Connectionism Research Assignment

Implementation Of A Multi-Layer Perceptron

Edward O'Neill

Contents

1 Introduction

The following paper details the implementation and experimentation of a Multi-Layer Perceptron (MLP). The Multi-Layer Perceptron was implemented using Python and its library numpy for the mathematical calculations needed within functions such as those found in forward propagation and backward propagation. The MLP was trained on the following tasks : training to learn exclusive or, training to learn sin where given an input vector v of four integers ranging between negative one and positive one, the output $\sin(v_1 - v_2 + v_3 - v_4)$, and letter recognition from data from the UCI Machine Learning Repository. The activation function and its derivative used in the exclusive or case was sigmoidal and in the sin case tanh and its derivative.

The Forward Propagation algorithm was constructed by applying an activation function to the multiplication of the input and the first set of weights to produce an output A1. This result was then multiplied by the second set of weights and the activation function was applied again to produce the output of the Forward Propagation, A2.

The Backward Propagation algorithm was derived by reasoning about what the state of affairs are after the Forward Propagation is performed. The target is taken away from the output A2. This result is used to calculate the derivative of the Forward Propagation output A2 by multiplying it by the result of the activation functions derivative being applied A2. The derivative of the second set of weights is then calculated with the dot product of A1 and the derivative of A2. The steps are repeated to find the derivatives of A1 and the first set of weights. The dot product of the derivative of A2 and the second set of weights is calculated. The derivative of A1 is then calculated by multiplying the previous dot product by the activation functions derivative being applied to A1. This result is then multiplied by the dot product of the input and itself to give the derivative of the first set of weights.

2 Coding Choices

The code base for the perceptron such as the propagation algorithms and other functions such as plotting results were put into the file mlp.py.

When I initialised the weights, I scaled them by a factor of $\sqrt{\frac{1}{X}}$ where X is the size of the input dimension which provided good results. Two metrics were considered : error and accuracy. The error was reasoned about being the distance a predicted value is from the target/actual value. Regarding accuracy however, in some instances I found a negative accuracy was produced and whenever this occurred, it was set to zero. The error was calculated in the following way :

Listing 1: Error Calculation

```
error = np.sum(np.abs(A2 - Y)) / len(Y)
```

In the exclusive or case, when a prediction was made, if the entry was below 0.5 it was set to zero otherwise one to give the prediction output.

3 Experiments & Results

3.1 Exclusive Or

The XOR was trained given the input (0,0), (0,1), (1,0), (1,1) and it's corresponding output (0,1,1,0). The outputs and hidden units in this case were both sigmoidal as the sigmoid function brings an integer n between the two possible outputs, zero and one.

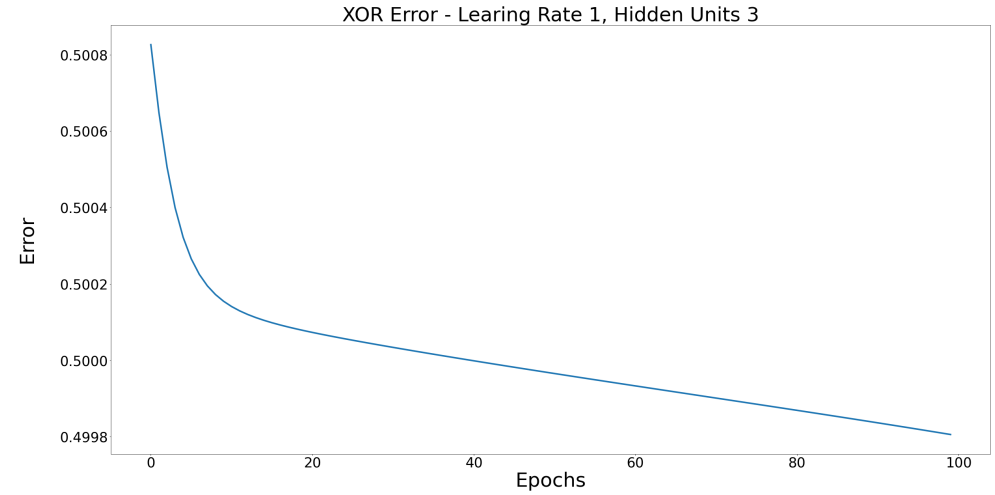
Since there is minimal input, it can be used entirely within the training of the Multi-Layer Perceptron. Later on, weights from an experiment which gave the correct output were tested against a large N sample space consisting of five hundred entries and the appropriate output.

