E9-253: Neural Networks and Learning Systems-I
HW-2

Submitted by - Ocima Kamboj
Serial Number - 06-02-01-10-51-18-1-15899

Problem-2 Lorenz Attractor

As per Haykin

Generation of data

The system of given differential equations is solved in MATLAB for the given values of σ, b and r. This gives us the values of $x(t)$, $y(t)$ and $z(t)$ at discrete values of time t.

We use the hyperbolic tangent activation function $\varphi(v) = tanh(v)$. In order to bring the range of output values of x, y and z within the range of the activation function, the input data is first preprocessed to make the mean values of x, y and z 0 each, and then it is normalised to bring the range to $[-1, 1]$.

Preparation of Training and Test data

Let the normalized data be represented by U, where the number of columns is 3, each corresponding to x, y and z. Starting from $i = 1$, we take $U(i, j)$ to $U(i+19, j)$ as our input data. j can be 1, 2 or 3 depending on whether we want x, y and z. The $U(i + 20, j)$ input is taken as the desired response. The value of i is increased by one until the required number of data points have been generated.

Three different networks are trained for x, y and z. The network trained by the specified set should predict $x(n)$ if it is given $x(n - 1)$, $x(n - 2)$, ... , $x(n - 20)$ as inputs.

Results

All the specifications are kept as per given in Haykin. That is, we use one hidden layer with 200 hidden neurons, 50 epochs for training, and linear annealing of η form 10^{-1} to 10^{-5}. We plot all the results for normalized values.

