Report

02 December 2024 17:05

Connectionist Computing MLP Report

Introduction

This report centers on the creation and evaluation of the Multi-Layer Perceptron (MLP) built independently of any pre-existing neural network libraries. The main goal is to investigate the basic principles of connectionist computing by implementing essential functions, such as forward propagation, backpropagation, and optimization of weights.

The MLP's performance is evaluated on tasks with differing complexities, including learning the XOR function, modelling random data through some function, and recognizing letters. However in this report I will not be creating and evaluating the MLP for letter recognition.

Design Choices for MLP

The MLP was designed as object oriented as it could, making it super flexible when it came to different configurations of Input, hidden units and Output as hinted and required by the assignment brief. This was mainly done in Java, however I later realised that python would have been the better choice in terms of built in libraries for visualization. So I continued to use Java for the MLP white writing data to a file, and using python on that file to create some visuals.

The architecture of the MLP;

- Input Layer: The input layer consists of input units, each representing a feature of the data. This layer receives the data and passes it to the next layer for further processing.
- Hidden Layer: The hidden layer contains hidden units that transform the input data through activation functions. The number of hidden units can be varied depending on the complexity of the problem.
- Output Layer: The output layer has one or more units, which represent the final prediction or classification result. The number of output units is determined by the nature of the task (e.g., a single unit for binary classification)

Algorithmic Details and Coding Choices

The following algorithms were implemented in order to create and train the MLP

Forwards Propagation

The process of computing the output by propagating the input vector through the network using weighted sums and Activations Functions.

<u>Weighted Sums:</u>
Each neuron in a layer computes a weighted sum of the inputs from the previous layer. Mathematically:

$$z = \left(\sum_{i=1}^{n} x_i \times w_i \right) + b$$

<u>Activation Functions</u>**:**
Once the weighted sums are computed, they are passed through an activation function to introduce non-linearity, and to determine if the neuron “activates” based on the weighted input. Sigmoid outputs a value between 0 and 1 suitable for probabilities and or binary output. Tanh outputs a value between -1 and 1 suitable for hidden layers as it centres data around zero.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$S(x) = \frac{1}{1 + e^{-x}}$$

Backwards Propagation

Refers to the method of modifying the weights and biases in a neural network to reduce the disparity between predicted results and actual outputs. It utilizes the error gradient related to each weight and bias, computed using the chain rule, to progressively enhance the network's performance. This process includes two main steps:

- Error Assessment: Identifying the error in the output layer.
- Gradient-Based Modifications: Sending the error backwards through the network to modify weights and biases, allowing the model to improve from its errors.

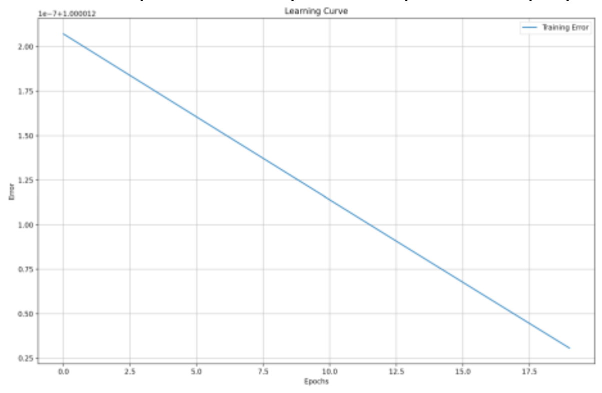
(NOTE; I RECEIVE AID ONLINE FOR BACKPROPAGATION METHOD AS IT WAS VERY COMPLICATED)

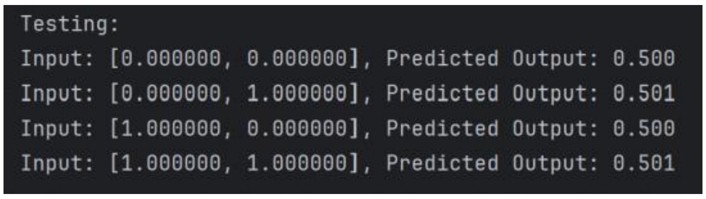
Updating Weights

The updateWeights() function modifies the neural network's weights by employing the gradient descent algorithm. For every weight in the input-to-hidden layer (W1) and the hidden-to-output layer (W2), the approach adjusts the gradients (dW1 and dW2) by the learning rate and deducts this amount from the existing weights. Following the update, the gradients are cleared to zero in readiness for the upcoming training iteration. This guarantees that weight modifications are gradual and in proportion to the calculated gradients, allowing the network to slowly reduce the error.

