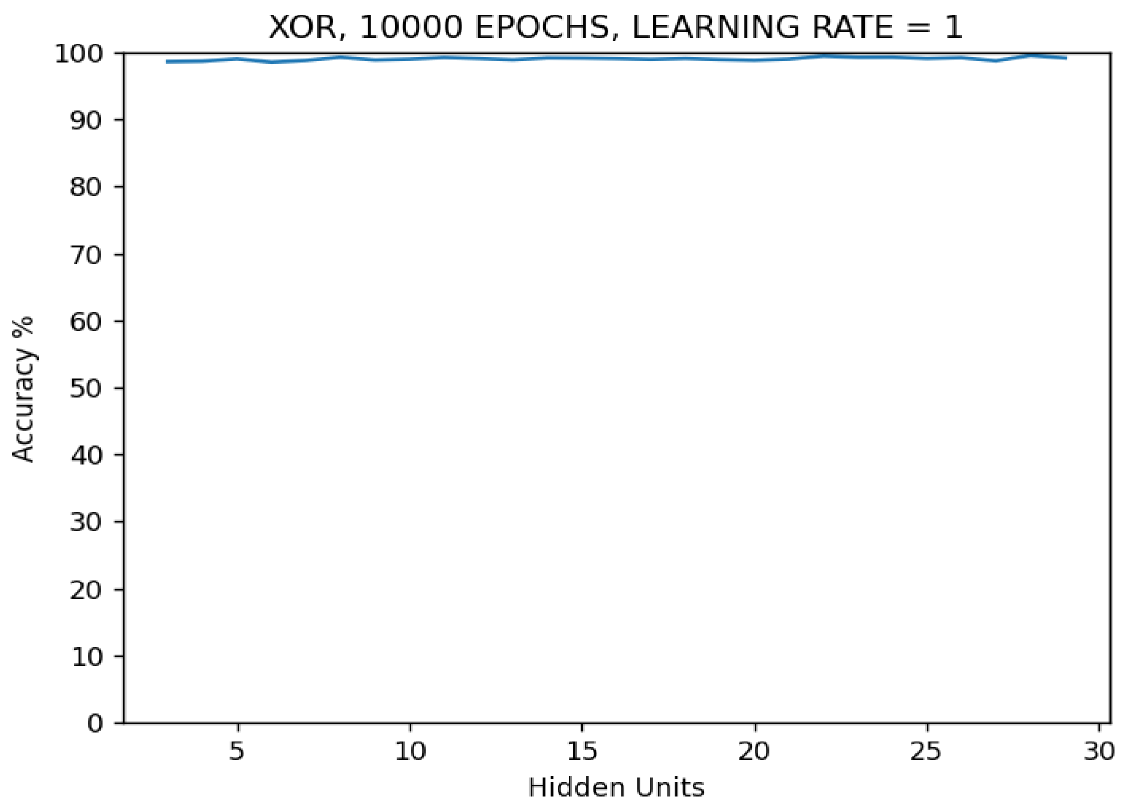
Connectionist Computing Programming Assignment