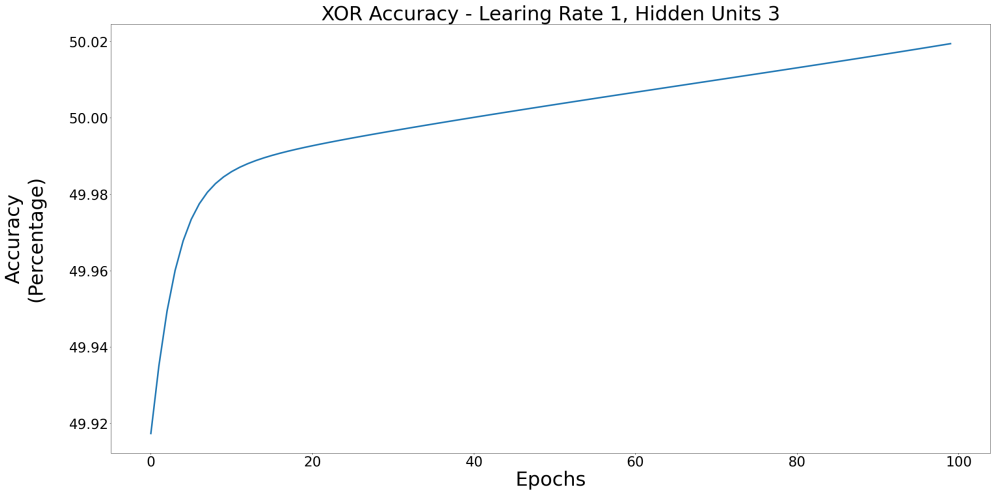
The experiments of learning the XOR function involved learning the best hyper-parameters, how the number of hidden units affect error produced, how different learning rates affect error produced, and then how well the best learned parameters are against those trained using large input and output. The results of the experiments can be found in the following table. P.Error and P.Accuracy refer to the error and accuracy of the prediction after training has completed.

3.1.1 Acquiring Good Hyper-Parameters

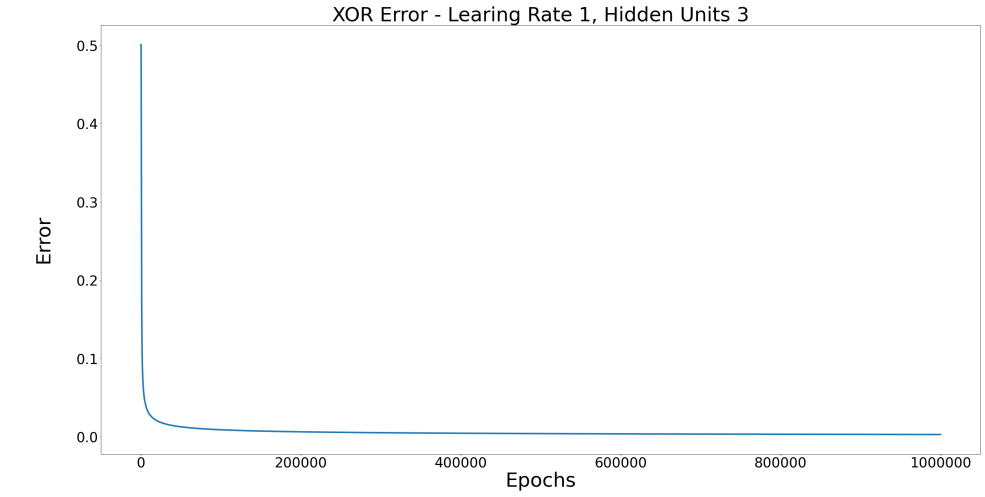
Experiment	Epochs	Hidden Units	Learning Rate	P.Error	P.Accuracy	Correct Prediction
1	100	3	1	0.4998	50%	No [0,0,1,1]
2	1000	3	1	0.1896	100%	Yes [0,1,1,0]
3	10000	3	1	0.0298	100%	Yes [0,1,1,0]
4	100000	3	1	0.0087	100%	Yes [0,1,1,0]
5	1000000	3	1	0.0027	100%	Yes [0,1,1,0]
6	1000000	4	1	0.0017	100%	Yes [0,1,1,0]
7	1000000	10	1	0.0011	100%	Yes [0,1,1,0]