Experiments

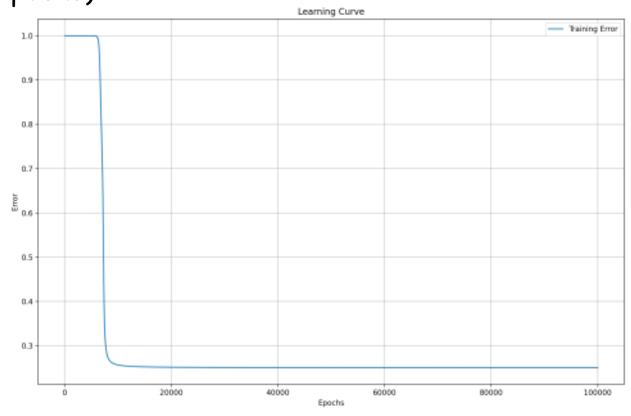
I created a network for the XOR problem with 2 inputs, 3 hidden units and 1 output. Using tanh and sigmoid activations functions and with bias 0 and learning rate 0.1 and for 20 epochs (Learning is done by a forward pass then by a back pass every epoch)

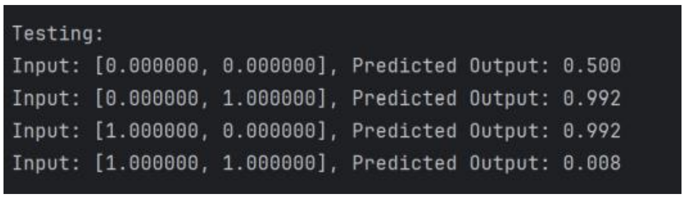




```
Testing:
Input: [0.000000, 0.000000], Predicted Output: 0.500
Input: [0.000000, 1.000000], Predicted Output: 0.501
Input: [1.000000, 0.000000], Predicted Output: 0.500
Input: [1.000000, 1.000000], Predicted Output: 0.501
```

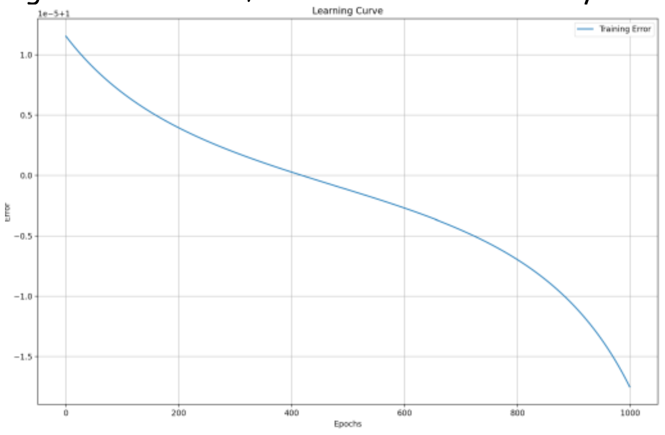
It did do some learning according to error visual, however it seems very random sitting at around 0.5 which seems to be the same as not learning. However I did see some major improvements when tuning up the epochs; see for 100,000 epochs; (as you can see best learning disappears basically after 10,000 epochs)



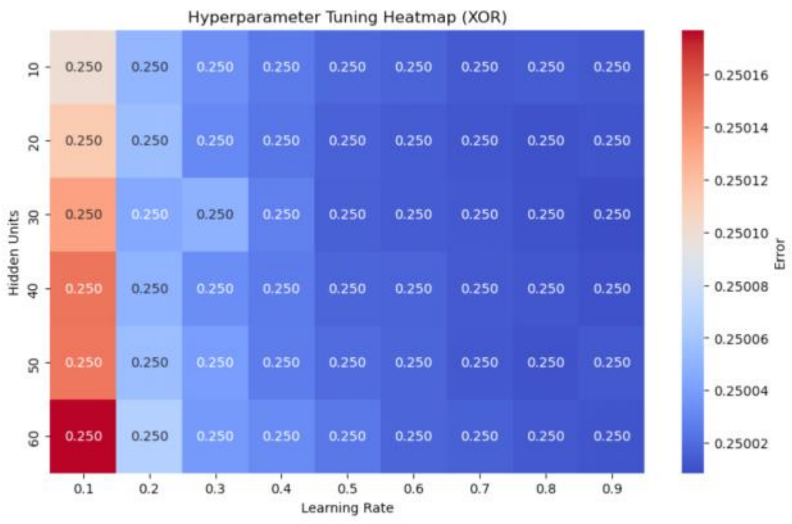


```
Testing:
Input: [0.000000, 0.000000], Predicted Output: 0.500
Input: [0.000000, 1.000000], Predicted Output: 0.992
Input: [1.000000, 0.000000], Predicted Output: 0.992
Input: [1.000000, 1.000000], Predicted Output: 0.008
```