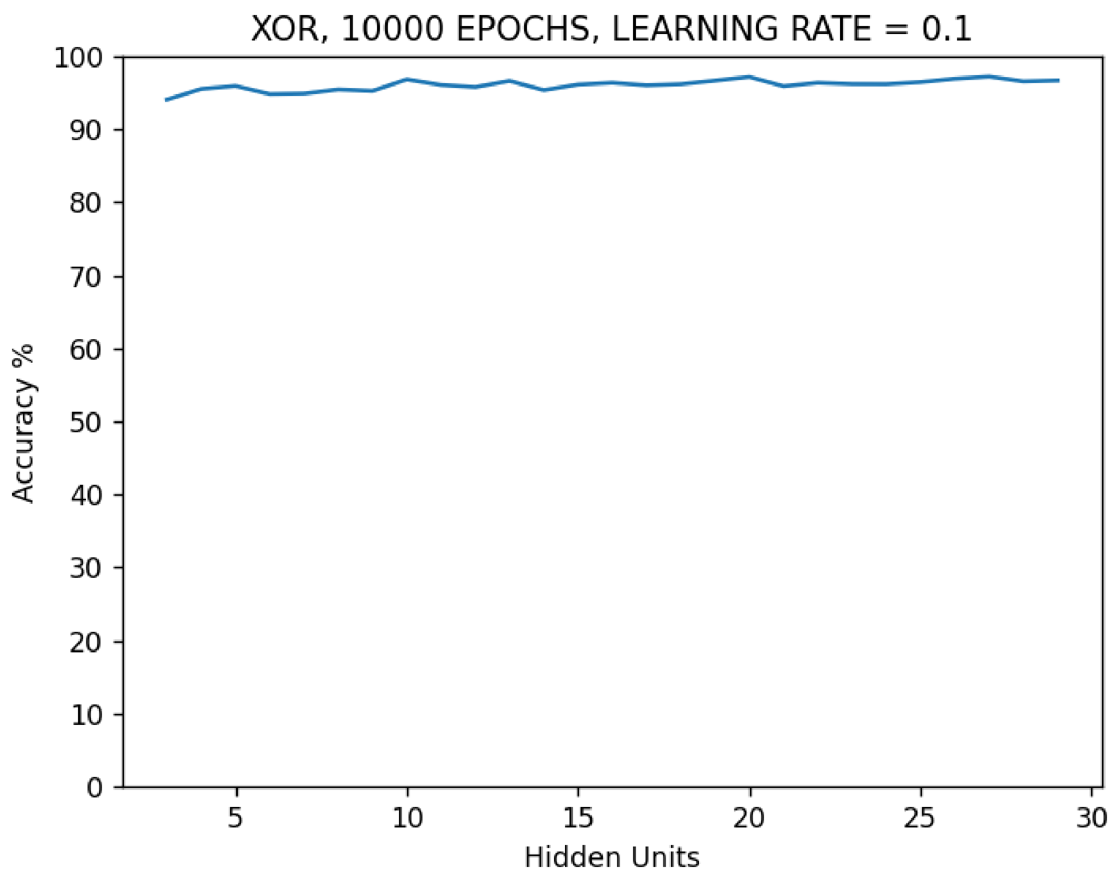
I was tasked with creating a MLP (Multi-Layer Perception) for 3 questions. I had to train it to learn the XOR function, sin operation on a vectors and letter recognition.

