Connectionist Computing

Multi-Layer Perceptron MLP

Conor Kiy

20204844

COMP41390

UCD School of Computer Science
University College Dublin

December 13, 2021

Table of Contents

Chapter 1: **Introduction**

The objective of this project is to build, and experiment with, a Multi-Layer Perceptron MLP. This report will briefly describe the implementation of the MLP used and will then discuss the experiments which were run. Two experiments on two different functions were carried out. For each experiment various tests were run using different hyperparameters and the results reported. A discussion of the findings is made in the conclusion.

Chapter 2: **Implementation**

The coding language used to implement the MLP was Python. Python was chosen for it's simplicity and due to my confidence in my ability and understanding of the language. The MLP was implemented as a class in a standalone file which allowed it to be imported as a module into the various test files.

The MLP class itself has all of the necessary attributes and the various methods: randomise, forward, backward, and update_weights. The training technique for the MLP is implemented in the test files. This implementation is specifically a two-layer (hidden layer and output layer) MLP which is capable of being initialised on any number of inputs, hidden units, or outputs.

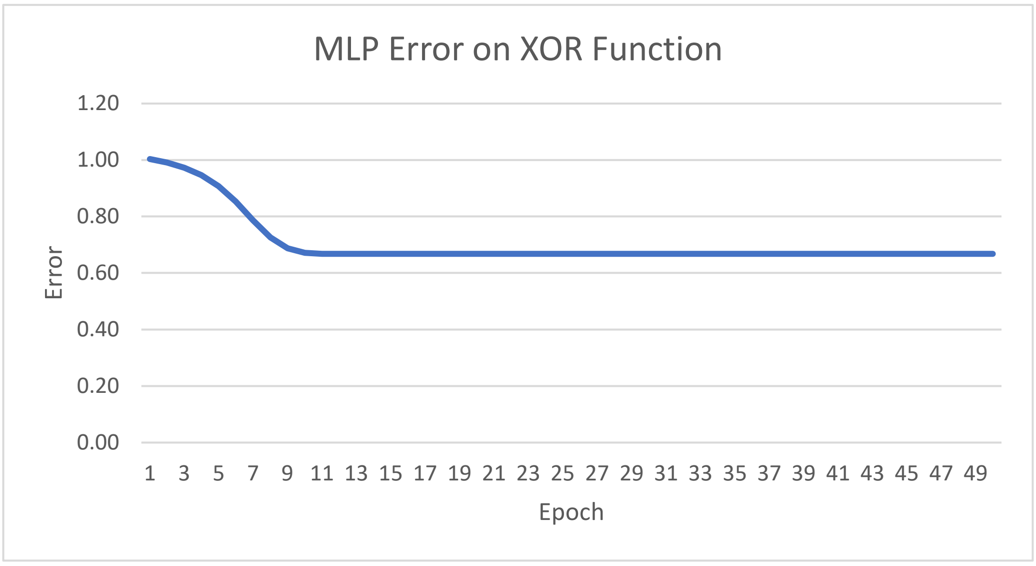
Each of the experiments were written in separate test files labelled 'test1.py' and 'test2.py'. In these test files, after initialising the examples on which the MLP would be trained, the hyperparameters were then set. These hyperparameters are: the number of epochs, the learning rate, the example interval after which the weights were updated, and the number of hidden units in the MLP. Setting the hyperparameters in this way allowed them to be easily tweaked between the various tests.

Two external Python libraries were used in the code: math and numpy. The numpy library was used in both the mlp file and the test2 file in order to created arrays of random values. The math library was used in the test2 file in order to calculate the sin function.