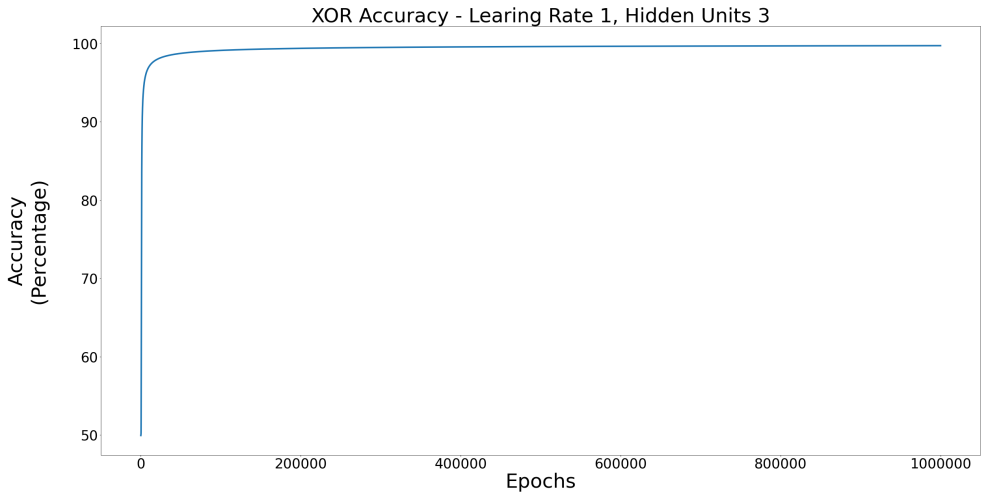
Interestingly, discovering the best parameters for the training of the exclusive or came quite easy. I began with a learning rate of one and found that at just a hundred epochs, the error was 0.4998 and the accuracy was 50% which was fairly good for the very first experiment.





The obvious step was to increase the number of epochs and I achieved the right prediction in the second experiment. I then decided to see what parameters gave the best output because the error produced could have been lowered. In experiment five, at a million epochs, I achieved an error of 0.0027 and an accuracy of 100% which were the best results achieved when the hidden units were set to three. Below are the results of experiment five.





3.1.2 The Effects Of Hidden Units

Increasing the hidden units to four had a beneficial effect. The error is further reduced to 0.0017. I suspected that increasing the hidden units even more would reduce the error further. Increasing the hidden units to ten gave an error of 0.0011 and an accuracy of 100% , a reduction of 0.0006. It became apparent that eventually there comes diminishing returns regarding finding the best error and accuracy. There really isn't any notable difference in the results in experiment seven and the results in experiment six bar the minute difference in error and given that they both produce the same prediction. I suspect that a much better and richer network would be able to reduce the error even

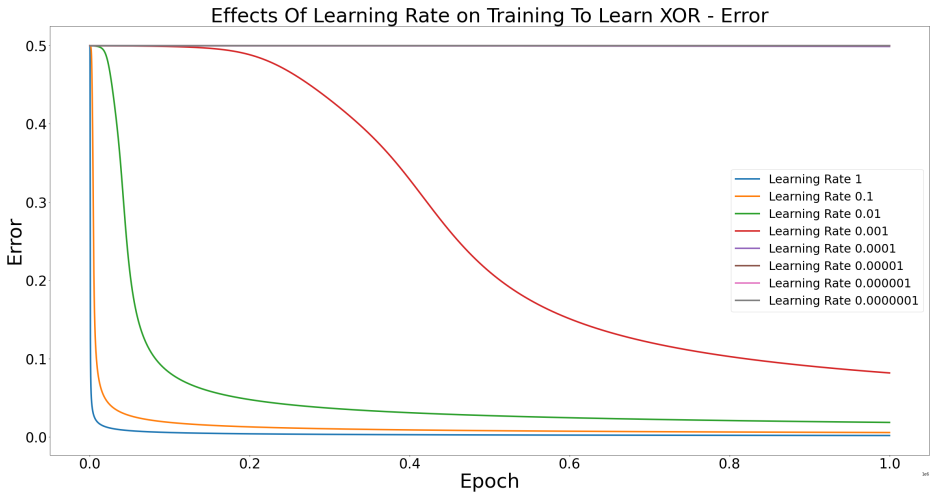
further to approach zero further or even reach zero but the results produced here are nonetheless satisfactory.

Knowing the best parameters provided context to the effects that learning rate has on training/learning. I would argue it is what effects training the most when one fine tunes a network as if one finds that the model learns but slowly, then the learning rate is what has to be adjusted.