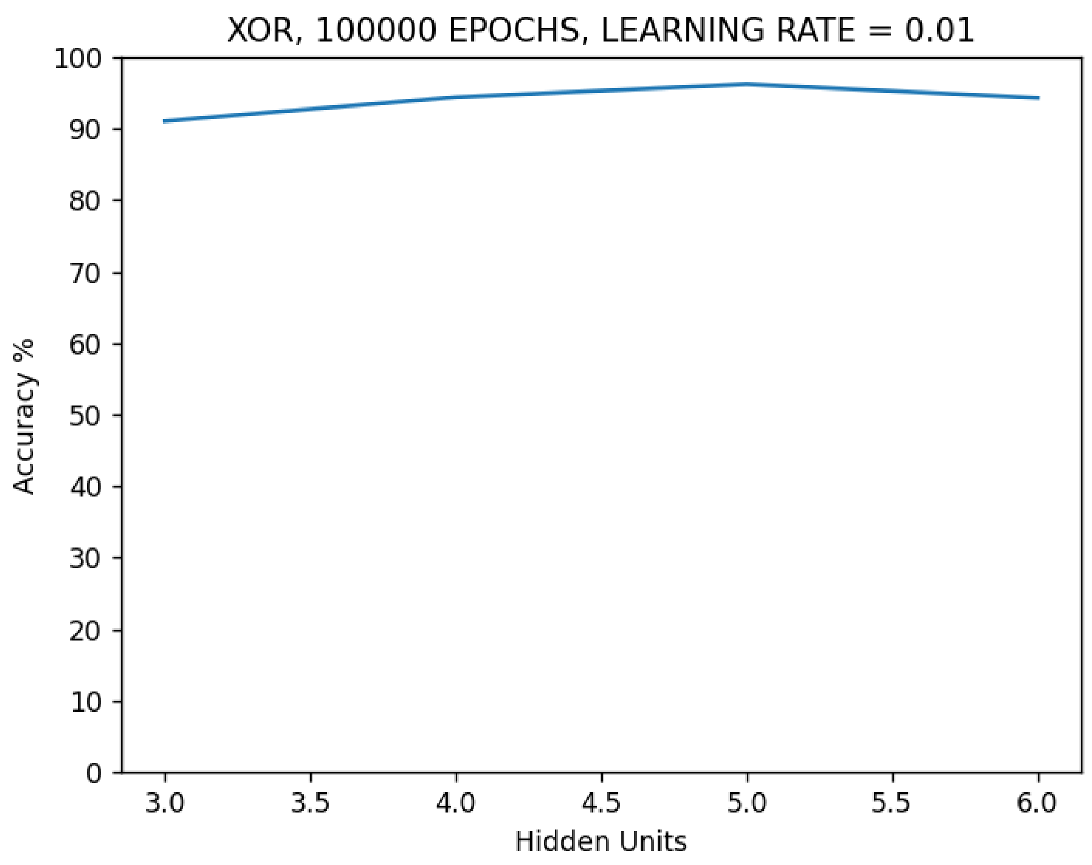
XOR Function

For the XOR function, my MLP took in 2 inputs representing the x XOR y, and then returned 1 output, which was the result of the XOR. I was left with 29 hidden units with a learning rate of 1 after adjusting these parameters. This resulted in an accuracy of 99.2%. I used a sigmoid activation function. This helped the model learn accurately by having neurons output a value between 0 and 1.

Issues I faced with XOR was that my values would converge to 1 or 0 or stop at 0.5. When adjusting parameters, I had a working XOR function for the first 3 examples (0, 0) (0, 1) and (1, 0) however my last example (1, 1) resulted in 0.5 output. I didn't realise that by adjusting these values it would output correct answers for all outputs apart from 1, I thought by adjusting learn rates everything would learn consecutively.




XOR, 10000 EPOCHS, LEARNING RATE = 0.1
Accuracy %
Hidden Units
0
10
20
30
40
50
60
70
80
90
100
5
10
15
20
25
30



10000 epochs were too little for training 0.01 so had to increase the epochs to 100k. I didn't test any smaller learning rates, because 0.1 and 1 were more efficient.