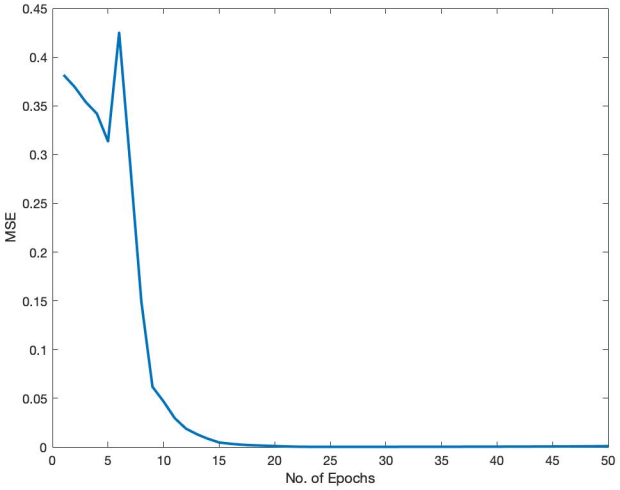

Figure 1: Learning Curve

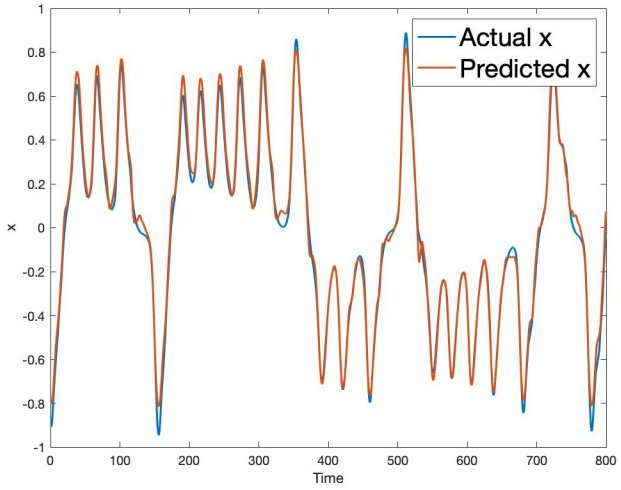

Figure 2: Predicted vs. Original

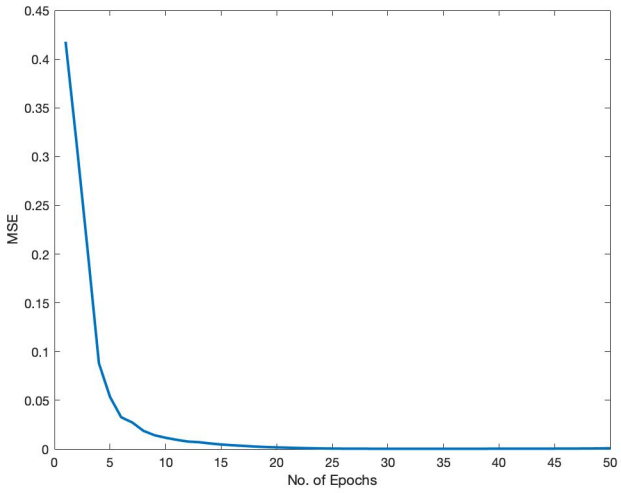

Figure 3: Learning Curve

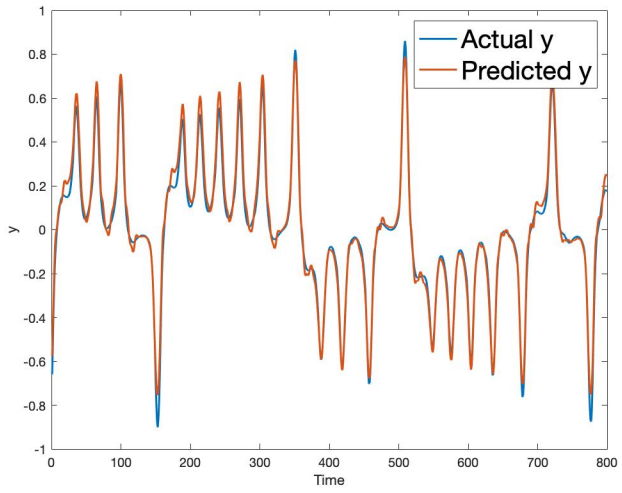

Figure 4: Predicted vs. Original

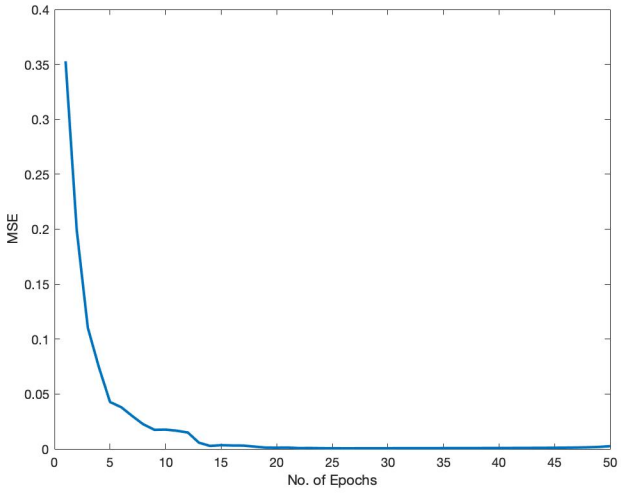

Figure 5: Learning Curve

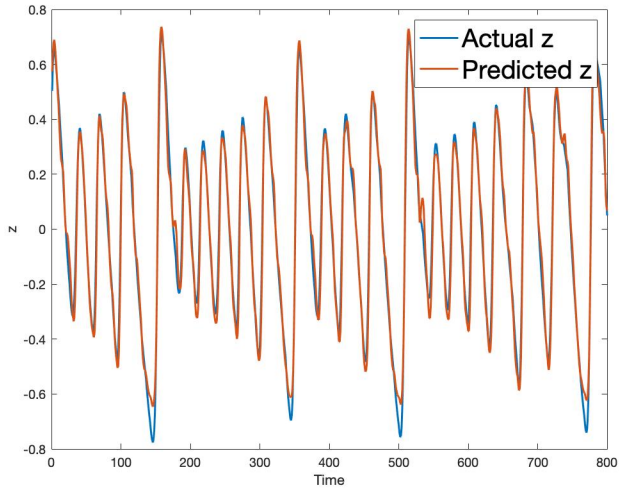

Figure 6: Predicted vs. Original

As per asked in HW

Generation of data

Same as the previous part.

Preparation of Training and Test data

Let the normalized data be represented by U, where the number of columns is 3, each corresponding to x, y and z. Starting from $i = 1$, we take $U(i, :)$ as our input data. The $U(i + 1, :)$ input is taken as the desired response. The value of i is increased by one until the required number of data points have been generated.

We predict $x(n)$, $y(n)$ and $z(n)$ given the values of $x(n-1)$, $y(n-1)$ and $z(n-1)$. The input and outputs are both three dimensional.

Results

All the specifications are kept as per given in Haykin. That is, we use one hidden layer with 200 hidden neurons, 50 epochs for training, and linear annealing of η form 10^{-1} to 10^{-5}.