3.1.3 The Effects of Learning Rate

The following table are the results of experimenting with different learning rates using four hidden units for a million epochs as this number of hidden units gave quite good results, better than when the hidden units were set to three. The learning began at one and was reduced by a factor of ten. The training was rewritten in each cell in the form of the loop from the MLP API such that I could take all of the errors produced to produce the results.

Experiment	Epochs	Hidden Units	Learning Rate	P.Error	P.Accuracy	Correct Prediction
8	1000000	4	1	0.0017	100%	Yes [0,1,1,0]
9	1000000	4	0.1	0.0055	100%	Yes [0,1,1,0]
10	1000000	4	0.01	0.0184	100%	Yes [0,1,1,0]
11	1000000	4	0.001	0.0816	100%	Yes [0,1,1,0]
12	1000000	4	0.0001	0.4988	50%	No [0,0,1,1]
13	1000000	4	0.00001	0.4998	50%	No [0,0,1,1]
14	1000000	4	0.000001	0.4999	75%	No [0,0,1,0]
15	1000000	4	0.0000001	0.4999	75%	No [0,0,1,0]



It was surprising that only half of the next set of parameters gave the correct output

however after some consideration it did make some sense. The set of parameters which produces the best output is finite therefore after a certain amount of alterations to the set, the output becomes incorrect again. Reducing the error shows that a laissez-faire approach cannot be taken when one performs hyper-parameter tuning. It requires careful management.

What I found interesting in observing the errors produced was seeing that at a certain value for the learning rate, the overall error calculated at the end converges upon the same point which can be seen in the results of experiments twelve to fifteen.

3.1.4 Testing With Large, Random Data

Experiment	Epochs	Hidden Units	Learning Rate	P.Error	P.Accuracy	Correct Prediction
16	1000000	4	1	0.2550	49%	No
17	1000000	4	0.001	0.0055	100%	Yes
18	1000000	10	0.001	0.0038	100%	Yes

To test the strength of the MLP, I generated five hundred random vectors and their subsequent exclusive or output and created an 80:20 split to create the training and test data. I then reran the training of experiment number six. Using the weights from this training, I tested them against the entire data and achieved a very good result of an error of 0.0017 and an accuracy of 100% . When I initialised new weights however and trained with the large training data and then carried out a prediction, the error was surprisingly large. The test error was 0.2550 and the test accuracy was 49% .

This was the point in which the importance of learning rate became that much more apparent. I reran the experiment using a learning rate of 0.001 as that gave good results in some experiments before and achieved better results of an error of 0.0055 and an accuracy of 100% .

From this I experimented by raising the number of hidden units to ten again. The purpose of this was to see if the size of the data and a high number of hidden units could influence the results unlike before where an increase in hidden units didn't amount to much. I was surprised by the results. I achieved an error of 0.0038 and an accuracy of 100% .

This perhaps suggests that increasing the number of hidden units of an already 'optimal' perceptron does help but has diminishing returns on how well it performs as it seemed to take a little longer to complete than the perceptron at four hidden units however both results are nonetheless satisfactory as even though there is a reduction in error, they both still produce the same prediction.