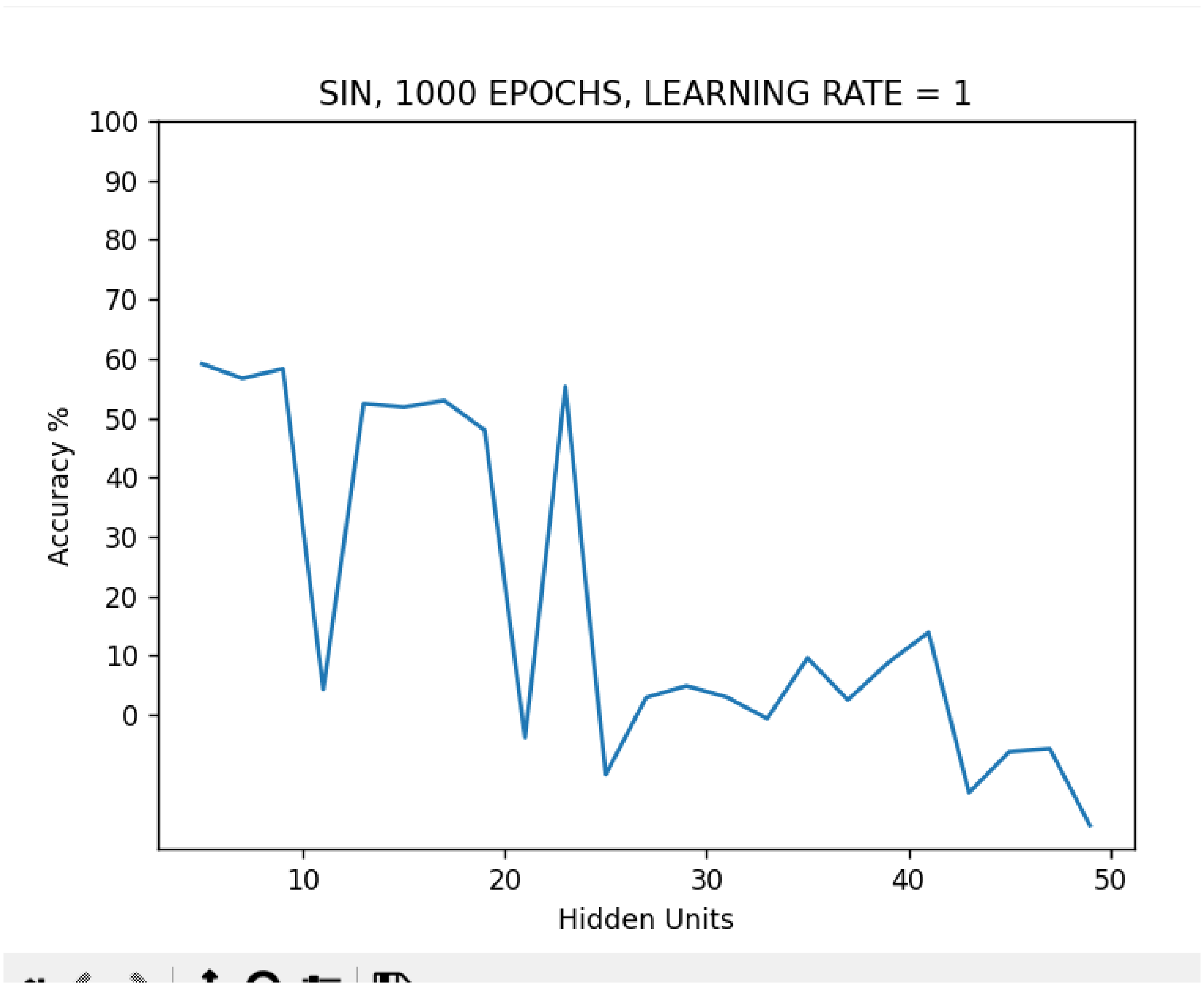
SIN Function

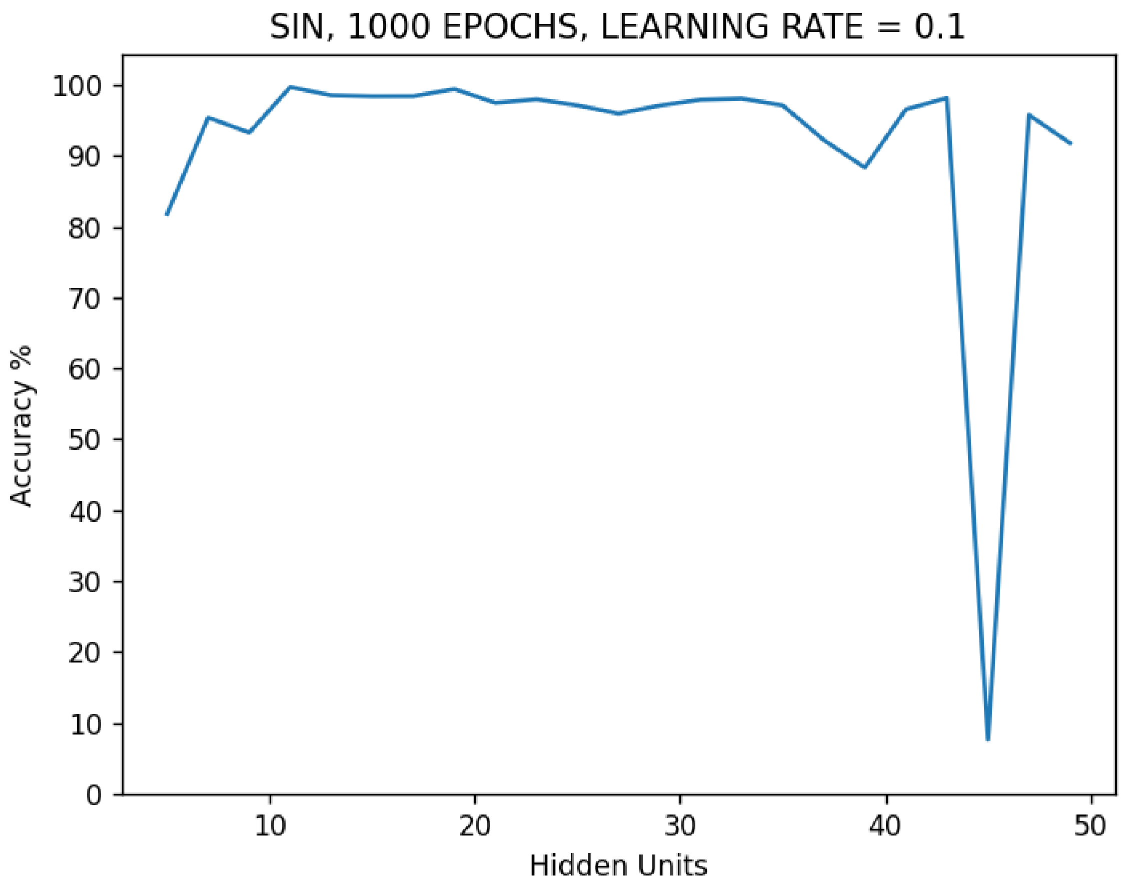
For SIN, I had to generate 500 vectors with 4 integers in each vector being values -1, 0 or 1. I then used the sin function on this as follows:

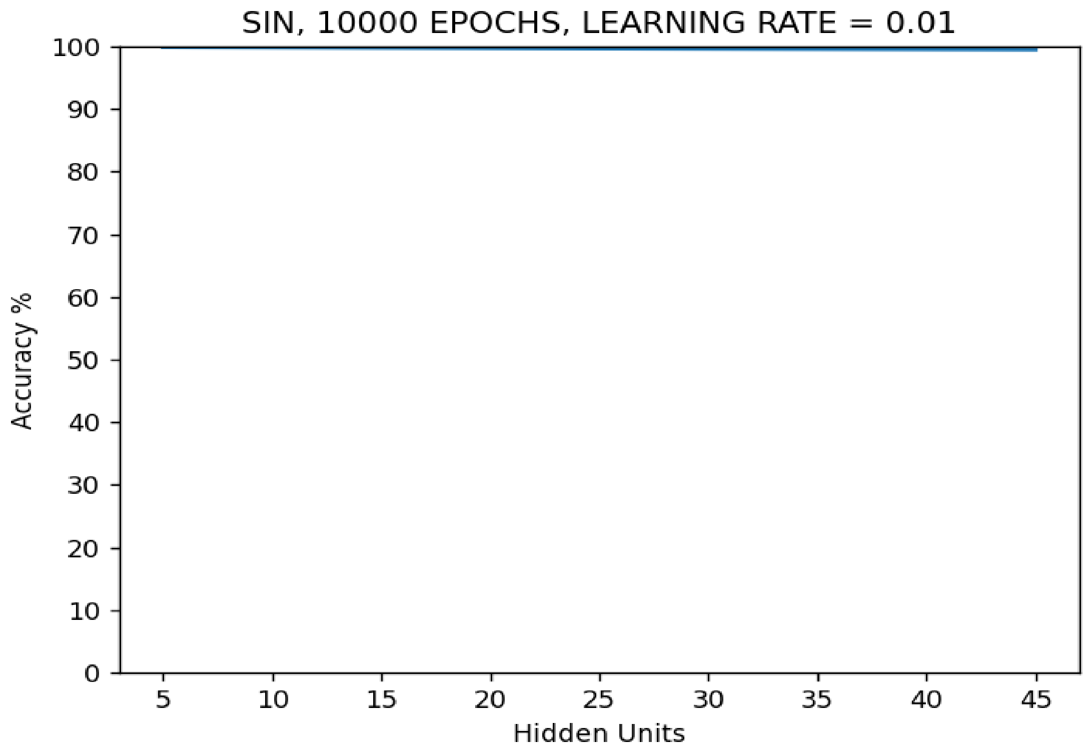
Sin(x1 – x2 + x3 – x4)

my MLP took in 4 inputs and 1 output. I used sigmoid initially, however this resulted in being trapped in local minima when running, and not so accurate results. So I decided to use the tanh function for the neurons because the tanh activation function results in higher values of gradient during training and higher updates in the weights of the network, which sigmoid was incapable of doing.

45 hidden units, 10000 epochs, 0.01 learning rate, resulted in 99.7% accuracy




SIN, 1000 EPOCHS, LEARNING RATE = 0.1
Accuracy %
Hidden Units
0
10
20
30
40
50
60
70
80
90
100
10
20
30
40
50


SIN, 10000 EPOCHS, LEARNING RATE = 0.01
Accuracy %
Hidden Units
0
10
20
30
40
50
60
70
80
90
100
5
10
15
20
25
30
35
40
45

From the above graph its hard to see, but 99.7% accuracy occurred at this level. 45 hidden units for at 0.01 learning for 10000 epochs had the highest accuracy. I was surprised to see such a high accuracy, I thought I would only achieve 97% or 98% but this shows the power of a MLP.

Letter Recognition

For letter recognition, we were provided 20000 examples of letters, and 16 values representing these values attributes. This data is split into 4/5 for training and rest for testing. My MLP had 16 inputs and 26 outputs, each output representing letters and each input representing the letters attributes.

