COMP30230 Connectionist Computing
MLP Assignment Report

Student ID: 18401492
Student: Daniel Gallagher

Introduction

In this assignment, we were tasked with the creation of a *Multilayer Perceptron* (MLP), with a variety of tests to validate it can "learn" data successfully. The MLP is coded in Python and contains an input layer of NI input units, a hidden layer of NH hidden units, and an output layer of NO output units. We will first discuss design methodology and coding decisions, then provide results and discussion on each of the tests. Finally, our learning experience and final thoughts will be provided in a conclusion.

Methodology

In building an MLP from scratch, one must spend a significant amount of time in understanding each separate process and stringing them together mentally. I began by sketching out a very simple MLP with 1 input, 2 hidden units and 1 output. The training data I would use was simply flipping a given bit, so if the input was 0 the output would be 1, and vice versa. Given simple weights, biases, and a sigmoid activation function, I was able to "follow" my data as it would be fed forward throughout the network. For the calculations involving the weights and biases, I refer to these as the affine layer. The output of the affine layer is then passed to the activation function.

Upon receiving the output, we can calculate the 'error' or 'loss'. For instance, the sum of squared residuals. This error is used for the second stage of the iteration in which we move back through the network, finding the gradients which show us how the error changes with respect to individual weights and biases. Here, we must use the chain rule in order to "propagate" our error backwards through the network and find which "way" will reduce our weights and biases. Intuitively, we can understand this as the larger a role an individual weight or bias played in deciding the error, the more it will be changed in the direction which minimises said error. The derivatives are then multiplied by the *learning rate*, which in larger neural networks is generally 0.1, 0.01, etc, however in our MLP will need to be larger due to its small nature. The weights and biases are updated, and the next iteration of the simple MLP would proceed. Understanding this simple example allowed me to gain an intuition for the network as a whole, and how each of the parts connects to one another.

After having sketched out the elements of an MLP and the calculations which are happening at each step, next came the implementation. I would begin with a simple implementation, involving little or no available choice of activation or loss functions. I would then expand this to achieve a greater variety of possibilities.

Language Choice

I decided to use Python to implement the MLP. The *numpy* library is extremely useful and powerful in working with N-dimensional arrays and would make some calculations much easier, especially in finding gradients during backpropagation. Python is also a very common choice for implementing machine learning algorithms when one is not completely focused on optimisation as one would be in an industry context.

Activation Functions
As suggested in the assignment document, I have included both *sigmoid* and *tanh* as the available activation functions. Each activation function is a class which contains a *forward* and *backward* method, with forward representing f(x) and backward representing it's derivative f'(x).

Loss Functions
To compute the error, I have implemented the *L1*, *Mean Squared Error (MSE),* and *Multi-Class Cross Entropy (MCCE)* loss functions. In connectionist computing, it has been clear that the choice of loss function has been incredibly important in determining how well a model will train. In deciding on these loss functions, I took into account the two different types of data my model would be working with. For classifying image data, MCCE would perform particularly well in reducing the model's uncertainty about which letter is the true one. For a regression problem such as the modeling of a sine function, MSE or L1 would work well.