Chapter 3: **Testing and Experiments**

3.1 XOR Function

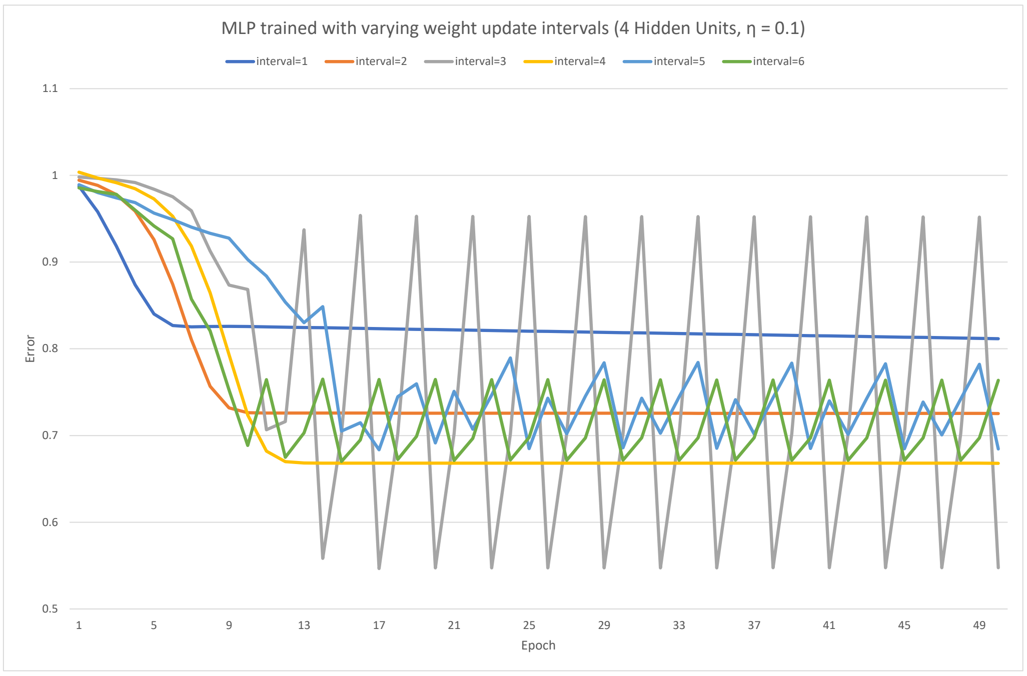
In this test an MLP was trained on the four examples of the exclusive or (XOR) function. An initial test was run with an MLP with 3 hidden units, a learning rate of 0.1, and a weight update interval of 4 examples. The test was run for 10 epochs and the error was found to decrease after each epoch. The test was run again over 50 epochs. From this test it became clear that the decrease in the error began to plateau after epoch 11 (see the graph below) however, the value at which it settled was unsatisfactorily high. The learning rate was decreased to 0.01 however this resulted in a much higher error which showed very little decrease over 50 epochs.



The number of hidden units was increased to 4 and was trained again using a learning rate of 0.1. This time the decrease in the error plateaued after epoch 20 but this time at a slightly lower level than that of the MLP with 3 hidden units. Again, decreasing the learning rate to 0.01 led to a higher error after the same number of epochs.

Another hidden unit was added and again the model was trained with a learning rate of 0.1. Once again the error settled around epoch 20 however this time marginally higher than the model using 4 hidden units.

In each test increasing the learning rate above 0.1 only lead to an increase in the error. The optimal number of hidden units in the MLP for learning this function appears to be 4. Using this information an MLP with 4 hidden units and a learning rate of 0.1 was trained with various weight update intervals, the results of which can be seen in the following graph.

