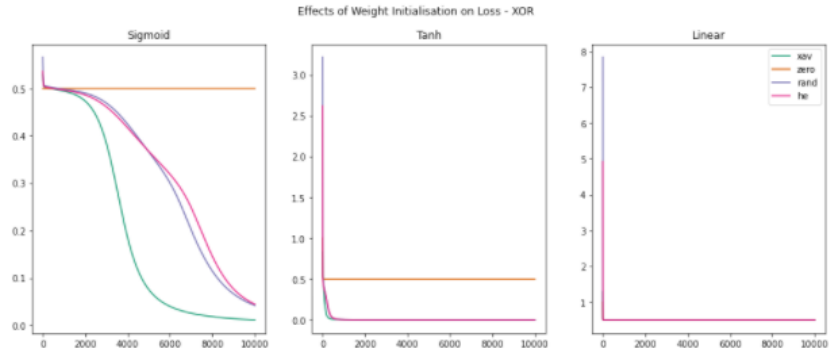

Figure 4, XOR loss for varying weight initialisations

With all the previously mentioned weight initialisation strategies, the bias terms were initially set to zero. Having read that for ReLU activations (which I chose not to include in my MLP class), initialising the bias terms to slightly above zero can improve performance[4], I was curious to see whether a similar effect would occur with the Sigmoid or Tanh activation. I kept the learning rate, number of epochs and number of hidden units unchanged and I used a Xavier initialisation strategy for these tests. I varied the bias terms from slightly below zero up to a bias of 1. As you can see in figure 5 on the right, the linear network and the network using Tanh

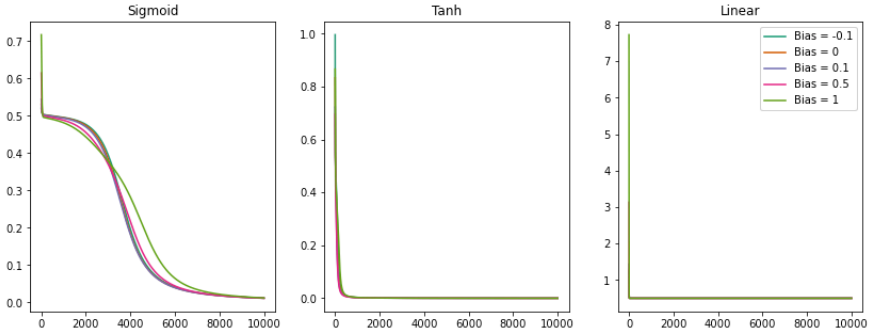

Figure 5, XOR loss for varying bias initialisations

[4] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ accessed 2/05/2021

activations were completely impervious to the variation in the initialisation value for the biases. The Sigmoid activation was sensitive to the bias term, but not dramatically so. It performed best when the bias was zero or close to it, and a bias of 1 slowed the convergence of the loss slightly. The network reached convergence closer to 8000 epochs when the bias terms were initialised to 1, as opposed to around 6000 epochs when the bias terms were close to zero.

	Sigmoid	Tanh	Linear
Test SSE	16.596315	0.446340	9.801122
Train SSE	7.913301	0.201218	3.184282

Figure 6, Sin SSE values for train and test sets

Sin experiments

The Sin dataset was created by randomly generating 100 samples containing four numbers in the range -1,1, with a target value calculated as *sin(x1 - x2 + x3 - x4)*. Hence, the network has four input units, N hidden units, and a single output unit. The target values are continuous values in the range -1, 1, and MSE was used as the loss function , and SSE as the error metric. The data was divided into a training set, and a test set, using a 75/25 split. The purpose of splitting the data is to be able to check the ability of the network to generalise to unseen samples after it has been fit, and hence verify that it hasn't been overfit. Overfitting occurs when a network has learned to predict the training data too closely, and is incapable of accurately predicting unseen samples.

