COMP30230- CONNECTIONIST COMPUTING

Programming Assignment

18391933 – Brian Byrne

Coding the Multi-Layer Perceptron

For the purposes of this assignment, I used the python programming language, as I am currently a data science student, who already deals with a lot of classification methods in python. Despite the fact we were prohibited from using any deep learning packages like TensorFlow or Sci-kit Learn, I still found python to be ideal for dealing with the various matrix and vector computations necessary for constructing a neural network from scratch. This was due to the fact the "*NumPy*" library has a large array of matrix and vector manipulation functions, making the code easier to implement and read. To add an element of abstraction and for ease of interpretation, I used object-orientated programming, creating a generic multi-layer perceptron.

The multi-layer perceptron followed the layout given in the assignment brief. It contained 3 instance integer variables for the number of inputs, hidden inputs, and number of outputs. We also had instance NumPy arrays representing the weights of each layer, changes to these weights, activations for each layer, an array for the values of the hidden neurons and an array storing the output values of the multi-layer perceptron. As a generic multi-layer perceptron, I added a constructor that takes the number of inputs, hidden inputs, and number of outputs as an argument, as these differ from function to function. Additionally, knowing that the XOR function has sigmoidal outputs [0,1] and that the Sin function has a hyperbolic tangent nature [-1,1], I allowed for each instance of multi-layer perceptron to select how it's output was calculated using functions for sigmoid and tanh, as well as separate functions for their derivatives, which are used in the backpropagation method.

The randomise method Initialises W1 and W2 to small random values and dW1 and dW2 to be all zeroes, providing structure for our weight arrays. Forward propagation is quite straight forward, here it takes in an input and produces an output, passing through the hidden units. Backpropagation is a little more complex. This learns which weight is the best, checking by how much a prediction deviated from the actual output and returning this figure as

fraction. Together when placed in a loop with the updateWeights method, these functions creep towards the correct solution.

The XOR Experiment

XOR takes 2 inputs and produces 1 output based on the exclusive-or properties i.e. true if the 2 inputs are not equal as can be seen here.

```
[0, 0] = [0]
[0, 1] = [1]
[1, 0] = [1]
[1, 1] = [0]
```

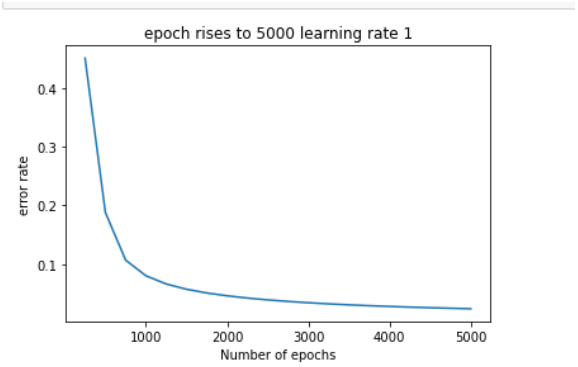
To identify patterns of behaviour in the function, I looped parameters through a the XOR function, using varying maximum sizes for epochs and various learning rates also. I structured the function to run on the multi-layered perceptron that I explained earlier, first pre-training the model by running the forward propagation algorithm to demonstrate the fruitless results produced by an untrained model.

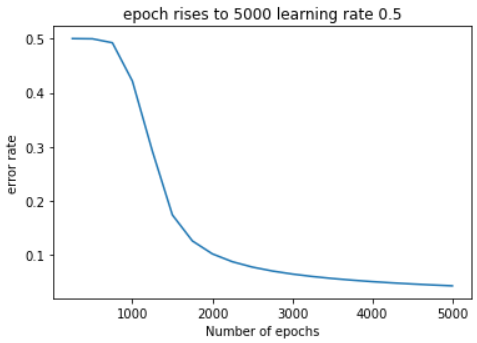
```
Before Training Analysis[0 0]Expected: [0]      Output: [0.70353385]
[0 1]Expected: [1]      Output: [0.76595564]
[1 0]Expected: [1]      Output: [0.7509969]
[1 1]Expected: [0]      Output: [0.79751755]
```

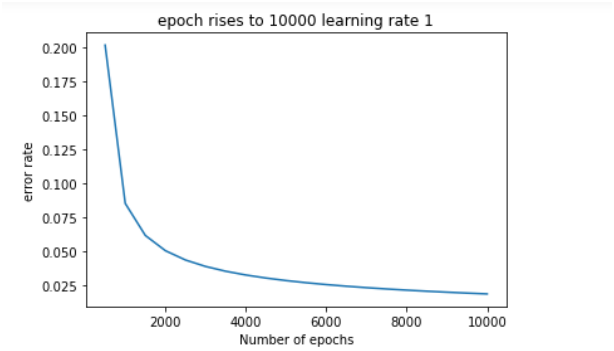
As expected, the model has preformed poorly and cannot identify how to distinguish outputs from the given data.