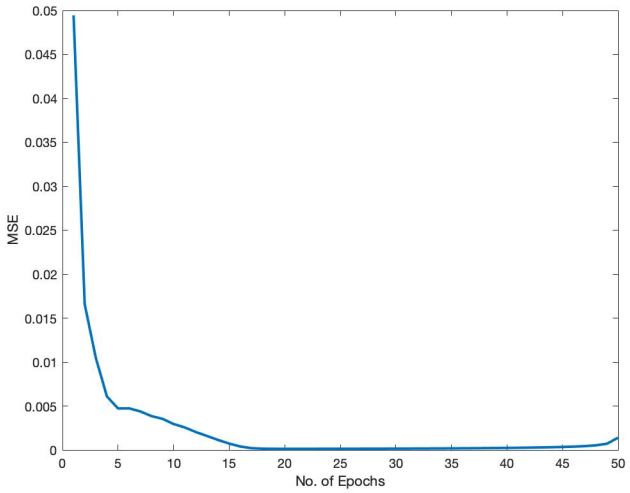

Figure 7: Learning Curve

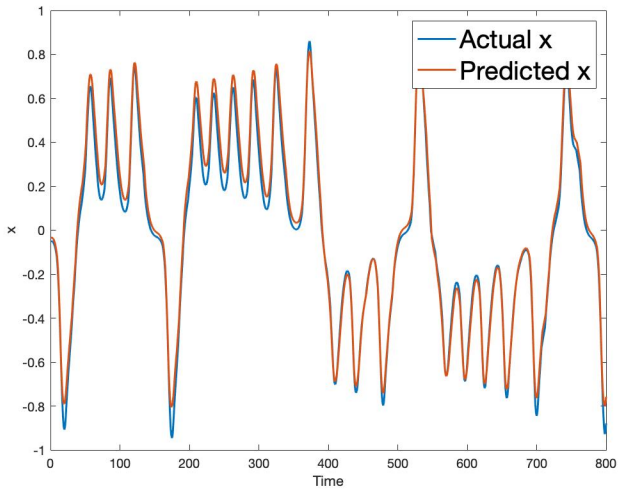

Figure 8: Predicted vs. Original

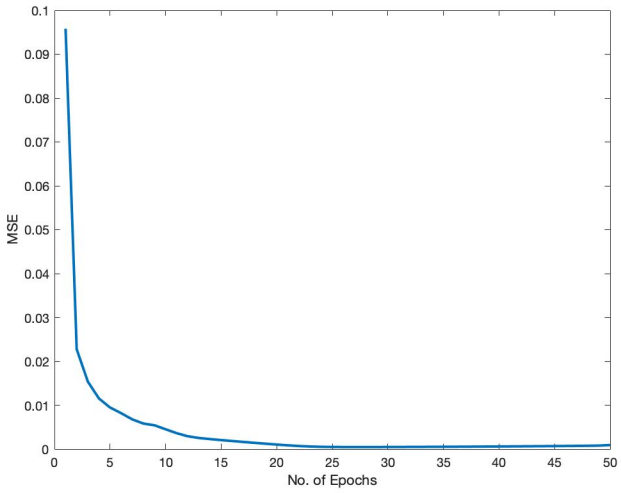

Figure 9: Learning Curve

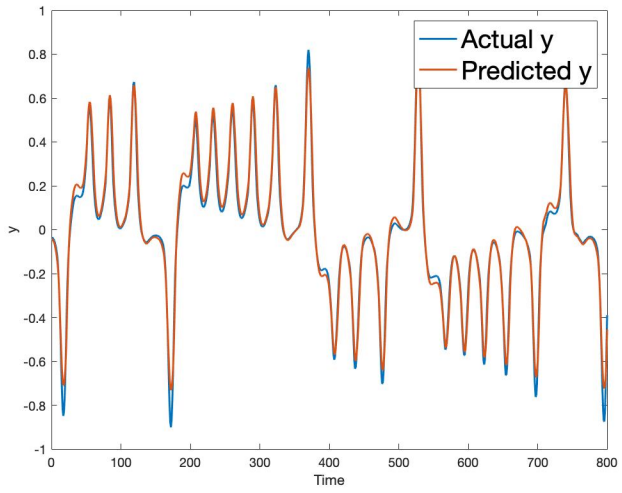

Figure 10: Predicted vs. Original

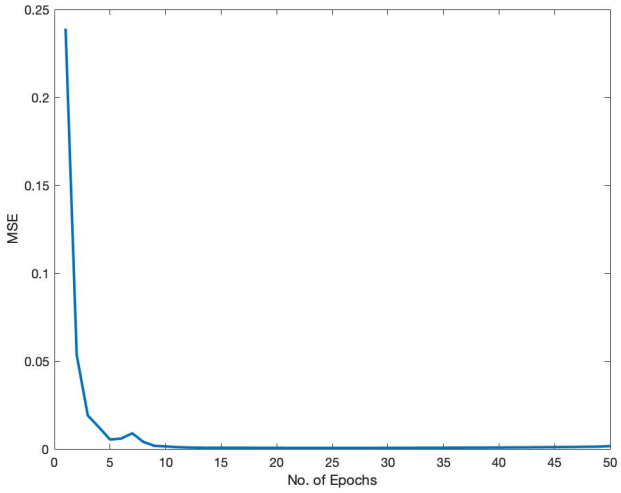

Figure 11: Learning Curve

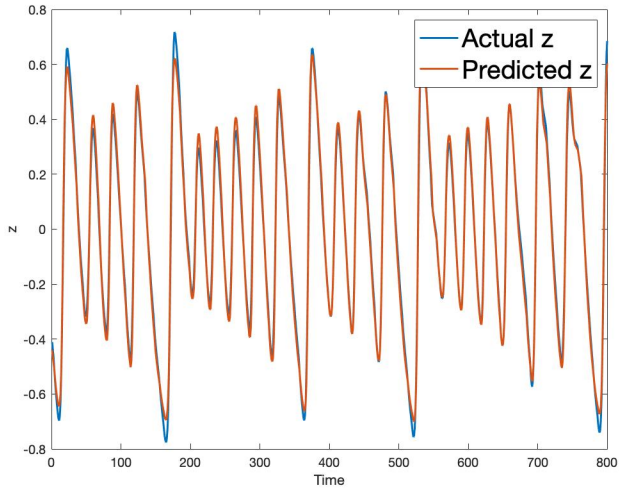

Figure 12: Predicted vs. Original

Observations

1. For both the cases, namely using past 20 values of x to predict the present x, or using past one value of x, y and z to predict the present value, the actual and the predicted sequences lie very close to each other.

2. The MSE during training becomes 0.

Problem-3 Classification using Multilayer Perceptron

Methodology

Experimental Setup
The programming of the MLP was done in MATLAB. For efficient implementation, the program was implemented in matrix form. A high-level pseudocode will be described below.

Let L be the number of layers in the network, excluding the input layer, and including the output layer. Thus, if $L = 2$, it implies that there is one hidden layer in the network.
The number of neurons in the input layer will always be equal to the dimension of the data, which in the case of the present grid classification problem is 2.
We need to specify the number of neurons in the hidden layers. For this we define an array $noNeurons$. $noNeurons(l)$ stores the number of neurons in layer-l.
In the grid problem case, the output layer always consists of one output, telling us which class the point belongs to. Therefore, the output layer always has one neuron in this case.

We define two different cell structures to store the biases and the weights of each layer respectively. Thus, we form a cell array B of length-L to store the biases of each layer. $B\{l\}$ is a column vector whose length is equal to the number of nuerons in layer-l. The i^{th} element of this column vector is the bias applied to neuron-i in layer-l.

Similarly we define a cell array W of length-L to store the weights of each layer. $W\{l\}$ is a matrix with number of rows equal to the number of neurons in layer-l, and number of columns equal to the number of neurons in layer-$(l-1)$. The w_{ij} component of this matrix is the synaptic weight connecting neuron-j in layer-$(l-1)$ to neuron-i in layer-l.

Let the input data be contained in a matrix $data$. $data(n)$ is a column vector, whose length is equal to the dimension of the input data, specifying coordinates of the n^{th} input data point. A separate array named $desired$ is defined. $desired(n)$ is the desired response for the n^{th} input.

Let $\varphi(.)$ be the activation function. Let η be the learning rate. Let Y be a cell array of length L to store the outputs of each layer. Let δ be a cell array of length L to store the local gradients of each layer.

Initialization The initial values for W and B are generated as uniformly distributed random numbers between $[-0.5, 0.5]$. This shifting from $[0, 1]$ has been done to keep the mean of the weights approximately zero.

Forward Pass Consider a particular time step n. The forward pass comprises of calculating all the function signals as follows -

```
for l=1 to L do
    if l==1 then
        v(n) = [B{l}(n) W{l}(n)] × [1 data(:,n)]^T;
    else
        v(n) = [B{l}(n) W{l}(n)] × [1 Y{l−1}(n)]^T;
    end
    Y{l}(n) = φ(v(n))
end
```

Calculation of errors

$E(n) = desired(:, n) - Y\{l\}(n);$

Backward Pass The error signal is propagated backwards to calculate the local gradients -

for *l=L to 1* **do**
  if *l==L* **then**
    $\delta\{l\}(n) = E(n). * \varphi'(Y\{l\}(n));$
  else
    $\delta\{l\}(n) = \varphi'(Y\{l\}(n)). * [W\{l+1\}(n)^T \delta\{l+1\}(n)];$
  end
end

Here, $.*$ denotes element-wise multiplication.

Update of weights The weights are updated as follows -

for *l=L to 1* **do**
  if *l==1* **then**
    $[B\{l\}(n+1)\ W\{l\}(n+1)] = [B\{l\}(n)\ W\{l\}(n)] + \eta\delta\{l\}[1\ data(:, n)]^T;$
  else
    $[B\{l\}(n+1)\ W\{l\}(n+1)] = [B\{l\}(n)\ W\{l\}(n)] + \eta\delta\{l\}[1\ Y\{l-1\}]^T;$
  end
end

This is continued until the required stopping criterea is met.

Generation of Data Points

An $N \times N$ grid with spacing D is formed. The grid is always centered around 0. This is done so that the mean of inputs is approximately 0. The class labels for this data are decided on the basis of the activation function so as to keep the output within the range of the activation function. For example, with the Sigmoid activation function, the class labels are 0 and 1, where as for hyperbolic tangent activation function, the class labels are -1 and 1.