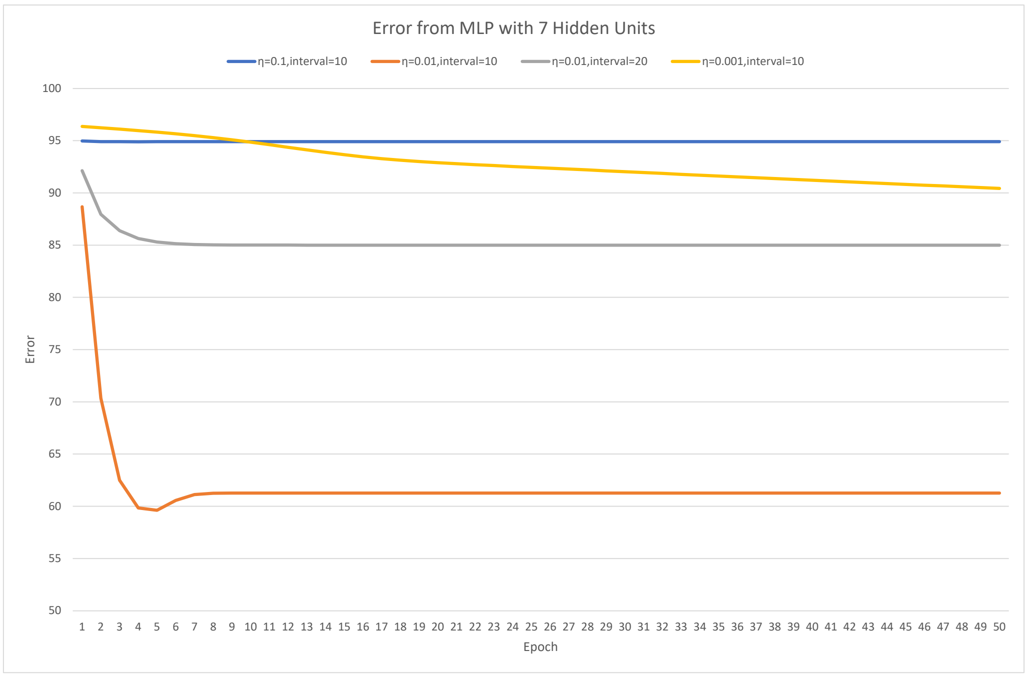


Interestingly, updating the weights after seeing every 3 examples results in the lowest error (0.5476 after 50 epochs), however, the error fluctuates greatly between epochs. The interval which produced the lowest reliable error was every 4 examples as seen by the yellow curve.

3.2 Sin Function

A second experiment was carried out in which the input was a vector of four components and the target was calculated as sin(x1-x2+x3-x4). In each test, the MLP was trained on 400 examples with the error reported after each epoch and then tested on 100 examples with the error here also reported. Using a learning rate of 0.1 and a weight update interval of 10 examples, an MLP with 5 hidden units was trained and tested. Three subsequent tests were executed with an additional hidden unit added for each one. It was found that, in these tests, the MLP using 7 hidden units produced the lowest error. Using 7 hidden units the learning rate was decreased to 0.01 which resulted in a significant decrease in the error. Using the same learning rate the weight update interval was increased to 20 however this lead to an increase in the error.

The errors on the MLP using 7 hidden units can be seen in the graph below.



It is clear that using a learning rate of 0.01 and a weight update interval of 10 produced the lowest error and that the error on this model plateaued around epoch 10.

Chapter 4: **Conclusion**

I found that implementing this MLP was harder than I had anticipated. Some methods were relatively trivial to implement but others, namely the forward and backward methods, proved harder mainly due to the unravelling of the layers of the network. Going forward I think it would be useful to implement a training method which could possibly take in as parameters an array of examples and the 4 hyperparameters that were being tweaked throughout the tests. I believe this would negate the need to rewrite the training code every time an experiment was carried out. Another possible area of work would be in the randomise method of the MLP in which all of the weights in the network are set to small random values. Currently the range of these values is hard-coded as -0.1 to 0.1, however, this could be coded to take into account the number of inputs of the network which may make an impact on the performance of the network.

In terms of training it is clear that each of the parameters that were tweaked during the tests had an impact on the networks ability to learn, and when tweaked in the right way produced a better performing network. The performance was found to be improved by adding 2-3 more hidden units than the number of input units. Increasing beyond this number actually saw the performance of the MLP start to deteriorate. Decreasing the learning rate to around 0.01 also proved to help the performance of the network in each experiment. Again, any level above or below this showed no real benefit to the performance. Tweaking the number of examples after which the weights of the network were update also showed to have an impact on the MLP's ability to learn efficiently. However, unlike the number of hidden units and the learning rate, this seemed to be more of a guessing game in order to find the optimal interval number. It was also found that having a sufficient number of epochs allowed one to see the full behaviour of the error as all eventually plateau at some point.