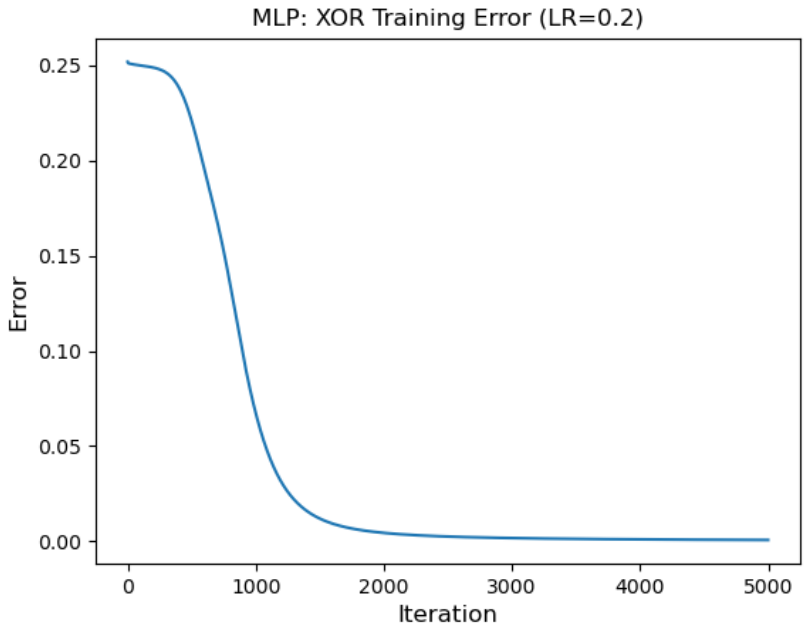
The Multilayer Perceptron
The MLP itself was implemented as a class containing methods for training, testing, and predicting. One needs simply to pass data of the appropriate dimensions to the *train* method and *N* iterations will be completed. It was suggested in the document to iterate through each tuple, however I have decided to handle the input all at once and update the weights after each iteration or epoch. One must first decide the number of inputs, number of hidden units, and number of outputs, that is NI, NH and NO, respectively. One must also choose their choice of activation function and loss function. The training parameters to be adjusted are the learning rate and number of iterations.

Optimisation Strategy
For this assignment I deviated from the given example code and decided to implement batch gradient descent as well as stochastic gradient descent. Batch gradient descent optimises the entire batch at the same time, which is less efficient but guarantees a reduction in loss function each iteration. Stochastic gradient descent iterates one sample at a time, and though not guaranteeing a decrease in the loss function each iterations, in the long run will generally result in good results. I found batch gradient descent to work well for *experiment 1* and *experiment 2*, while stochastic gradient descent worked well for *experiment 2.* The details of the reasoning behind this will be discussed below.

Experiment 1: XOR Function
In this experiment, we must train our model to act as an XOR gate. XOR (or *exclusive OR),* can be thought of as a regular OR gate but must be either A or B and not both. (0,1) and (1,0) will yield 1, while (0,0) and (1,1) will yield 0. The XOR is a particularly interesting gate because it's a non-linear function, that is, it cannot be represented by drawing a line through a diagram. Therefore, the non-linear activations of our MLP will play an important role in appropriately modeling the data.

After testing multiple variations of training parameters, I found that the sigmoid activation function with an MSE loss function was very effective on this data. I decided on a learning rate of 0.2 and ran the model for 5000 iterations. One concern which one would normally

have in training an MLP is overfitting or underfitting, however this is not the case in modeling the XOR as “new” data cannot be created. Therefore a lower error function through more iterations does not bring up issues of overfitting.



We see here the error plotted with the iteration number. The reduction in error at each step plateaus around the 1500 iteration mark as it reaches closer and closer to 0.

Predictions
After training, I fed the possible XOR inputs into the model in order to check the model accuracy, and confirm whether the model has learned successfully to replicate an XOR gate. Below are the results, which can be confirmed through running the script:
Sample: [0 0]
Prediction: 0

Sample: [0 1]
Prediction: 1

Sample: [1 0]
Prediction: 1

Sample: [1 1]
Prediction: 0

The model has correctly predicted all the examples.

Experiment 2: Modelling Sine

Dataset Description

In this question we will be modeling a sin function using our MLP. 500 vectors containing 4 components each are generated with a uniform distribution of range (-1, +1). The target variable is the function sin(x1-x2+x3-x4) applied to each of the four components in the attributes 1-4 (attribute i corresponds to xi). An MLP with 4 inputs, 5 hidden units and 1 output will be used in this question. The training set will consist of ⅘, or 400 of the total set, while the test set will have the remaining 100.