Prediction

The data is classified to a particular class based on a threshold Th. If the output is greater than Th, it is assigned class-1, otherwise it is assigned class-2. This threshold depends on the activation function, and it is chosen to be in the middle of the range of that activation function. For example, for the sigmoid activation function, the threshold is 0.5, whereas for hyperbolic tangent activation function, the threshold is 0.

(a) Aim - Classify the points marked as 'cross' and 'dot' into two classes and plot the decision boundary. Use the sigmoid activation function. Vary N.

Activation Function -

$$\varphi(v) = \frac{1}{1 + e^{-v}}$$

The class labels were assigned a value of either 0 or 1. The threshold was set to 0.5.

Determination of hyperparameters
Determination of Number of layers - The Universal Approximation theorem states that one hidden layer is sufficient for a multilayer perceptron to approximate a given data set. Therefore, we start our testing with one hidden layer.

Determination of Number of Hidden Neurons - The XOR problem required two lines to separate the two classes. XOR was a 2×2 grid. Minimum two neurons were needed to achieve this separability. An $N \times N$ grid would require $2N - 2$ lines to separate the two classes. Therefore, we start our testing with $2N - 2$ hidden neurons.

Notation followed for the plots -
N **- size of** $N \times N$ **grid**
D **- Distance between two subsequent 'dot' and 'cross'**
η **- learning rate**
epochs - the number of epochs that the MLP algorithm was run
H.L - the number of hidden layers
H.Neurons - an array of the number of neurons in each hidden layer
C.A. - Classification Accuracy

In all the plots, decision boundaries are plotted with colour maps to distinguish the two regions.