I used sigmoid for this as it had a slight advantage over tanh.

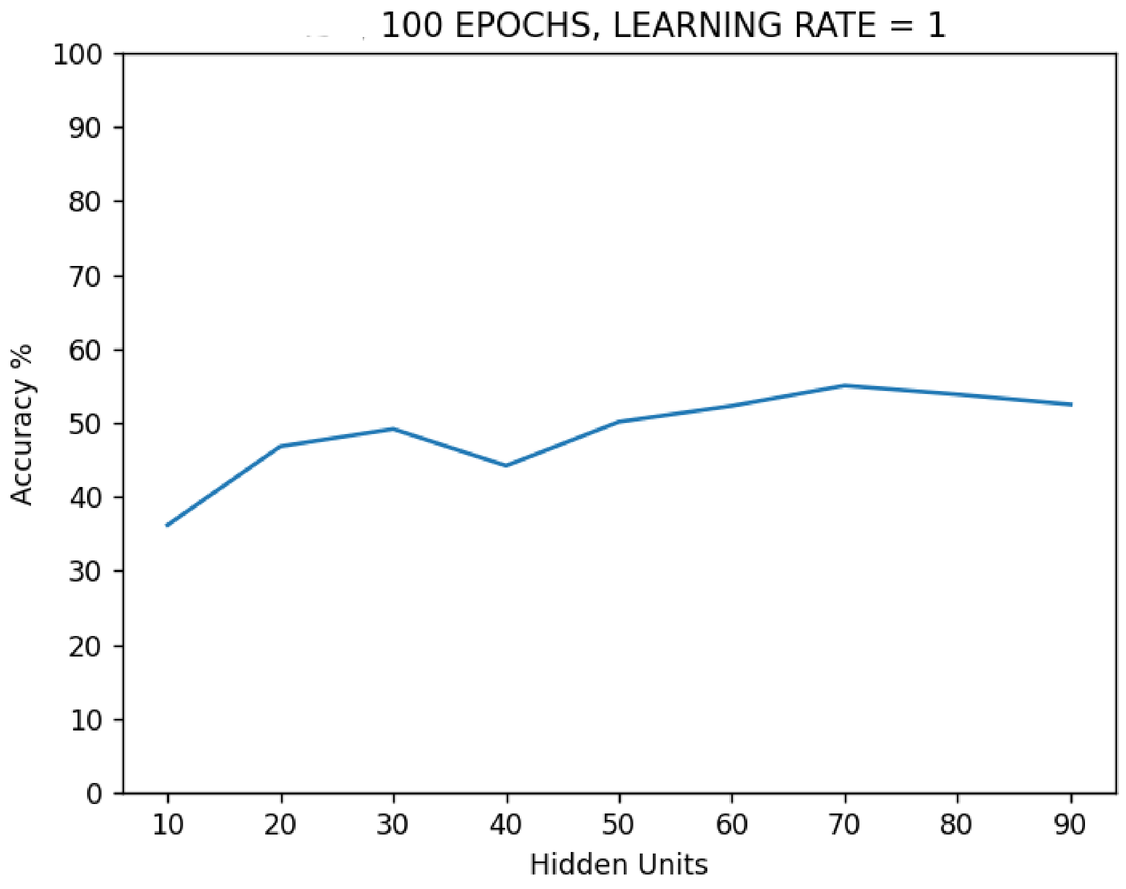
The inputs provided were from 0 to 15, and this value didn't work well with my MLP because I needed to transform features to be on a similar scale as the model**.**so I decided to use min max normalisation to allow values to be between 0 and 1, to benefit the model. Because the smallest value was 0 I divided all values by 15, but then realised that not all letter attributes contain 0 or 15, so I then got the maximum and minimum of each letter attributes and normalised it that way. This resulted in a ~3-4% increase in learning.