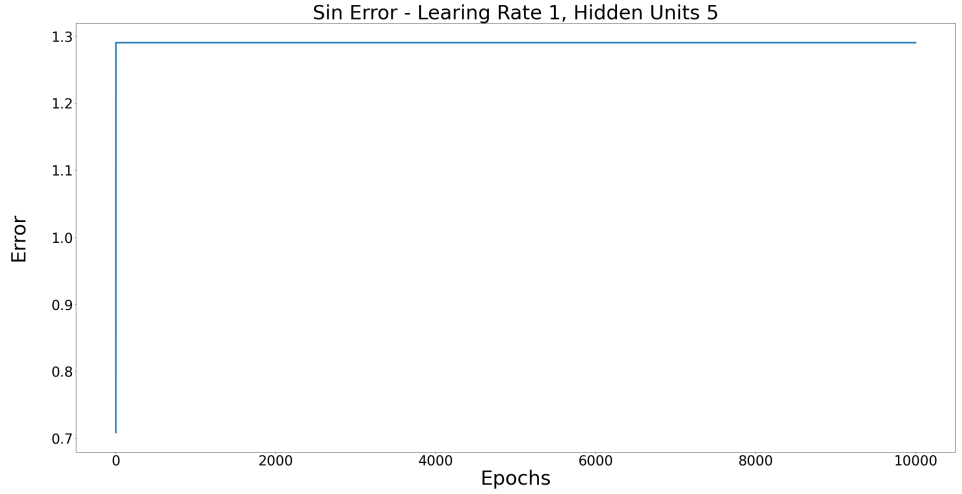
3.2 Sin

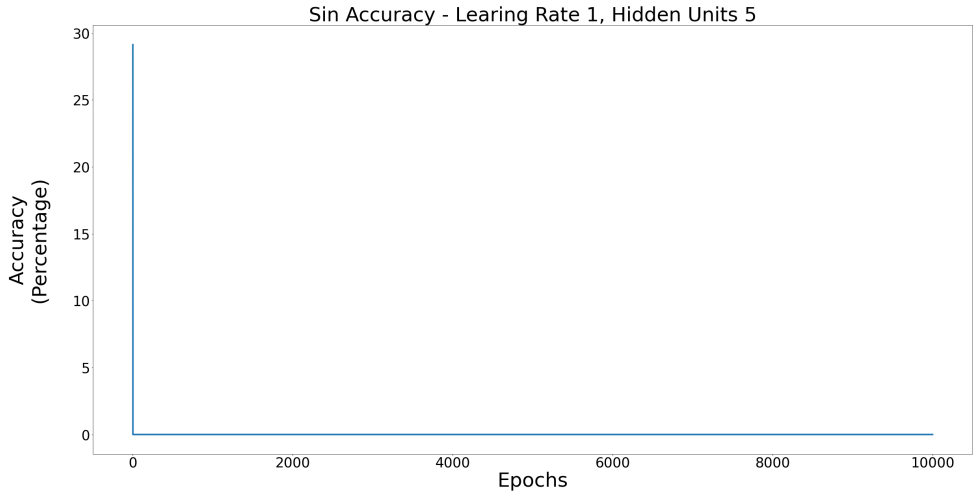
The experiments carried out here were to see how the learning rate is critical to acquiring the best hyper-parameters and the effects of increasing the number of hidden units. Five hundred vectors were generated consisting of four entries ranging between -1 and 1 and the corresponding output of any vector v $\sin(v_1 - v_2 + v_3 - v_4)$. Four hundred of these were reserved for training and one hundred for testing.

3.2.1 The Effect Of Learning Rate On Acquiring Good Hyper-Parameters

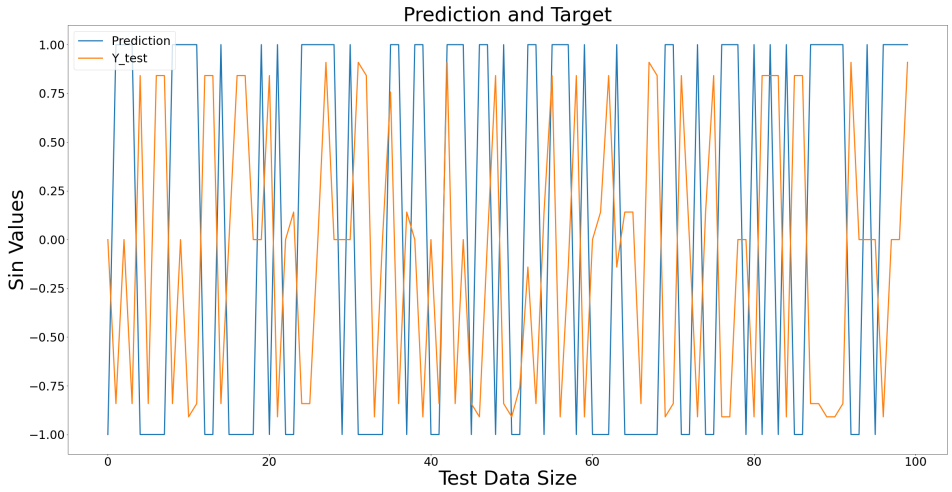
Experiment	Epochs	Hidden Units	Learning Rate	P.Error	P.Accuracy
19	10000	5	1	1.3527	0%
20	100000	5	1	1.3527	0%
21	1000000	5	1	1.3527	0%
22	1000000	5	0.1	0.4504	54.9563%
23	1000000	5	0.01	0.1504	84.9556%
24	1000000	5	0.001	0.0086	99.1320%
25	1000000	5	0.0001	0.0245	97.5460%

Going from what I learned from the first experiment, I began testing using ten thousand epochs, five hidden units, and a learning rate of one. Observe the results of experiment nineteen.