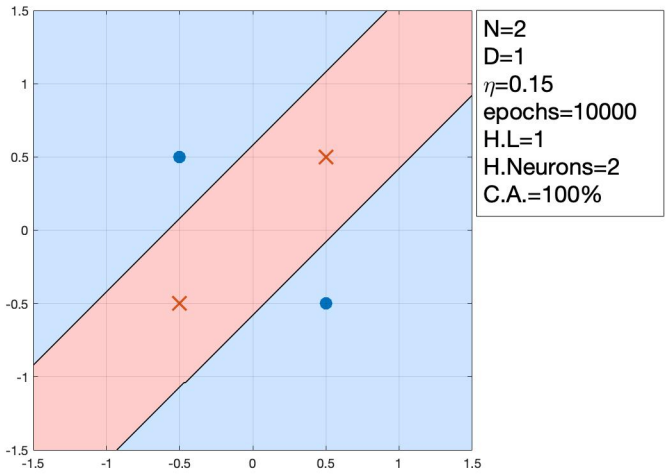

Figure 13

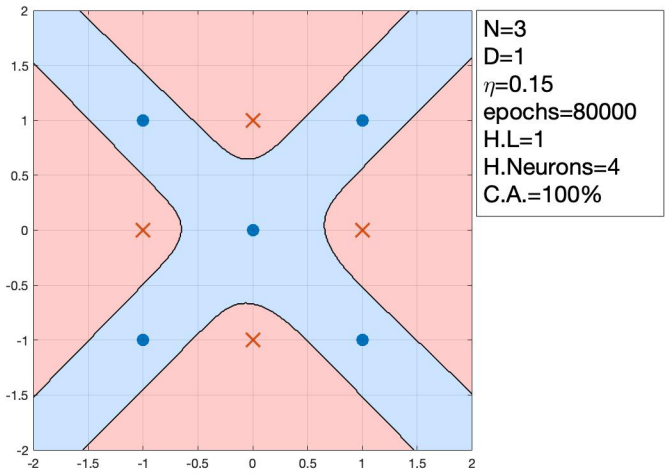

Figure 14

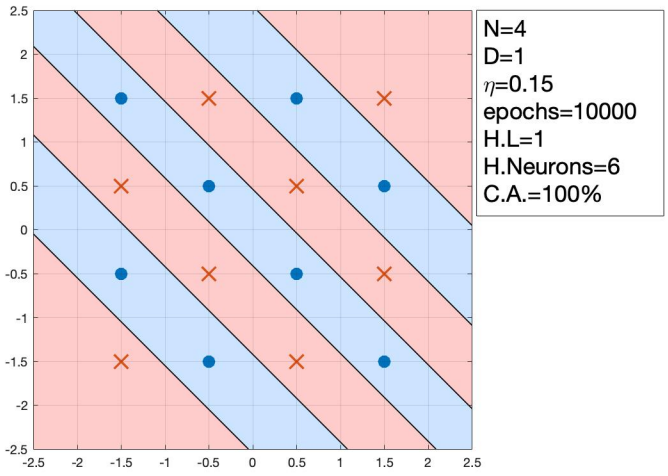

Figure 15

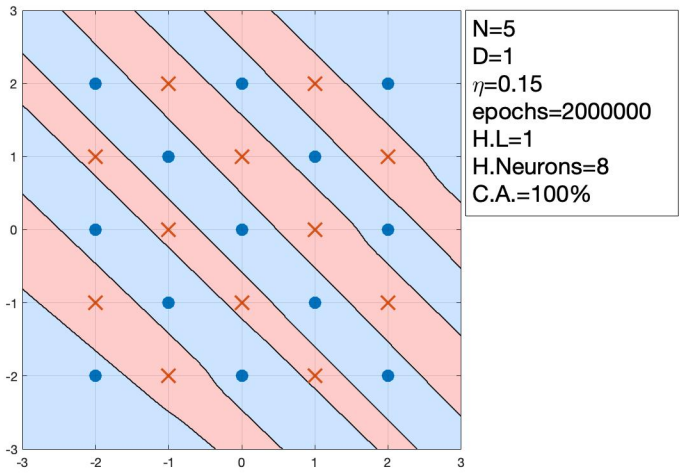

Figure 16

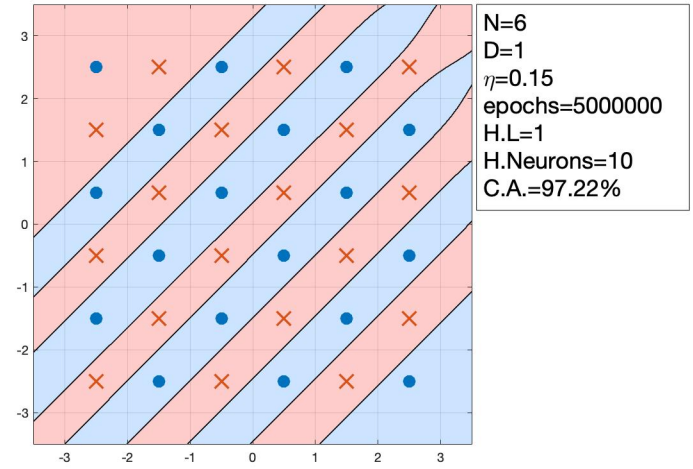

Figure 17

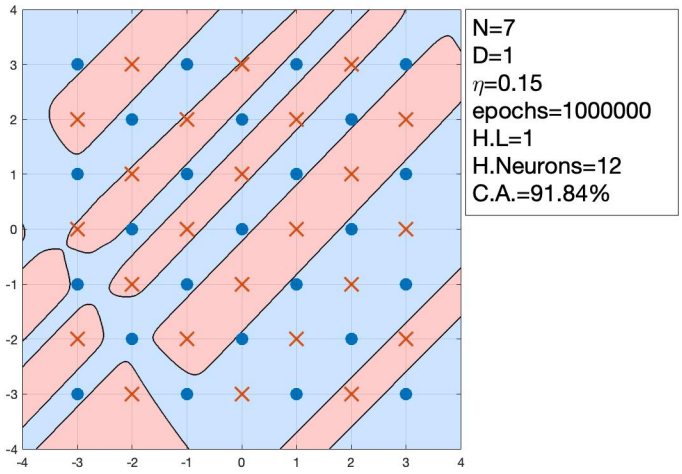

Figure 18

For an increasing value of N, our original estimate for the number of hidden neurons to be $2N - 2$ is proving to be unsufficient. Therefore, we see the effect of increasing the number of neurons.

Changing the Hyperparameters

Increasing the number of neurons in a single hidden layer
The number of neurons in the single hidden layer were increased until a classification accuracy of 100% was reached. The final results are plotted below.

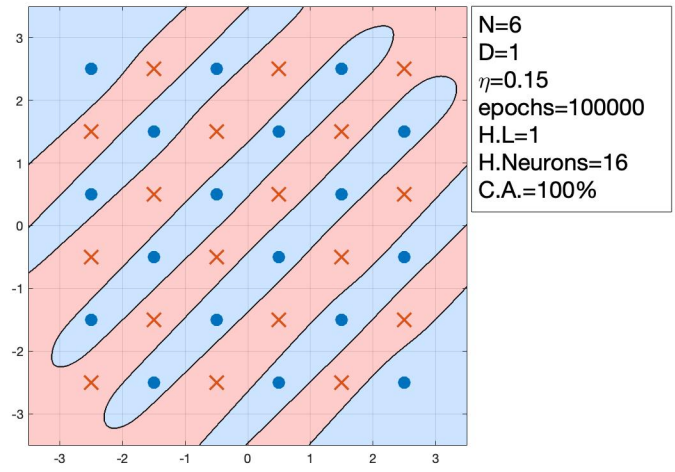

Figure 19

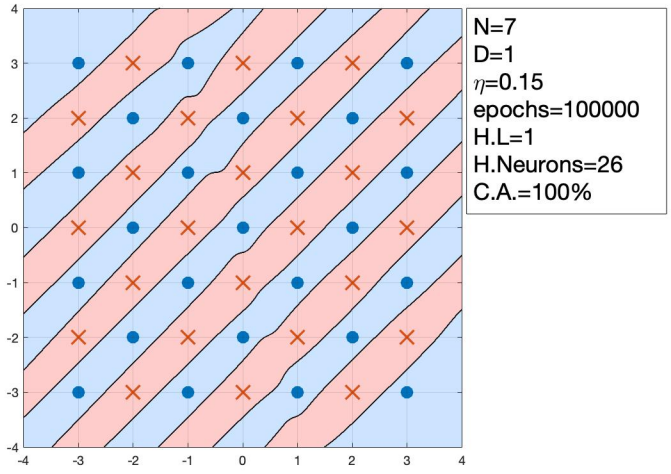

Figure 20

Increasing the number of hidden layers

We see the effect of increasing the number of hidden layers to two.

For comparable computational complexity, we start with the same overall number of neurons that were sufficient to classify the points successfully in the single-layer network. That is, if for a particular N, the single layer network required $2k$ neurons in the hidden layer to classify all the points correctly, then, for the two hidden layers case, we initialize the number of neurons in each of the hidden layers to k. We then increase the number of neurons in these hidden layers until we get a classification accuracy of 100%. The final results are plotted below -