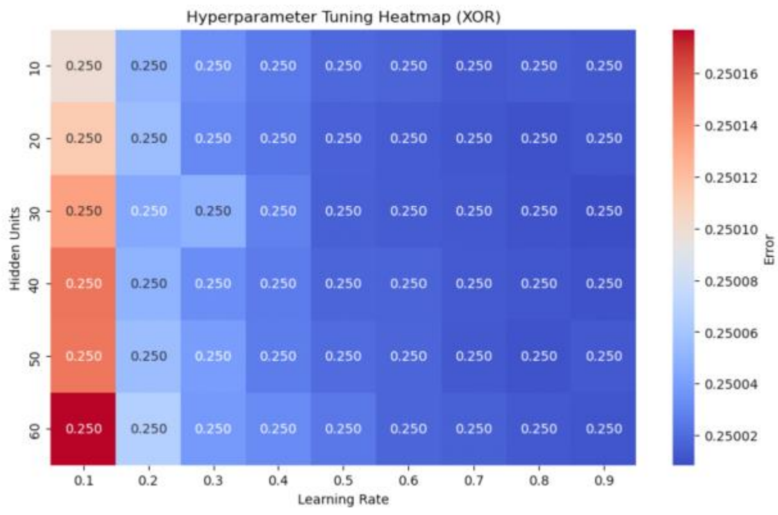
I played around with the hyper parameters a lot, after some frustration of not making significant progress learning at the low number of epochs. visualizations are crucial for analysing network performance. and I discovered that learning is very slow because of the learning rate being 0.1 and me not observing really what was going on. The problem was when I increased the learning rate by a significant amount, the network was basically unlearning by overshooting the error to the negatives!



After that I attempted to put all of that work into visualization using a heatmap (Top one for 100,000 epochs and below for 10,000 Epochs). However, since the variation in the errors were not much showing that hyper parameter tuning didn't really effect the overall results by much.

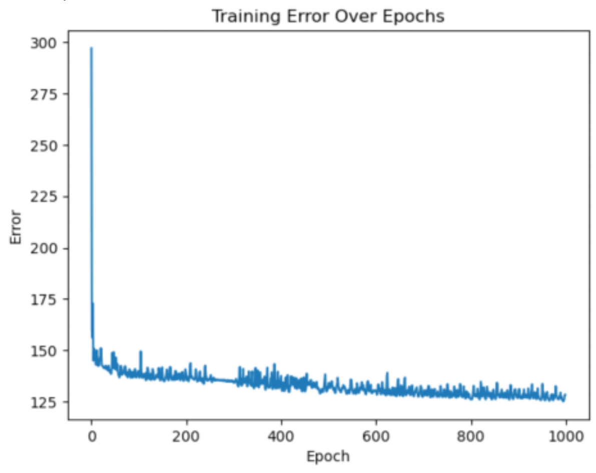


Hidden Units	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
10	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250
20	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250
30	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250
40	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250
50	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250
60	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250	0.250

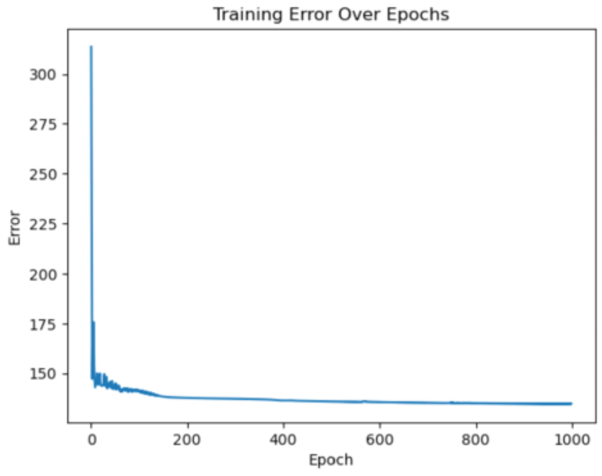


For part 3, I needed to train the MLP on randomly generated input vectors, each containing four components with values between -1 and 1. For each input vector, the corresponding output is calculated using the sine function: sin(x1 - x2 +x3 -x4). l split the generated data into a training set (400 examples) and a test set (100 examples), then train the MLP using backprop on the training set. After training, I evaluated the models performance by calculating and comparing the training and test errors.
I used the same network from XOR problem, first I tried batch learning of which I didn't see good results, I then modified the network to do online learning, because of this, it's now much slower though I obtained better results. However, I do believe that if I had chosen a deeper network to begin with parameter tuning might have been more insightful with good results and loads of computational power saves as well as time.