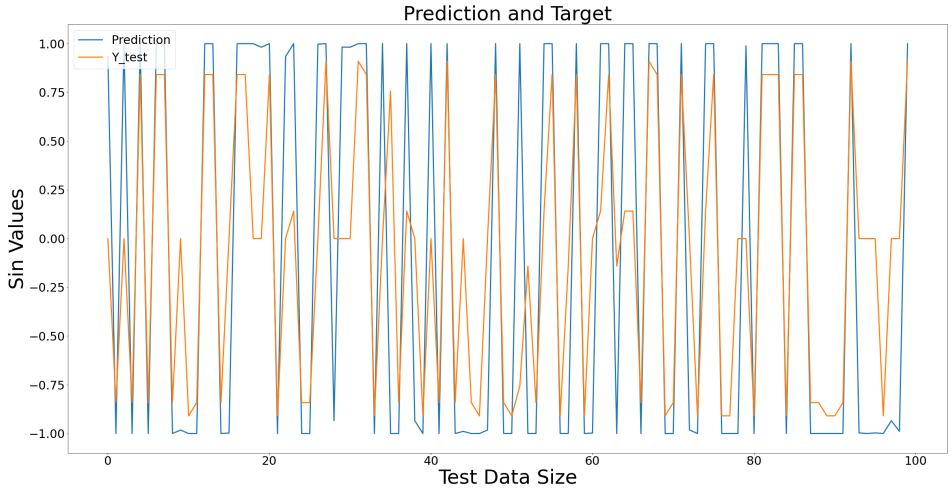
What I found very surprising was how much more meaningful it was to observe plotting a prediction against the test set. This way, it was easier to see how well the predictions stacked up against the actual values. For example, the results at the beginning of the testing produced the following where orange is the test set and the blue is the prediction:



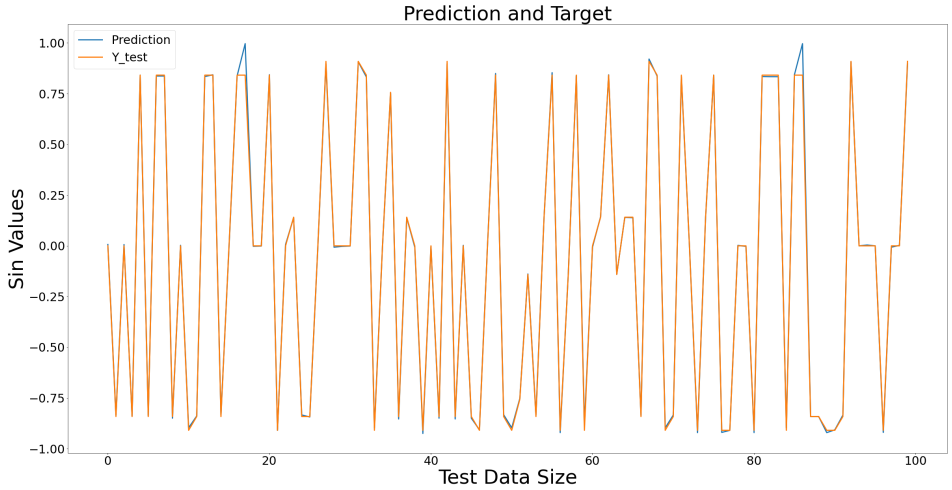
The greater the overlap the greater the success as the test set was plotted second therefore less blue present means that the prediction is more inline with the actual value of the test set.

Increasing the amount of epochs that the perceptron trains for does help but at a certain point the learning rate has to be reduced. Below in experiment twenty-two, at

a million epochs and a learning rate of 0.1, the results produce are much closer to zero. This gave an error of 0.4504 and an accuracy of 54.9563%



However the differences between the prediction and test set are too stark, reducing the learning rate again to 0.001 in experiment twenty-four below gave the best results of an error of 0.0086 and an accuracy of 99.1320% .