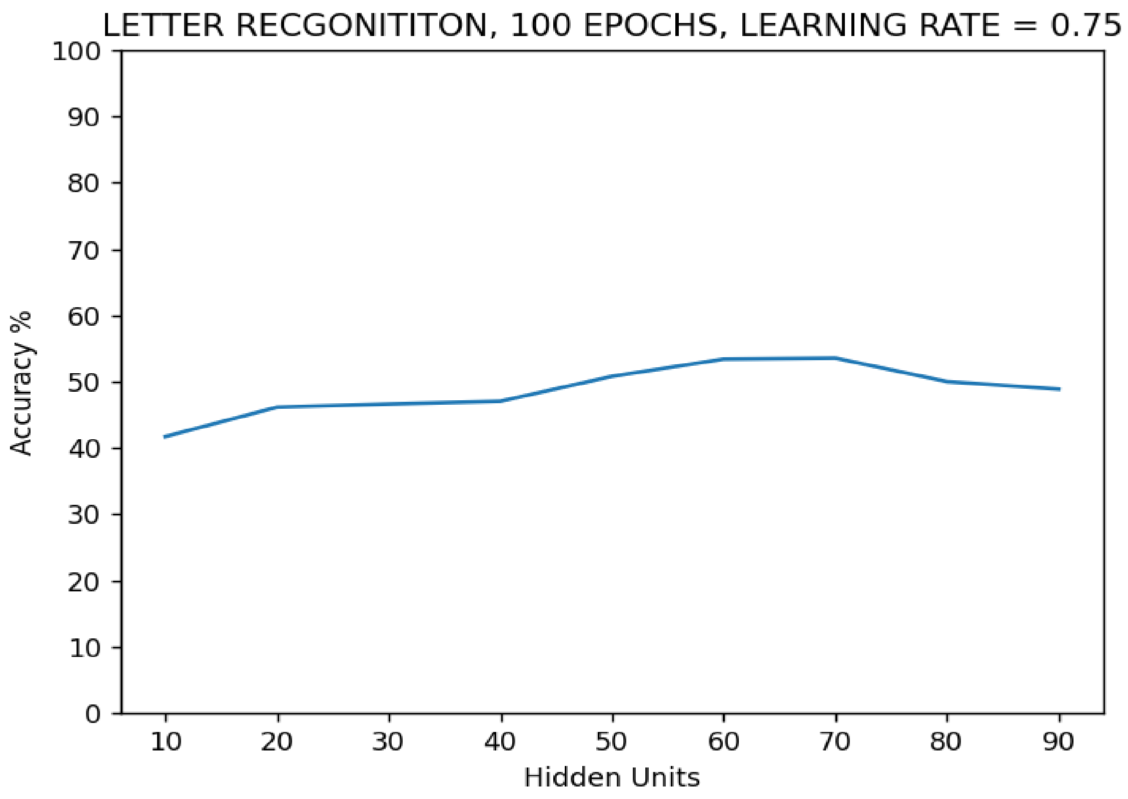
(X – min)/ (max – min)

First issue, involved my first letter getting recgonised and every other letter not. This was because my backpropagation only learned off of the first output, and not the other 25 outputs, this first output represented the letter A, so the letter A was correctly predicted, and nothing else. I only encountered this error here because the first 2 questions only had a single output, so this wasn't an issue for them.

Ran for 10k epochs at 0.01 and only learned ~49%, a larger learning rate is more effective, I would've thought smaller learning rates would be better for larger inputs and outputs, but this shows larger was better.

60.050000000000004% accuracy 400 Hidden units, 30 epochs, 1 learning rate was the highest accuracy I received. I tried adjusting these parameters to find a more accurate answer, but this was the best I got; However, it wasn't consistent. In the file letters1.txt were some tests I had ran, with different parameters to show the range in accuracy.





```
49.25% 0.01 learn rate, 30 hidden units, sigmoid
31.95, 0.01, 30, tanh
50.24%, 0.1, 30, sigmoi, 1000 epochs,
47.675%, 0.1, 30, sigmoid, 100 epochs,
52.125 0.1, 30, sigmoid, 1000 epochs,
52.175000000000004 50 hidden units 1000 epochs , 0.1 learning rate ^
53.974999999999994 30 hidden  2000 epochs, ^
47.699999999999996 30 hidden units, 10000 epochs, tanh 0.01 learning rate
52.37500000000001 30, 300 0.3
55.50000000000001 100, 300 epoch 0.3 learning rate
51.87500000000001 100 hidden units, 300 epoch, 0.5 learning rate,
56.225 100 hidden units, 500 epoch, 0.3 learning rate
56.35 50 hidden units, 500 epoch, 0.3 learning rate
50.075 50, 500, 0.5
56.025000000000006 100 hidden units, 1000 epochs, 0.1 learning rate
51.425 100 hidden units, 2500 epochs, 0.1 learning rate
51.0 100 hidden units, 1000 epochs, 0.05 learning rate
45.7 100 hidden units, 100 epochs, 0.05 learning rate
54.0 100 hidden units, 250 epochs, 1 learning rate
55.65 100 hidden units, 100 epochs, 1 learning rate
60.050000000000004 400 Hidden units, 30 epochs, 1 learning rate
```

These were different results I got from adjusting parameters, and it seems the most consistent was 100 hidden units 1000 epochs and 0.1 learning rate. This was off my own prediction and trying to find the perfect parameters.