Initial Issues

Of all the questions given to us in this assignment, this one furthered my understanding of vanishing and exploding gradients the most. After feeling confident in my model from the first question, I began training on this new dataset to discover that my weights were not updating. The error each time was remaining constant, and after 500 iterations, I realised something had gone very wrong.

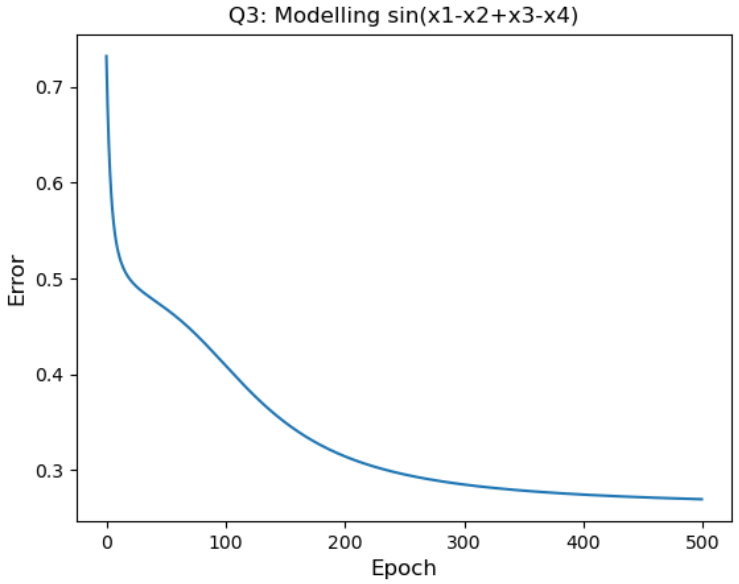
After some investigation, I realised that my weight updates during the weight step were so tiny (approx. 10^-60) that no changes were taking place. Below we can see a small piece of the model during training when the gradient for the second layer of weights is being calculated. Each value below represents the *mean* of the corresponding matrix, not the actual matrix itself.

Computation of dW2 (mean):
X: 0.4987
w: -0.0345
dout: 0.2598
dw = X.T * dout = **52.6341**

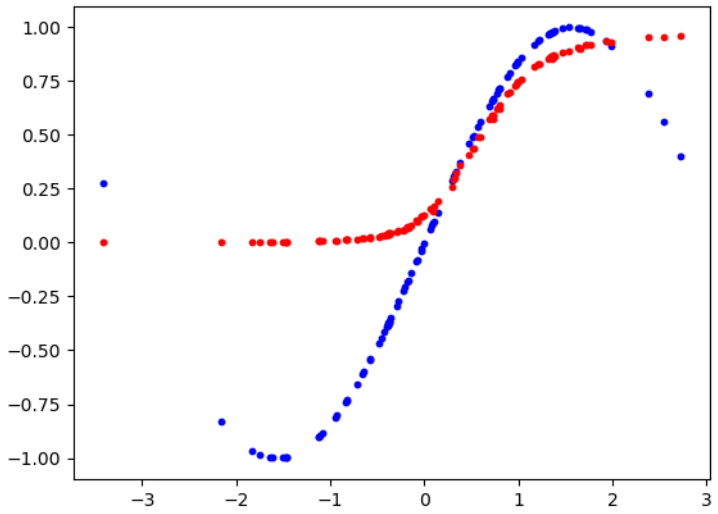
While the input looks harmless, it is the dot product computation here which causes the gradient to explode. The 400 samples were being summed to create a very large output as shown by this mean weight. This is then passed to a sigmoid, where it becomes too small to make any tangible difference when the step occurs. I realised that this was occurring due to my use of batch gradient descent instead, which I had successfully used for the previous example. I decided to implement stochastic gradient descent, in which each sample is iterated individually and weights are updated each time. This fixed my problem of exploding and vanishing gradients, and I learned a lot about when to use various optimisation methods and how this affects the training of your model.