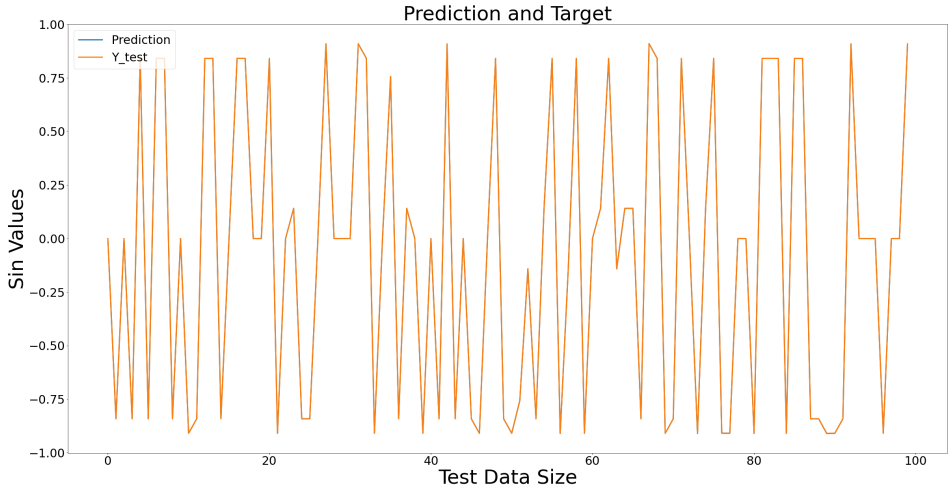
These results were very interesting as they showed in general, the process of fine tuning this task was very much similar to the tuning of the exclusive or task and it showed the importance that learning rate can have to acquire an optimal prediction/solution to a task.

3.2.2 The Effects Of Hidden Units

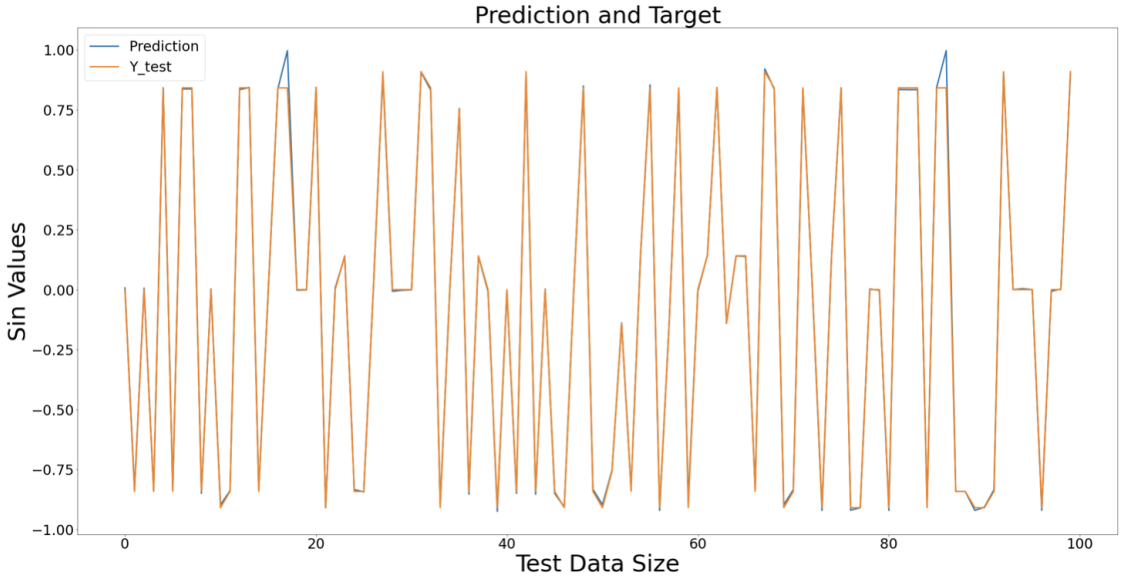
With the best hyper parameters found for when the number hidden units were five, the next step was to see the effects of increasing the number of hidden units from five to six and onward until there was no tangible benefit.

Experiment	Epochs	Hidden Units	Learning Rate	P.Error	P.Accuracy
26	1000000	6	0.001	0.0459	95.4093%
27	1000000	7	0.001	0.0102	98.9769%
28	1000000	8	0.001	0.000149	99.9850%
29	1000000	9	0.001	0.000505	99.9494%
30	1000000	10	0.001	0.000287	99.9712%

The effect of increasing the hidden units has a great effect on the task. The result of increasing the hidden size to eight approaches zero. This result shows that increasing the number of hidden units can have a great effect to an extent then it doesn't particularly make much of a difference for example in experiments twenty-nine and thirty because as the error produced, although extremely low, increased. Increasing the hidden units in this task however had a much better effect than in the exclusive or case. The difference between the errors of experiment twenty-four, the best set of experiments from the previous set at an error of 0.0086, and twenty-eight is large, at around 0.008 which is substantial given that they're both small but it can be seen more clearly in the results below. The final results in experiment twenty-eight produce the following output :



And the results of twenty-four:



As one can see, the difference between the two results is stark as the results of twenty-eight are near enough perfect whereas in twenty-four, part of the prediction is incorrect as denoted by the blue.