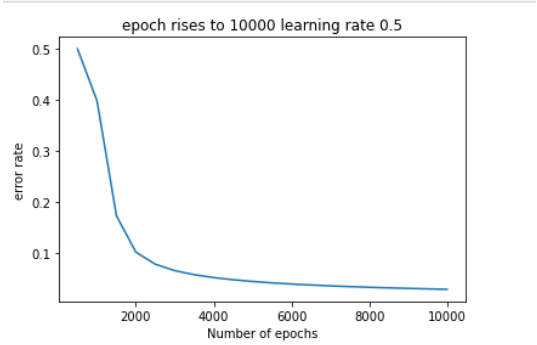
The next step involved training the model. I chose to train the mechanism without splitting training and test data, as the goal of this experiment is to check if the MLP predicts correctly all the examples, and also that in such a small data set, it is clear that the model couldn't learn near adequate-enough amounts to accomplish its goal. Training involved forward propagation, followed by backwards propagation to calculate errors and to update the weights in regard to the learning rate. Here I was able to simultaneously print and write, the change in error in relation to epochs. Finally, I tested the MLP's XOR function, by running forward propagation to show how much our model had learned.

XOR Analysis









From our experiments here we can see that an increase in learning rate produces initial better results, as well as faster and more reliable learning regardless of the maximum number of epochs training the model. The elbow-shaped curve appears in all graphs, however it is still evident that as expected, the more we train a model (epochs), the better the model performs.

After training, all of the trained models can distinguish between outputs in a binary fashion (round -to-nearest), as you can see from the best performing model where max epoch = 10000 with a learning rate of 1, predicting very clear and accurate results :

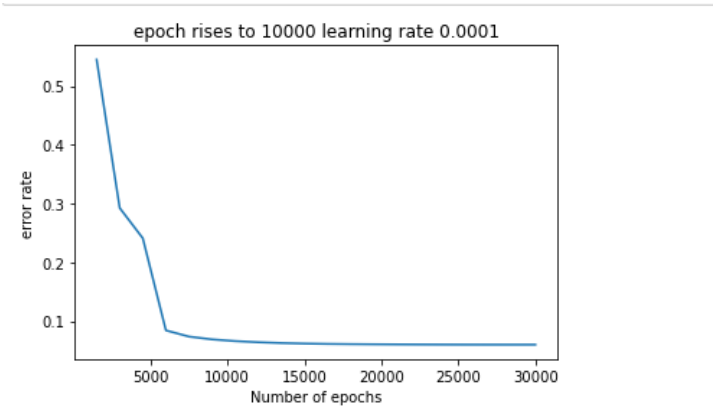
```
Testing the Perceptron
[0 0]Expected: [0]      Output: [0.02921052]
[0 1]Expected: [1]      Output: [0.98109777]
[1 0]Expected: [1]      Output: [0.98137398]
[1 1]Expected: [0]      Output: [0.00791995]
```

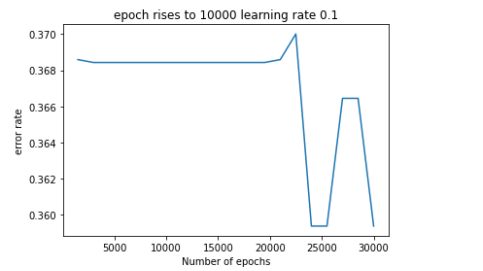
The SIN Experiment

The Sin experiment was conducted very similarly to the XOR experiment. We populated the data set here producing 500 vectors, each containing 4 values ranging between -1 and 1. We calculated the actual outputs using sin(x1-

x2+x3-x4) for each vector and stored them. From this method we know that we have 4 inputs mapped to one output. I learned whilst doing this experiment that a large number of hidden units, greatly increased the accuracy of our model. Due to the much larger complexity and computational cost of the Sin function, I decided to pass only varying learning rates as arguments in our MLP. Finally, I implemented an 80:20 train test split on this model, not using the last 100 vectors for training, but testing where I could compare predicted results to the actual ones and determine the accuracy of each model (there were too many features to manually compare each result).

The SIN Analysis





Once again, we can see that the number of epochs decreases the error rate within an elbow like curve. However, unlike the previous experiment, the sin function actually improves with a lower learning rate, which may be caused by its logistic tangent nature versus the logistic sigmoid.

learning ra	Mean accuracy
0.1	0.625
0.01	0.9042
0.0001	0.9506

Finally, as we can see from the mean accuracy of each model, dependent on learning rate, in this sin function, we can accurately predict 95% of the test data's outputs using the tanh MLP, which is impressive.