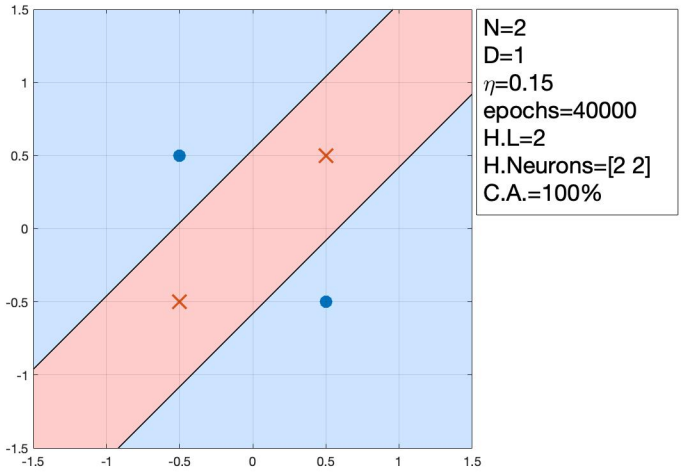

Figure 21

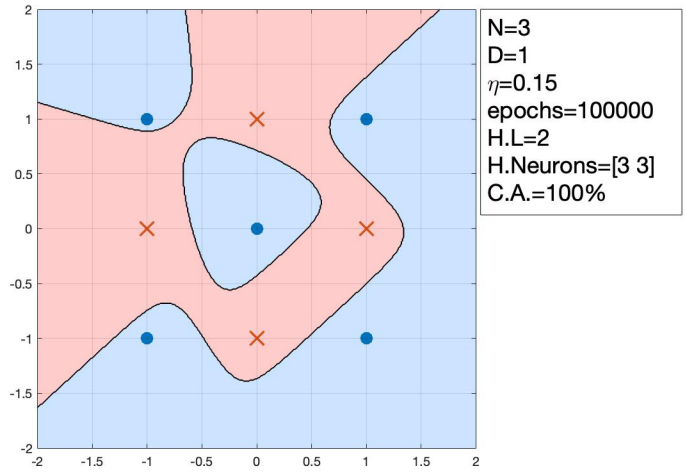

Figure 22

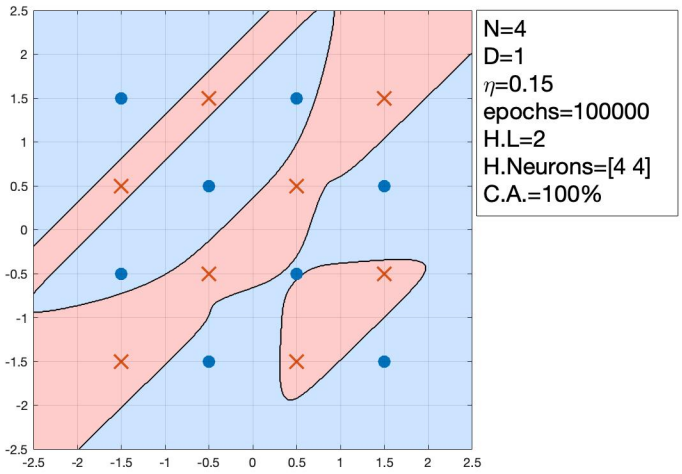

Figure 23

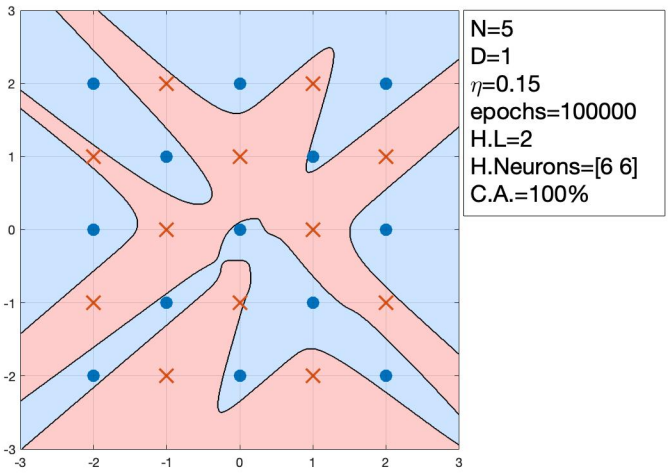

Figure 24

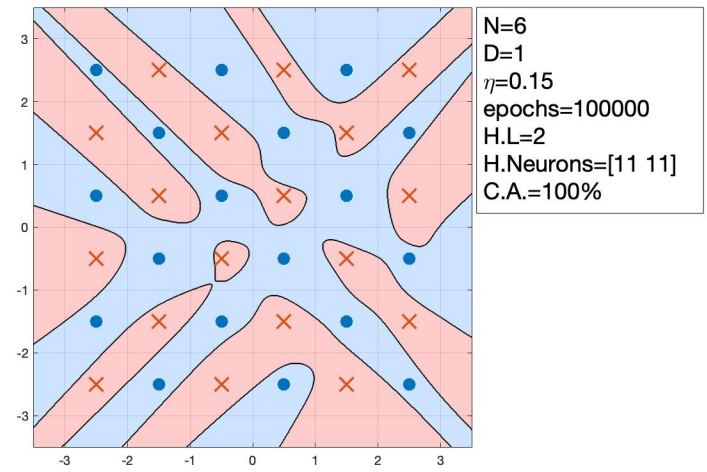

Figure 25

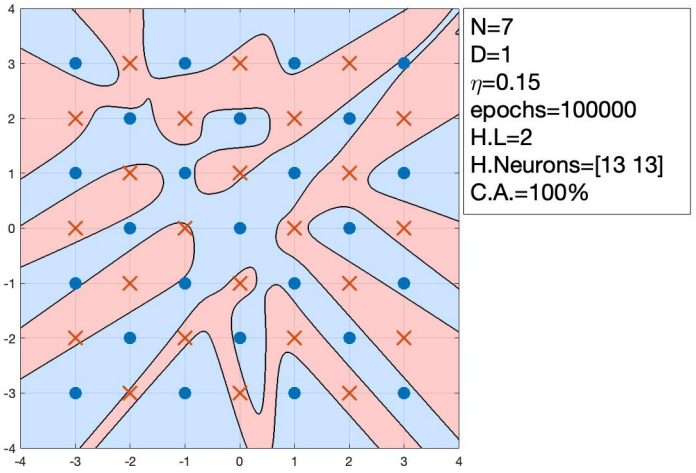

Figure 26

Observations

1. Multilayer Perceptron trained with backpropagation is able to correctly classify non-linearly separable patterns.

2. The estimated number of neurons required for 100% classification accuracy in a grid of size $N \times N$ is $2N - 2$. This is based on the minimum number of hyperplanes that are required to separate the grid points.

3. For small values of N, the estimated number of neurons prove to be sufficient to classify the points correctly.

4. For $N > 5$, our algorithm with $2N-2$ neurons is not converging for even 50,00,000 epochs. The time required to complete these many epochs was about half an hour. It's a guess that if the algorithm is allowed to run for alot of time, $2N - 2$ neurons would prove to be sufficient for 100% classification accuracy. But, for practical purposes, we don't want the run time of our program to be too large.

5. The number of neurons required, beyond our estimate of $2N - 2$, for correct classification increase with the increase in grid size N.

6. We see that a single hidden layer is able to correctly classify the grid problem with varying N. This is in accordance with the *Universal Approximation Theorem*.

7. For a network with two hidden layers, the total number of neurons required for correctly classifying the data points is more than the number of neurons required when we are using a single hidden layer.

8. The decision boundaries generated for the single hidden layer case are, in general, more symmetric, or 'smooth', than the ones generated for the double layer case. This, along with the previous point, suggests that the use of just a single hidden layer is a good choice for the present problem.