Model Training

I could now begin to train the model. I trained the model using 500 epochs and a learning rate of 0.001. The activation function used for both the hidden units and the output was the sigmoid. We can see the results of training below. The training error decreases rapidly and plateaus, leading to a final training set error of 0.27.



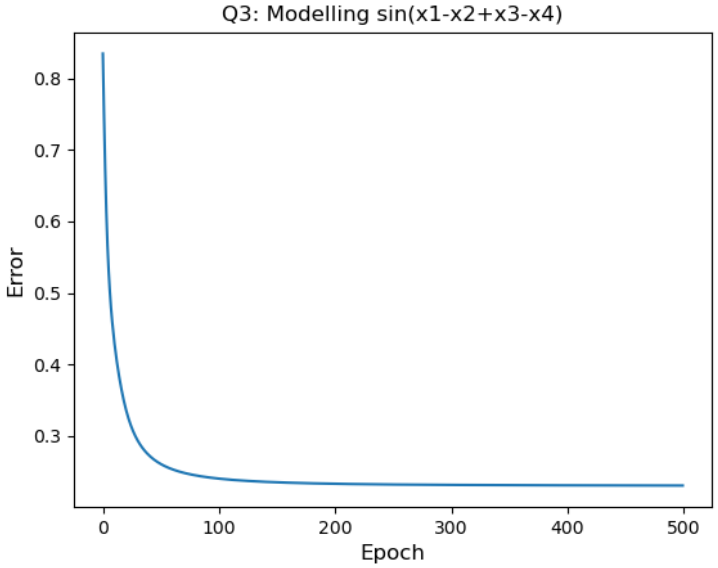
After training, I tested the model on the test set. The test set error was 0.73, much higher than the training set error. In order to investigate further, I plotted the true y values and the predicted y values for the test set. The true values are shown in blue, while the predicted values are shown in red.

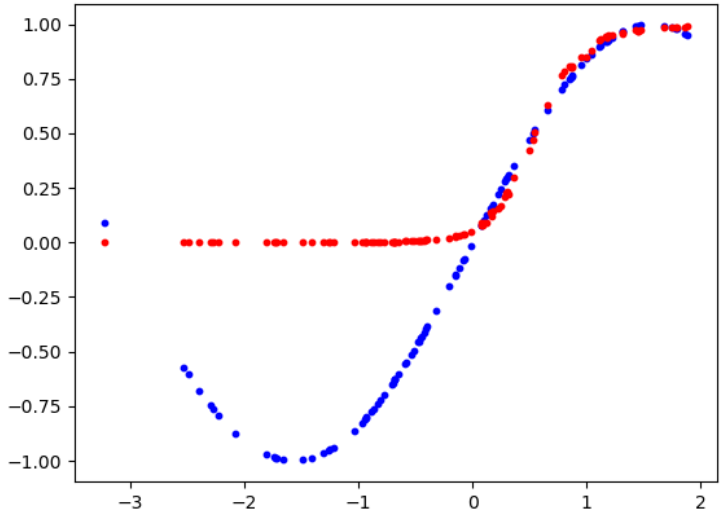


The X axis represents the summation x1-x2+x3-x4, while the Y axis represents the sine function applied when applied to it. We can see that the model accurately predicts values where X >= 0 and is quite poor at predicting values when X < 0.