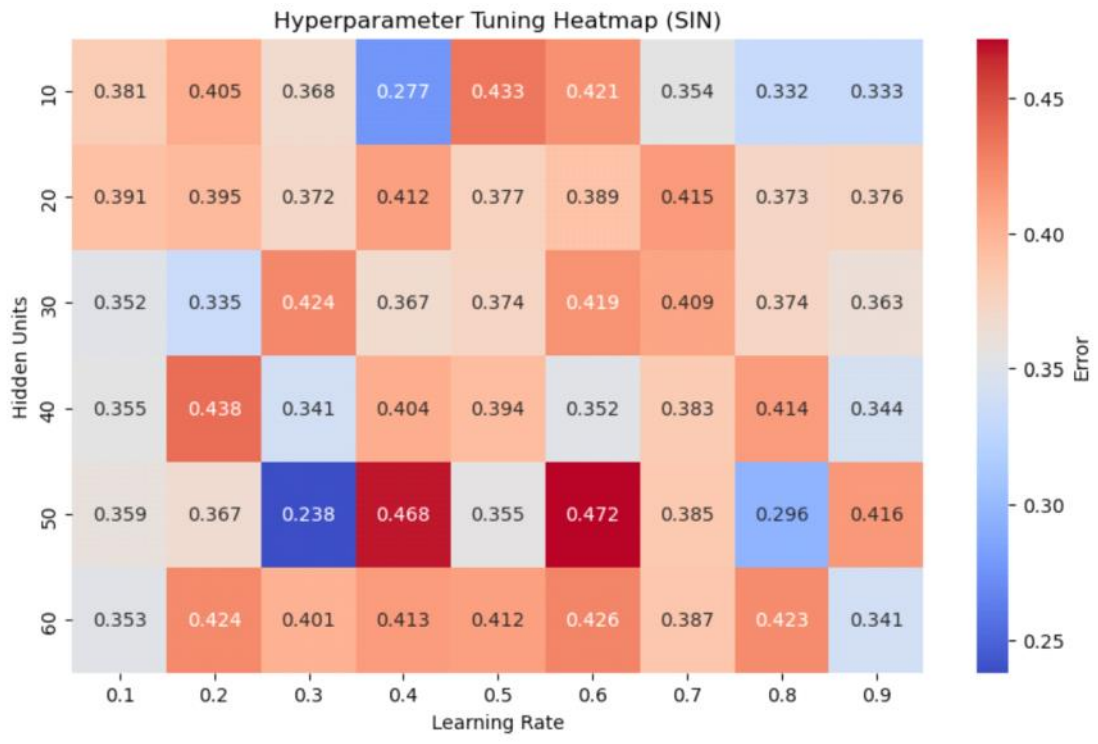
I observed loads of different error curves on different values of learning rates, hidden units sizes and maxEpochs: (frst one for HU = 10, LR 0.5)



Great fluctuations seen here.



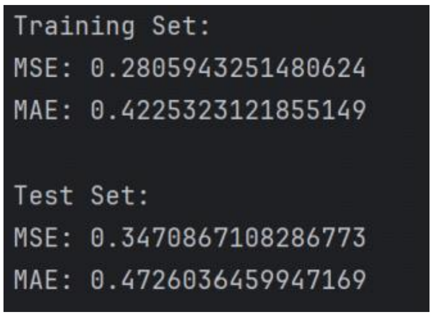
Decreasing the learning rate to 0.25 smoothens the error reductions indicating better convergence.



1000 Epochs

Best results Seen in with HU: 50 and LR: 0.3.

For large number of hidden units and epochs, this took much time, I wasn't able to get results for those.


```
Training Set:
MSE: 0.2805943251480624
MAE: 0.4225323121855149

Test Set:
MSE: 0.3470867108286773
MAE: 0.4726036459947169
```

(part 4) According to the MSE and MAE which are low, I think I have learned ok- but there are lots of room for improvement. And I do see the model has kind of memorised the training example as compared to the test examples. Results are not satisfactory, I think implemented a deeper 2 layers network will perform better.

Conclusion

Even though I only performed the bare minimum, I can still recognize the significant change in computation that arises from adding layers (I didn't do this, I tried) or enlarging the current ones. Determining the number of units per layer, the number of hidden layers, and the appropriate learning rate is crucial, as we aim to avoid wasting computational resources on an inadequate setup. My strategy was to initially run several epochs using various combinations of learning rates and layer sizes, followed by investigating the most promising ones. I can understand why this would be impractical with extensive networks.

From my experiments, it became clear that the learning rate plays a pivotal role in determining the speed and stability of training. Using a high learning rate often caused overshooting, leading to unlearning, while smaller learning rates resulted in slower but more stable convergence. These observations underscore the importance of adaptive optimization techniques that could mitigate such challenges in more complex networks.

Moreover, experimenting with different architectures provided valuable insights into the behaviours of the network under varying configurations. Although I only explored single-layer architectures, I suspect that deeper networks would have performed better on complex tasks, such as modelling sine-based functions. However, the added computational cost and time constraints prevented further exploration.