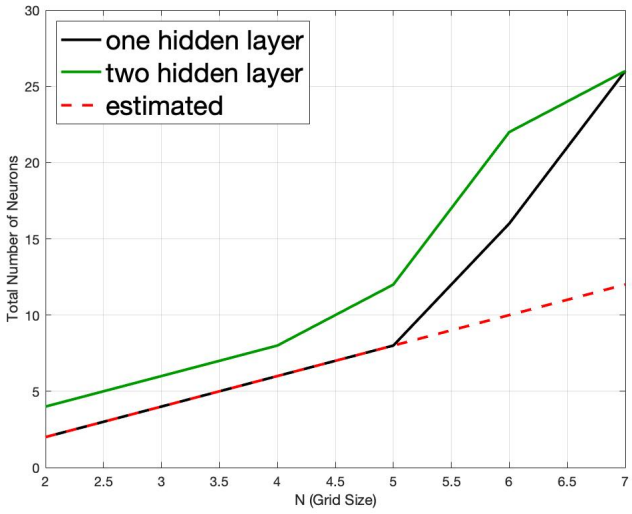

Figure 27: Total Number of Neurons vs. Grid Size

(a) Aim - Repeat (a) with activation functions: 1) ReLu, 2) tanh(.) 3) erf(.), and 4) heavy-side function.

1) ReLU activation function

Activation Function -

$$\varphi(v) = max(0, v)$$

$$\varphi'(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{otherwise} \end{cases}$$

The class labels were assigned a value of either 0 or 1. The threshold was set to 0.5.

Determination of Hyperparameters
It was observed that to introduce non-linearity with the ReLU activation function, more number of hidden layers and neurons were required. Therefore, the number and layers were increased. Final results are plotted below.