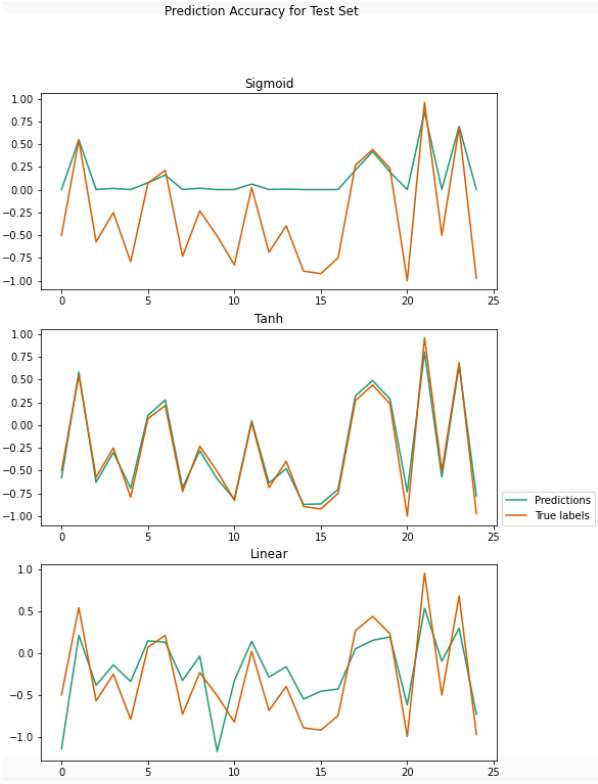

Figure 7, Sin - test set predictions and true labels plotted

I initially tested the network by training it for 10000 epochs, with a learning rate of 0.01, two hidden units and varying activation functions. The test and train error for the Sigmoid-activated network was very high as compared to the others, which can be seen in the table in figure 6 on the right. Once plotted, it became clear why this was so. In figure 7, you can see the predictions for the test set plotted against the actual targets. The Sigmoid function squashes values into a 0,1 range, and is hence mis-predicting every value that falls below zero. The linear network is approaching a fair approximation of the data, and the network using Tanh is performing extremely well.

I then varied the number of hidden units and considered the effect on the loss curve. As was observed with the XOR problem, the Sigmoid-activated network proved more sensitive to variation in the hidden units than the Tanh and linear networks. That said, it ultimately did nothing to improve the overall performance of the Sigmoid network. For both the Tanh-activated and linear networks, smaller quantities of hidden units slowed the convergence of the loss slightly, but ultimately the loss converged before 250 epochs for either network, independent of the number of hidden units used. The resulting loss curves can be seen in figure 8, on the right.

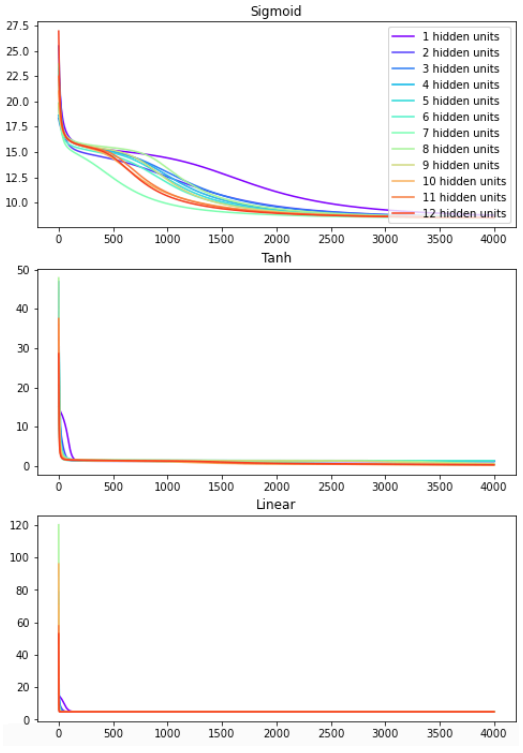

Figure 8, Sin - loss curves for varying numbers of hidden units

LR	sigmoid	tanh	lin
1.0000	7.8718	6.9150	NaN
0.5000	7.8734	6.0607	NaN
0.1000	7.8739	0.7234	NaN
0.0500	7.8735	0.1742	2.7601
0.0100	7.8935	0.1125	2.7601
0.0050	7.9337	0.4102	2.7601
0.0010	10.0839	0.4205	2.7601
0.0005	10.2542	0.4464	2.7601
0.0001	11.3057	0.4957	2.7603

Figure 9, Sin - SSE results for test set predictions

I next varied the learning rate for the models. The linear model reacted extremely adversely with learning rates larger than 0.1. The loss quickly exploded and the resulting models were unusable for the purpose of prediction, as we can see in the SSE results for the test set in figure 9, on the left. On the next page we can see in the resulting loss curves. The network

using a Sigmoid activation is again reacting positively to larger learning rates, with smaller learning rates not reaching its' potential minimum loss within 10,000 epochs. The Tanh network is behaving almost inversely, with larger learning rates provoking great instability, and smaller learning rates converging within the 10000 epochs trained. The linear model, disregarding the three iterations for which it failed to train (with learning rates greater than or equal to .1), converged quickly and seems entirely insensitive to the decreasing learning rates.