4 UCI Machine Learning Repository

I was surprised by the results here. I wasn't able to get to the perceptron to learn. To obtain the output values, I set Y to be the set of values from the first column which gave me the set of possible values, the 26 outputs / alphabet. I then converted these to ASCII values such that they could be used in training because it wouldn't work if the data was of two different types of data. X consisted of all the columns. I then converted the first column in X to its ASCII representation as done for Y. Then both X, Y were converted to numpy arrays. X had dimensions of (20,000, 17) and Y had dimensions of (1, 26) then the 4/5 split between training and testing was done.

I then went to run the training but what kept occurring was that I ran into broadcasting errors within forward propagation with regards to calculating the error and also the back propagation if I didn't calculate the error within the forward propagation.

I then decided to approach the problem differently and use the list of letters as my output in its entirety and keep the entire data set as my input. This approach did kind of work. It did train but there were a few complications. I began with a learning rate of one but this kept throwing an overflow error in the training as I was using sigmoid as my activation function. I had to reduce the learning rate to 0.00001 in order to mitigate this problem. Strangely the error always began at a more or less set number then drop to a constant error. For example in one instance in experiment thirty seven, the error at 10% was 77.21306576450324 then it dropped to 76.51600026241898 and the error at the end was 76.51600025748117 and the prediction error was 76.51975025681995. This

behaviour occurred at different learning rates and even when I raised the number of epochs and when I changed the activation function to tanh. I thought perhaps the problem stemmed from the activation function I was using so I tried used softmax but I ran into errors within forward propagation as there was dimension mismatch in it which couldn't have been solved by means of transpositon.

5 Conclusion

Overall I would say this project has taught me a tremendous amount in regards to the construction of networks and it has given me an appreciation of the work that has gone into the field of Connectionism. I had more experience with libraries such as Pytorch and so coming to this project wherein I had little experience constructing models from scratch, I found building the MLP very challenging but quite rewarding. I am extremely happy in the MLP that I have constructed as it performed satisfactorily on the learning of the exclusive or and of sin. However I am unhappy in the results of the letter recognition task. I would consider this as a testament to the difficulty of constructing Connectionist models. What I found upon the completion of all tasks is that careful planning of the network such as generalising Forward Propagation to use a general activation function 'activation_function' rather than hard-coding in one like sigmoid can mitigate problems in its construction but tuning the parameters of the network is surprisingly where one would spend most of their time.

This happened in my own case wherein I spent a lot of time testing in the thirty experiments to explore the effects of learning rate and hidden units. Throughout the experiments it became more and more apparent that the hyper parameters have a large impact on the results produced. For example in the very beginning, increasing the number of epochs from one hundred to one thousand in experiments one and two greatly helped with regards to exclusive or in that it was able to make a correct prediction. Increasing the hidden units however did have a beneficial effect but it didn't really change the results as much as I thought at first because I thought it would have a large impact on tasks in general, not just the sin task. This would indicate to me that increasing the number of hidden units has a large benefit for some tasks for example in Sin where in experiment twenty-eight, the prediction was tremendously close to the target, but not others like in the exclusive or cases where changing the hidden didn't have a large effect on the error produced for example in experiments six and seven wherein despite a similar error reduction to the Sin cases (in the case of the hidden units), 0.0006 as opposed to 0.0008, the two experiments both produce the exact same prediction. This then shows what constitutes an acceptable error can be predicated on the task at hand. A reduction of an already small error is some tasks doesn't mean much if the results are the same. The reduction in error was stark in the Sin case as observed in the graphs above but less important in the exclusive or case as any experiment with a small error gave the right prediction. However changing the learning rate is an applicable measure to take for all tasks.

To conclude, I am extremely happy with the results that I achieved in the two tasks

and implementing the Multi-Layer Perceptron has given me a great appreciation for the field of Connectionism and the work that has gone into richer models and libraries like Pytorch and Tensorflow that aid one in the creation of models in general.