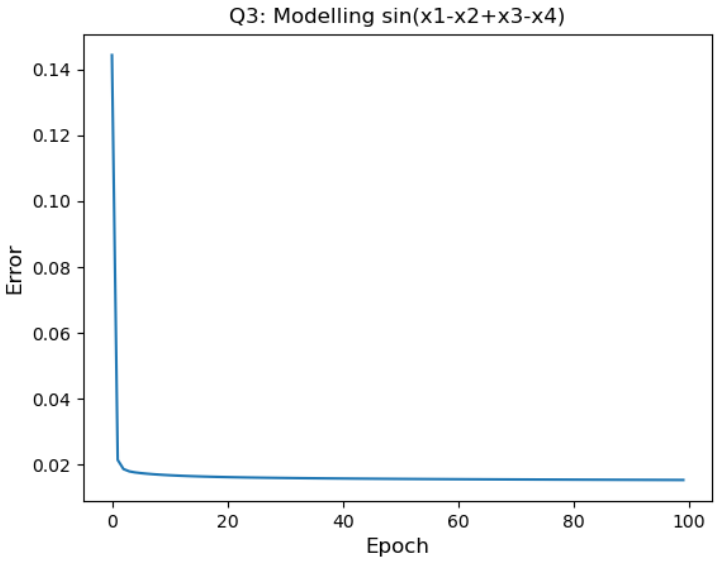
How Can It Improve?

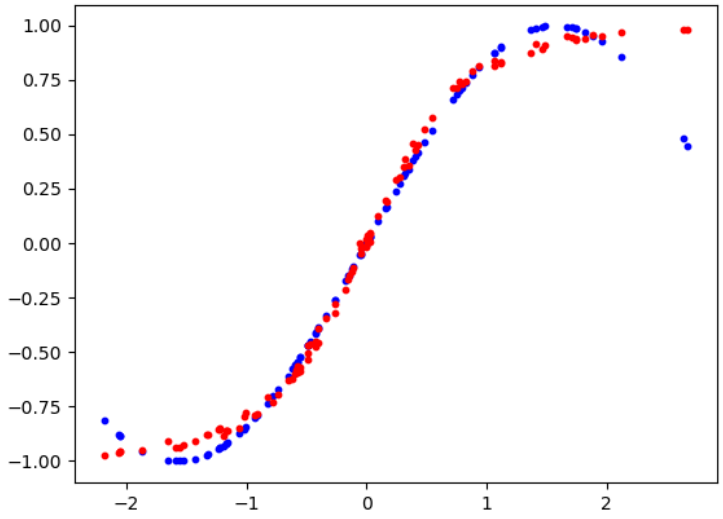
An MLP is very much affected by the choice of activation function. In this case, especially once plotted, I sensed that the weaknesses of the sigmoid function were beginning to show. I decided to experiment with the Tanh function rather than Sigmoid. I will use the same model as above, however it will be trained using a Tanh function on the hidden layer and keeping the Sigmoid on the output layer. We can see the results below.





This didn’t prove the magic solution I was hoping for. However, I had one trick left in my arsenal: a Tanh used on the output. The most interesting insight I gained from this previous training was that the output sigmoid activation could actually be seen in the results on the test set. I decided to use Tanh for both the hidden and output layer with the aim of improvin this accuracy when x < 0.





What do you know! This was the solution I was looking for. We can see that the Tanh allows the training set error to reduce dramatically to a small 0.02 and the predictions on the test set have become accurate when x < 0. Finally, I believe that the model has learned satisfactorily. Though this question took significantly longer than expected due to exploding and vanishing gradients, I felt I learned much about the MLP through these difficulties. Finally, I had one beast left to tackle: the *UCI handwritten letters.*

UCI Handwritten Letter Set

Dataset Description

This special test involves the use of the UCD Machine Learning repository for handwritten letters. There are 16 attributes as well as the target variable, each attribute representing a particular feature of the original image. These attributes are all of type integer, and the target variable is a character corresponding to the actual handwritten letter. There are a total of 20 thousand samples in the dataset.