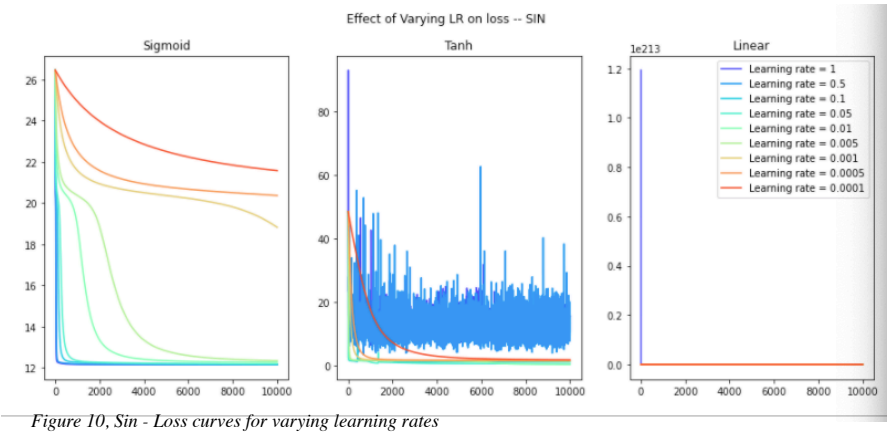

Figure 10, Sin - Loss curves for varying learning rates

In figure 11 on the right you can see the plotted loss curves produced by varying the weight initialisation technique. The network this time was trained for 4000 epochs, and the learning rate was maintained at .01. We can see that again, both the Tanh-activated and linear networks are relatively insensitive to weight initialisation strategy variations. The exception being zero initialisation, which remains a poor choice for all. The Sigmoid network is again more sensitive to variation, but even the zero initialised network manages to achieve parity with the other initialisations, due to the inability of the Sigmoid-activated network to learn this data.

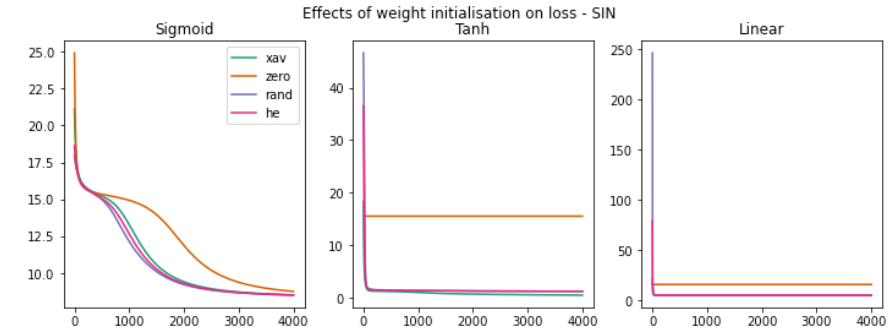

Figure 11, Sin - Loss curves for varying weight initialisations

Letter Recognition

The letter recognition data was provided, but required some preprocessing prior to its' use in the network. First of all, the target values needed to be converted to a numerical representation. In order to achieve this, I converted the characters directly to ints which corresponded to their ascii value, and then deducted 65, so that the letter 'A' became 0, 'B' became 1, etc. Next, the target values needed to be converted to a wide representation. The Softmax function essentially returns the probability of a sample being one of multiple classes, and those values when summed across all potential classes add up to one. Hence, our network will have 26 output units (for the 26 letters of the alphabet) and our target training values must also be 26 units wide, with the target class equal to 1 and all other classes equal to 0. Finally, the feature vectors are normalised by dividing by the maximum value present, ensuring that all values are between 0 and 1. The network has 16 input units, N hidden units, and 26 output units. The chosen loss function is Cross-entropy loss, which works well with the Softmax activation, and internally the training error is recorded using a misclassification rate metric. The Softmax activation should only be used on the final layer, so for the first layer we will still be testing Sigmoidal activations, Tanh activations, and linear activations.