MY MLP

My MLP was coded in python, this was due to python’s simplistic style and the library NumPy, this library helped assist the runtime due to the way it runs computations for vectors and matrices. I initialise the number of inputs, hidden units, and outputs. This network has 1 hidden layer. I then randomly initialise my weights and biases. My forward pass then multiplies these inputs by the weights and add the biases. Then I get the activation function of these values to see if the neuron fires, either sigmoidal or using tanh. This calculates output, so then I do the backwards pass, which calculates the error and then the delta weights. The error is calculated by getting the absolute value of the predicted output and the actual output (distance from the actual output). This is then multiplied by the derivative of the activation functions output to get a vector of the changes in the layer. This vector is summed together by rows and dimensions are kept the same to result in the change in biases. And for the weights it is multiplied by the hidden layer to get the change in weights. This change in layer matrix is multiplied by the weights from the hidden to output to result in a value to calculate loss for the inputs to hidden layer. This is then multiplied by the activation function derivative to get the change in weights from inputs to hidden layer. And again, the error in layer is summed up to get change in biases. The weights and biases are updated by adding the weight and biases difference multiplied by the learning rate to the current weight and biases.

Conclusions

XOR

I was surprised to see that a large learning rate would result in efficient results. The hidden units didn’t seem to affect the results except for 0.01 learning rate, where you can see a clear point where extra hidden units diminish the results. This was interesting considering I would've thought extra weights and biases would improve a network, but I assume that it's over complicating the network and minimal units will be enough to handle the inputs. When I ran for minimal epochs, I would receive results where I would get decent results, but 1 unit would be 0.5. This interested me because I assumed everything learned in parallel, but it shows if not properly trained with the right parameters, some values aren't fully learned while others are.

SIN

With sin I learned about getting stuck in local minima. When I used sigmoidal this approach wasn't efficient, so I decided to switch to tanh. This improved significantly. I also realised that large learning rates aren't as efficient for everything, like it worked in XOR, but not for SIN. Each question has a unique set of parameters to learn effectively. I also learned how close a network can predict, considering I got 99.7% accuracy. Also, at around 45 hidden units you can see a big drop in accuracy. This taught me that sometimes networks can't be 100% trusted because they can still cause inaccuracies. I was unsure why this happened, but it appeared fine for the other values. Large learning rates were also not efficient for this problem, as you can see the low accuracies of having learning rate at 1, so I know to check all levels of learning rates to find maximum efficiency.

Letter Recognition

There was a lot I learned from the Letter Recognition. Firstly, I learned that values must be normalised to fit the data, and this is what first kickstarted my MLP to work. Because the values being dealt with are between 0 and 1, so should be inputs. I also only trained off the first output by not accounting for all output neurons. I learned that increasing epochs does not always result in better loss calculations. Using my model, when I increased the epochs, I would drop sometimes 4-5% accuracy. I assume that my learning method mustn't have been as efficient as it needs to be, but from what I was using I got to a

60% accuracy, so I was happy with this outcome, considering initially it only worked 4% of the time, which was random assignments. Large hidden unit values seemed to work effectively but not cause major improvements in the accuracy. And I learned that low learning rates don't do well either. Values ranging from 0.1 to 1 seemed to have best effect on the accuracy.

What I found most interesting out of this project was how there is no certain way to set parameters such as learning rates, hidden units, and epochs for any question, it all must be trialled and tested until the good accuracy is found.

references

NORMALISATION

https://developers.google.com/machine-learning/data-prep/transform/normalization#:~:text=The%20goal%20of%20normalization%20is,training%20stability%20of%20the%20model.

BUILDING NEURAL NETWORK FROM SCRATCH

https://youtube.com/playlist?list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3&si=fYKDYj9kKWVlzC7F NNFS

XOR ARTICLE

https://medium.com/analytics-vidhya/coding-a-neural-network-for-xor-logic-classifier-from-scratch-b90543648e8a