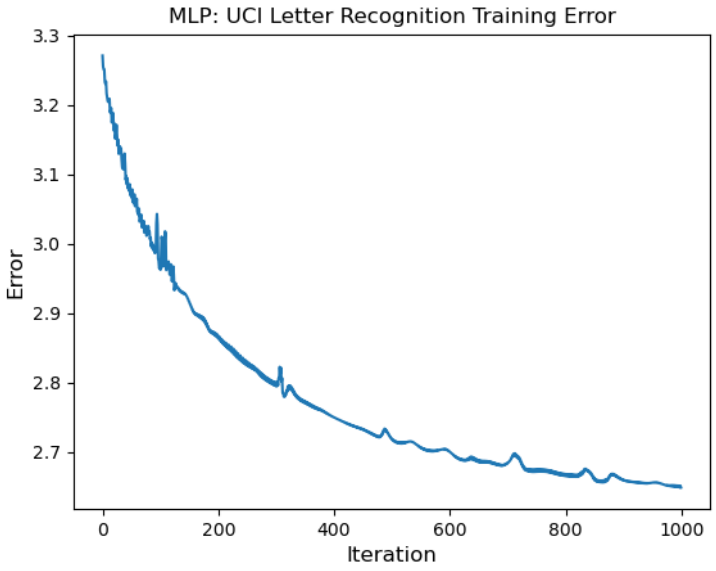
Training

We will split the dataset into training and test sets. The training set will contain ⅘ of the original set, while the test set will contain the remaining ⅕. This corresponds to 16000 training images and 4000 test images. The attributes had to be one-hot encoded in order to be fed as integers into the MLP. The target variable was also encoded, however, with each letter being represented by an integer.

MCCE was the most suitable loss function in this case as it is particularly effective at calculating loss for non-binary classification problems. The activation function chosen was the sigmoid. I tested multiple sets of training parameters. In this learning problem, overfitting and underfitting was possible (unlike with XOR), therefore more iterations didn't necessarily mean higher accuracy. We can see the results of multiple configurations below.