	Sigmoid	Tanh	Linear
Train	0.533636	0.139009	0.038803
Test	0.537600	0.136400	0.041400

Figure 12, Letter Recognition - accuracy for initial test

As an initial test of the network I set the learning rate to 0.001, initialised 10 hidden units and let the network train for 2000 epochs, varying the first activation function. The linear function had issues with exploding loss and essentially did not learn. Going forward, I decided to use only Sigmoidal or Tanh units in experiments. In the results on your right in figure 12, we can see the classification accuracy rate for the

network using Sigmoidal units is significantly outperforming the Tanh and linear networks on this initial try.

I next tested the effect of varying the number of hidden units from 1 to 26. Both the Sigmoidal-activated network and the Tanh-activated network were quite sensitive to the number of hidden units. A box plot summarising the accuracy scores on both the train and test data can be found in figure 13 on the right. It is clear from this graph that the network using a combination of the Sigmoid activation with the Softmax activation is wholly out-performing the network using Tanh. Please see the table in figure 14 for the full results of these experiments. Both networks perform their best when the number of hidden units is high. The Sigmoidal network has its' maximum test and train accuracy with 26 hidden units, and the Tanh network has its' maximum test and train accuracy with 25 hidden units.

I then varied the learning rate, with the number of hidden units fixed to 26. Neither the Sigmoid-activated nor the Tanh-activated networks learned well with a larger learning rate in the context. Both networks begin to learn well when the learning rate was smaller than or equal to 0.005, with the Sigmoid-activated network preferring a learning rate of 0.001, and the Tanh-activated network achieving its' best accuracy scores with a learning rate of 0.0005. The full results table of these scores can be seen in the Jupyter notebook.

I then tested the network with varying weight initialisation strategies. The accuracy results can be seen in figure 15 on the right. Globally the He and Xavier initialisations perform very similarly, but the Xavier initialisation does slightly outperform the He initialisation in three of the four accuracy scores. The randomly initialised weights do okay, and when the weights are initialised to zero, the networks fail to learn.

Finally, I decided to try and improve the performance of the network by implementing learning rate decay. The intuition behind learning rate decay is that as the gradient descent algorithm moves down the error surface, it may need to start to take "smaller steps" in order to avoid getting stuck in local minima, and to continue moving towards a global minima. For this portion of the testing, I used only the Sigmoid activation with the Softmax activation, as it had been consistently outperforming Tanh. I implemented a scheduled learning rate decay, where one can choose an interval value and a decay value. At every N epochs, the learning rate will be

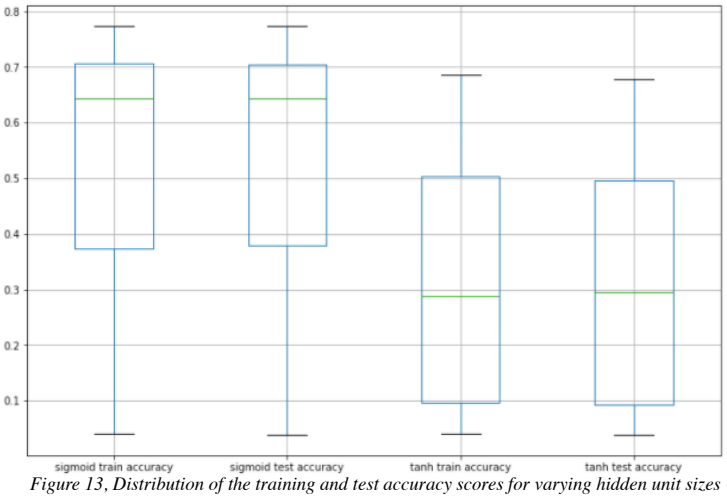

Figure 13, Distribution of the training and test accuracy scores for varying hidden unit sizes

Hidden Units	sigmoid train accuracy	sigmoid test accuracy	tanh train accuracy	tanh test accuracy
1	0.041269	0.0388	0.041269	0.0388
2	0.041269	0.0388	0.041269	0.0388
3	0.084406	0.0780	0.041269	0.0388
4	0.087939	0.0836	0.073538	0.0716
5	0.325088	0.3260	0.090206	0.0854
6	0.321621	0.3294	0.111674	0.1114
7	0.342023	0.3476	0.041269	0.0388
8	0.494566	0.4996	0.081005	0.0740
9	0.611174	0.6246	0.245616	0.2466
10	0.467698	0.4724	0.150277	0.1516
11	0.638776	0.6422	0.176878	0.1780
12	0.641243	0.6402	0.350823	0.3546
13	0.627509	0.6302	0.254884	0.2610
14	0.654310	0.6578	0.322755	0.3286
15	0.688979	0.6802	0.376825	0.3740
16	0.713448	0.7076	0.468431	0.4734
17	0.643510	0.6492	0.384959	0.3828
18	0.643576	0.6434	0.235216	0.2252
19	0.683246	0.6768	0.570105	0.5752
20	0.699513	0.6964	0.512968	0.5044
21	0.716914	0.7092	0.476365	0.4666
22	0.758451	0.7600	0.632909	0.6278
23	0.707380	0.7078	0.531569	0.5310
24	0.742249	0.7434	0.579105	0.5744
25	0.741249	0.7354	0.685312	0.6778
26	0.773852	0.7732	0.652644	0.6486

Figure 14, Training and test accuracy scores for varying hidden unit sizes

Weight Init	sigmoid train accuracy	sigmoid test accuracy	tanh train accuracy	tanh test accuracy
xav	0.7745	0.7696	0.6666	0.6558
zero	0.1829	0.1912	0.0413	0.0388
rand	0.6768	0.6666	0.5606	0.5604
he	0.7739	0.7732	0.6526	0.6486

Figure 15, Training and test accuracy scores for varying weight initialisation

multiplied by the decay value. I initially tested using an interval value of 500, for 1500 epochs (meaning the learning rate would change twice; at the 500th epoch, and the 1000th epoch), and with a range of decay values from .95 to .4.

Decay Rate	sigmoid train accuracy	sigmoid test accuracy
1.00	0.7745	0.7696
0.95	0.7973	0.7870
0.90	0.7939	0.7798
0.85	0.7938	0.7834
0.80	0.7944	0.7846
0.75	0.8051	0.7966
0.70	0.8035	0.7950
0.65	0.8003	0.7938
0.60	0.7982	0.7898
0.55	0.7948	0.7862
0.50	0.7917	0.7828
0.45	0.7878	0.7786
0.40	0.7833	0.7742

Figure 16, Training and test accuracy scores for varying decay rates

The resulting accuracy scores on both the training and test sets can be seen in figure 16 on the left. The decay rate of 1 is equivalent to no learning-rate decay, and it can be seen than every decay rate improved the training and test scores of the network as compared to training with no learning-rate decay. To test this further, I decided to let the network train for significantly more epochs (105,000) both with and without learning-rate decay, and compare their final accuracy scores. I chose .8 as the rate of decay, and set 1000 as the interval. I also set a minimum bound for the learning rate of 0.0001, to avoid the learning rate disappearing into nothing over the long training time. The network with learning-rate decay had a final accuracy score on the training set of .886 or 88.6%, and a final accuracy score on the test set of 87.06%. The network without learning-rate decay, having trained for the same amount of time with the same hyper-parameters, had a final training set accuracy of 84.1% and a final test set accuracy of 82.4%. This is a significant difference, and when we take a look at the plotted loss curves it's very clear that the network without learning rate decay had a very unstable loss descent, and was missing minima due to its' overly large learning rate.