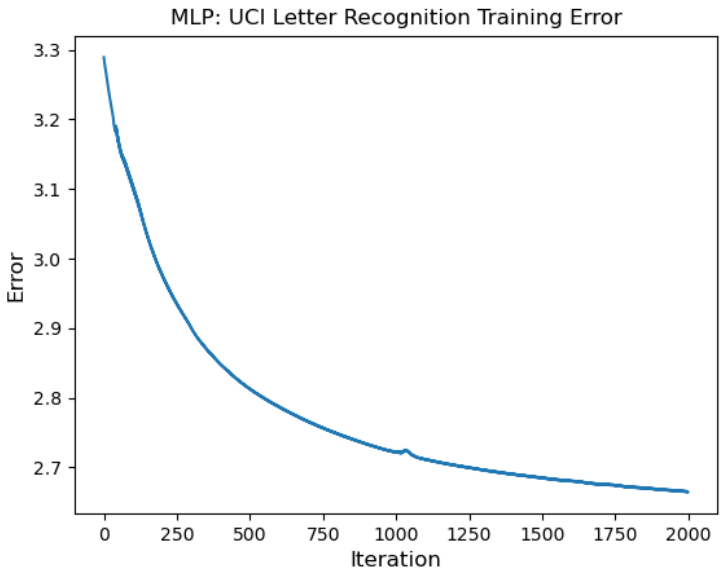
Result 1

Iterations: 1000
Learning Rate: 5
Hidden Units: 10
Test set accuracy: 52%

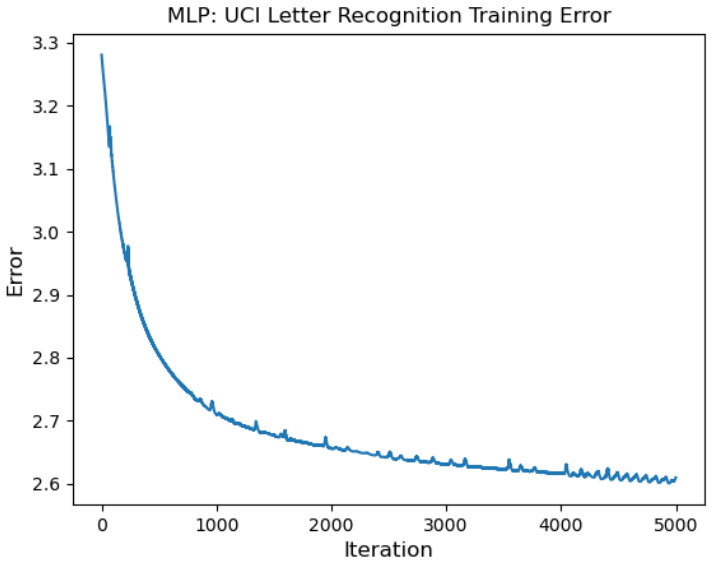


Result 2

Iterations: 2000
Learning Rate: 2
Hidden Units: 10
Test set accuracy: 43%



Result 3
Iterations: 5000
Learning Rate: 3
Hidden Units: 10
Test set accuracy: 46%



Discussion of Results
The greatest accuracy achieved by the model on the test set was 52%, with 1000 iterations, a learning rate of 5, and 10 hidden units. In binary classification, this would be a very poor accuracy (only slightly better than a random guess), however in a classification problem such as this where there are 26 classes, it is a relatively accurate model. The likelihood of a random guess being correct would be about 0.04. Results 2 and 3 had altered parameters

but achieved slightly worse accuracy scores at 43% and 46%. These trained for 2000 and 5000 iterations, respectively. This is an example which shows that more iterations does not equal better accuracy on the test set, but only on the training set. We must always generalise our model on the training set so that it doesn't become a *training set* predictor rather than a *real-world* predictor. I believe that the MLP has learned satisfactorily achieving a score of 52% on the test set, especially given the difficult dataset and the constraints of a small model.

Conclusion

Though a difficult assignment, I found this to be one of the most enlightening in my time at UCD. Building an MLP from scratch is analogous to teaching students C before Python, it teaches you how and why much of machine learning works the way it does. Until now I felt I had a solid grasp on the effectiveness of many machine learning models, however forcing myself to understand backpropagation and the exact steps involved in training an MLP has given me an entirely new understanding of these models and has brought a new confidence to my understanding of machine learning in general.

Before any experiments were run, there was (what felt at the time to be) the mammoth task of programming a multi-layer perceptron from scratch, having only *numpy* to save me from my woes. One of the biggest learning points I have taken home from the building of an MLP is: dimensions matter! I began building this project with a clumsiness in tracking the dimensions of everything that was moving through the layers, however this is what caused a bottleneck in the project as I realised how necessary it was to apply transformations. *Reshape, reshape, reshape* I now repeat through my head if building an MLP. It was for this reason that I started this project twice, the second time humbled and with careful attention paid to the exact dimensions of every matrix moving through the MLP.

I found the three different objectives to each have unique potential for learning. The first question was an excellent way to begin, as an XOR gate is important and widely understood by us as computer scientists, and is as simple as a non-linear problem can be. This allowed me to gain a clear understanding of the difference between non-linear and linear models, when put in terms of gates and plotted it made quite a lot of sense to me. Additionally, there was no potential for overfitting in the first question which was a good way to get the model off the ground.

Modelling the sine function brought with it the problem of vanishing and exploding gradients, which up until this assignment I had understood abstractly but had failed to see practically and in detail. Solving this issue brought me through almost everything we have learned in this module and I found this frustrating difficulty became one of the biggest learning opportunities for me during this module. There was nothing like the satisfaction of seeing the weights updating and the error reducing after changing optimisation strategy!

The final question had the most complex dataset and was the 'special test' of this assignment. Handwritten letter recognition is the foundation problem of those aspiring to work in the realm of machine learning and I learned a lot from using this real dataset and training my model on it. This question took a lot of parameter adjusting in order to achieve

the results I wanted, with multiple different configurations making it to the ‘final cut’ of this report. I learned a lot more about fine-tuning than in the previous smaller-scale experiments.

Going forward, I feel much more confident in working with connectionist models and have learned a huge amount through this assignment and module.