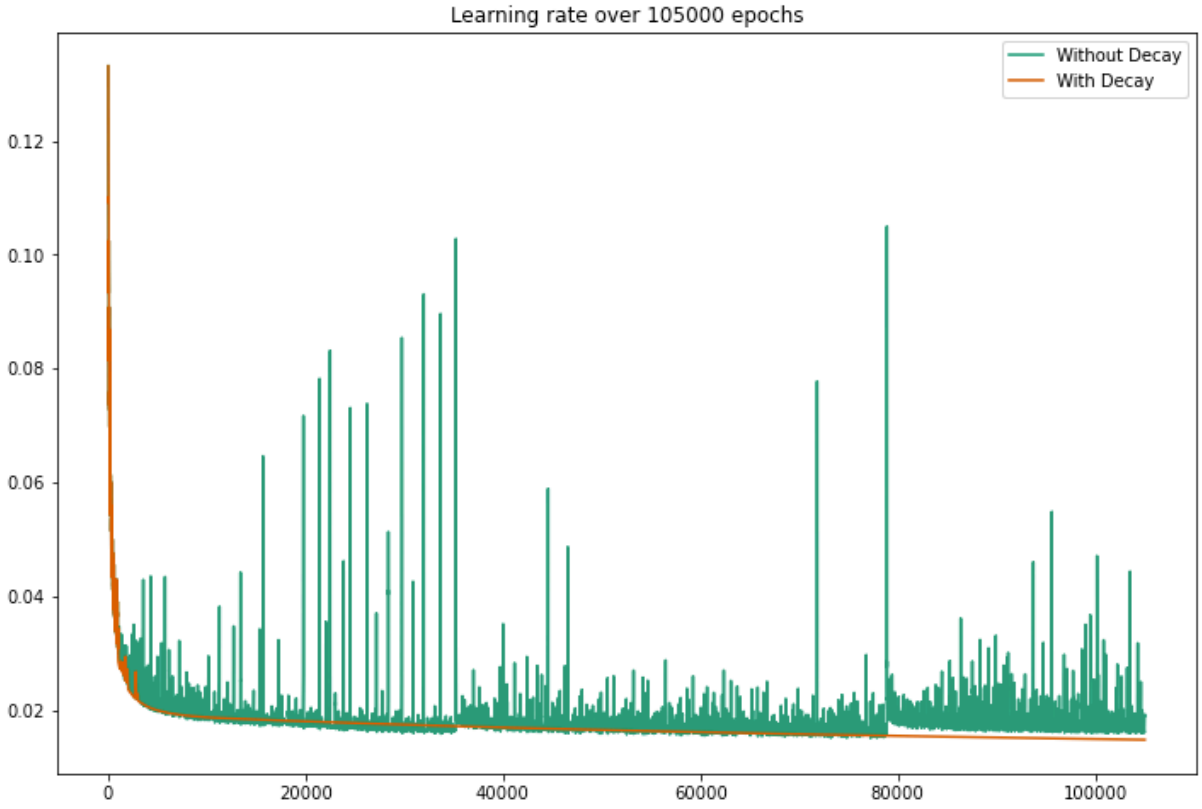

Figure 17, Loss curve for 105,000 epochs with and without learning rate decay

Conclusions

When I initially established that my implementation of gradient descent was working, and that my network was learning the XOR problem, I thought that the hardest part of this project was finished. I was quite mistaken. The implementation of a working gradient descent algorithm is but a key in the door to a labyrinth of potential hyper-parameter choices, and the successful training of a network is dependent on a massive number of small and large decisions. As it is, I'm fairly certain that the accuracy score for the letter recognition data could be further improved with added optimisations such as implementing an Adam optimiser or L2 regularisation. Or, possibly exploring mini-batch gradient descent to speed up training times. There are also other loss functions and activation functions that could have been tested. In short, the more I learn, the more there seems to be to learn, and the possibilities for tweaking a network are vast. That said, I do ultimately feel that my testing and experimentation strategies were solid. Establishing a good activation function, learning rate, number of hidden units, and weight initialisation strategy, led to a solid foundation for each network.

Secondly, working with the letter recognition data really reinforced the importance of data normalisation. When initially implementing the Softmax activation and performing exploratory tests on my data, I was convinced my code was flawed or broken somehow. I spent many hours trying to find a bug in my code which didn't exist. I had simply failed to normalise the data and the network was failing to learn because of this. This simple step, which equates to one line in my code, was absolutely key to the successful performance of my neural network. One can tune as many hyper-parameters as one wants, but if the data is not pre-processed correctly, the network will not perform to the best of its' abilities. It is also essential to take the time to understand your data in order to be able to best pick the correct loss or activation functions. In the sin dataset, had I chosen to use only a Sigmoid activation, without looking at or considering the range of target values, I would have a network performing sub-optimally and no clear understanding of why.

Overall, I feel that my MLP class has been successful in the three prescribed scenarios. The XOR model achieved 100% accuracy when using a Sigmoid or Tanh activation. The sin model, when trained with Tanh activations, achieved an SSE score of only 0.365. On the letter recognition data, my model achieved 87.06% accuracy on the unseen test set.

In conclusion, I am coming away from the project with a much better understanding of backpropagation and the gradient descent algorithm, the effects of the activation functions, weight initialisation, learning rates and the challenges of tuning a Connectionist model. Neural networks are not "one-size-fits-all" and small choices can have significant consequences.