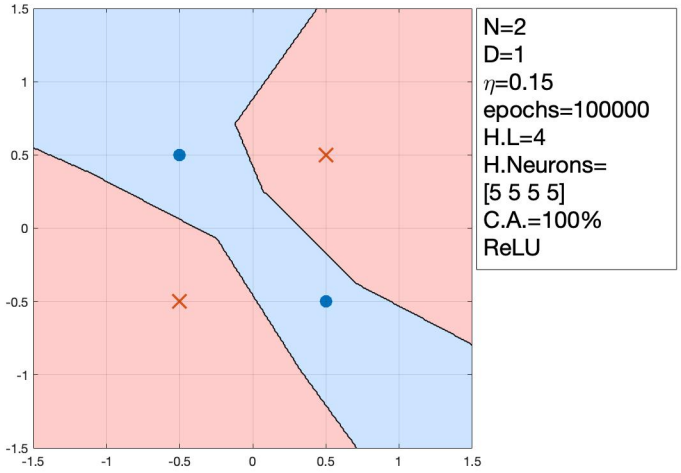

Figure 28

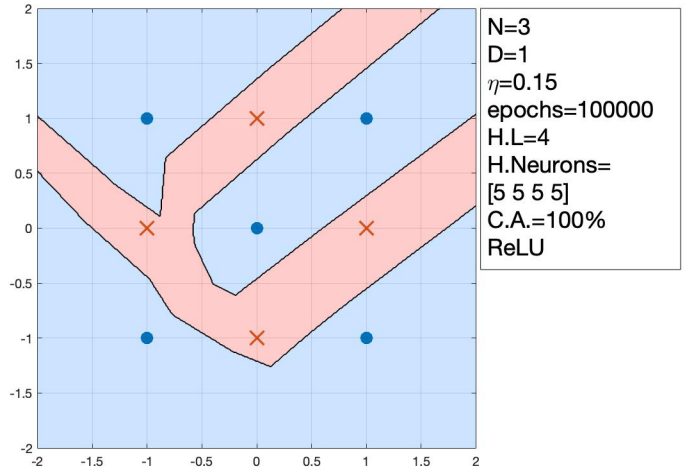

Figure 29

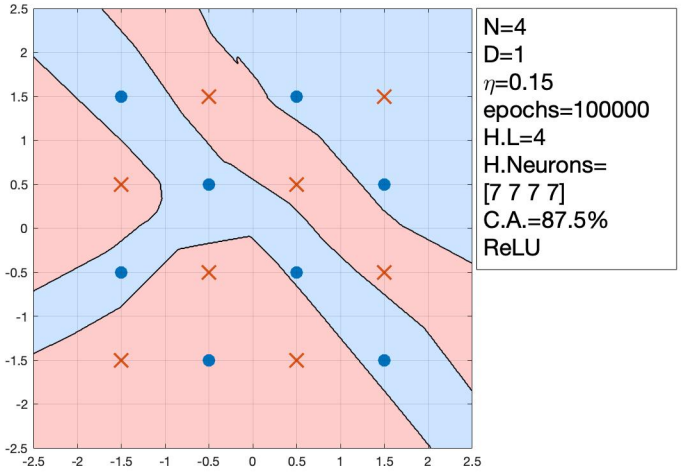

Figure 30

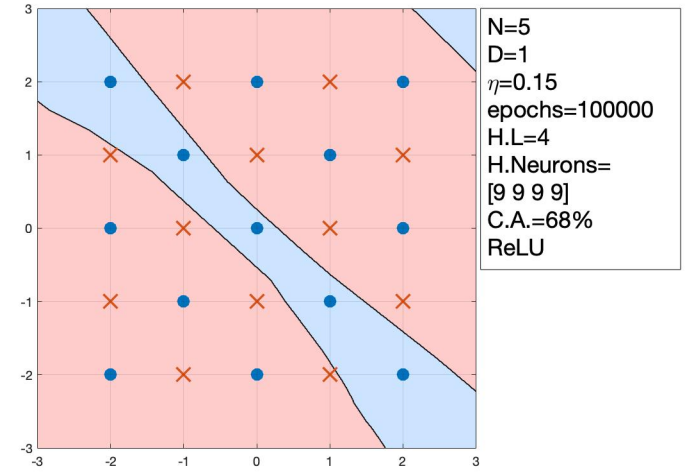

Figure 31

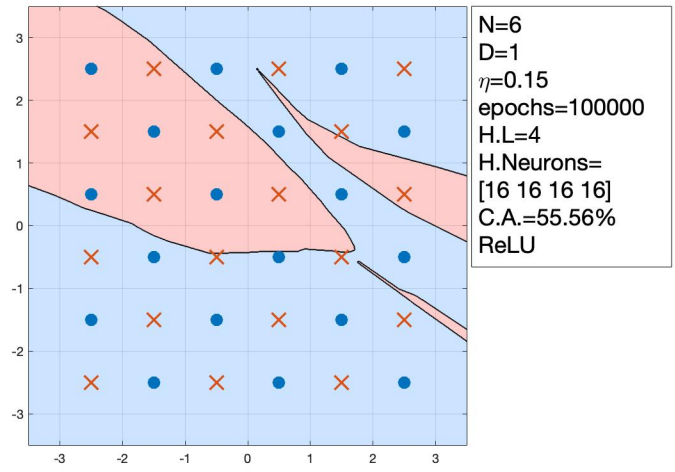

Figure 32

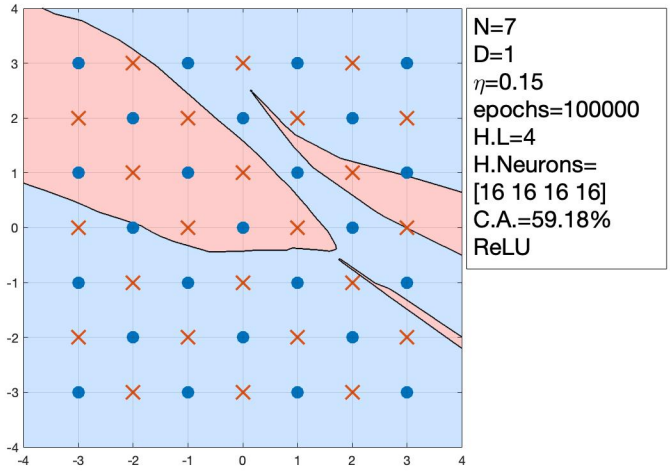

Figure 33

Observations

1. When size of grid is very small, the ReLU activation function is able to classify the points correctly.
2. To introduce non-linearity in our decision boundaries, more number of layers are required with Re-LU function.
3. With ReLU function, the classification accuracy degrades for large values of N.
4. The training times were 2-5 minutes.

2) tanh activation function

Activation Function -

$$\varphi(v) = tanh(v)$$

The class labels were assigned a value of either -1 or 1. The threshold was set to 0.

Determination of Hyperparameters
Like the sigmoidal case, we keep a single hidden layer. We start with $2N - 2$ neurons and increase the number of neurons until a classification accuracy of 100% is reached. The final results have been plotted below.

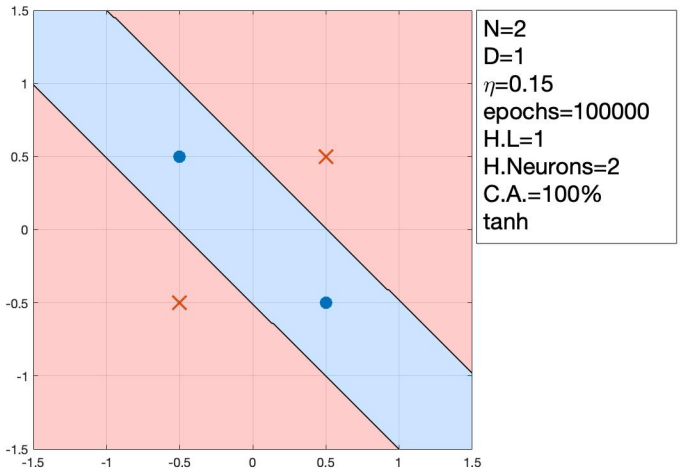

Figure 34

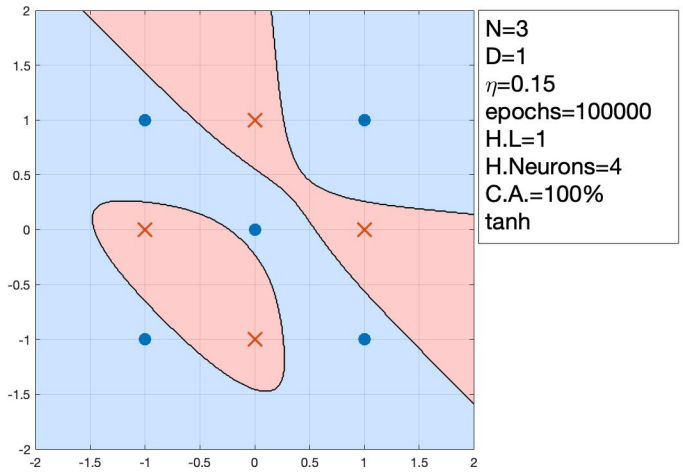

Figure 35

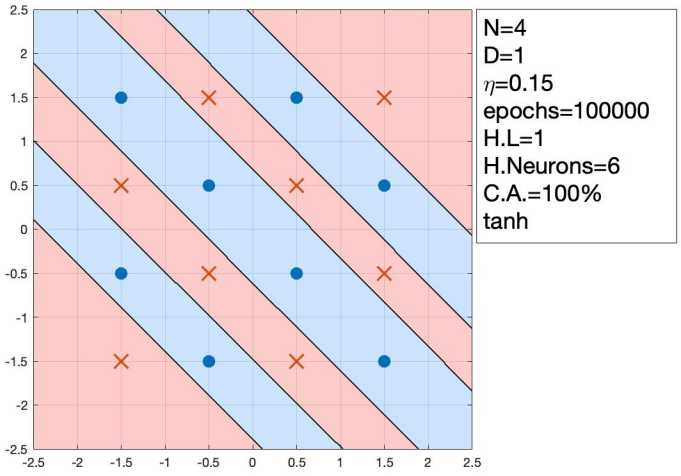

Figure 36

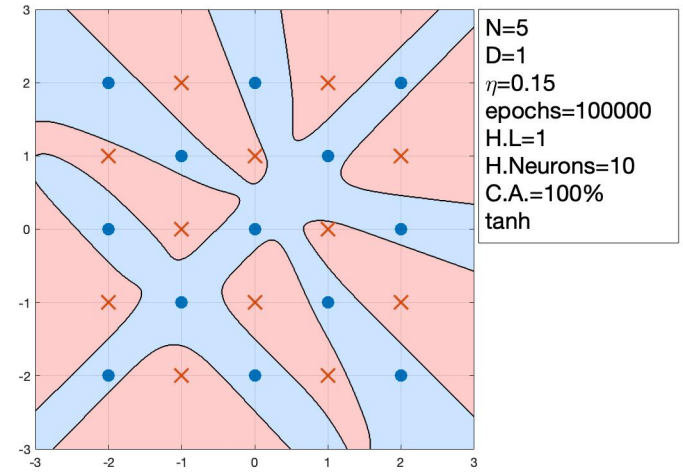

Figure 37

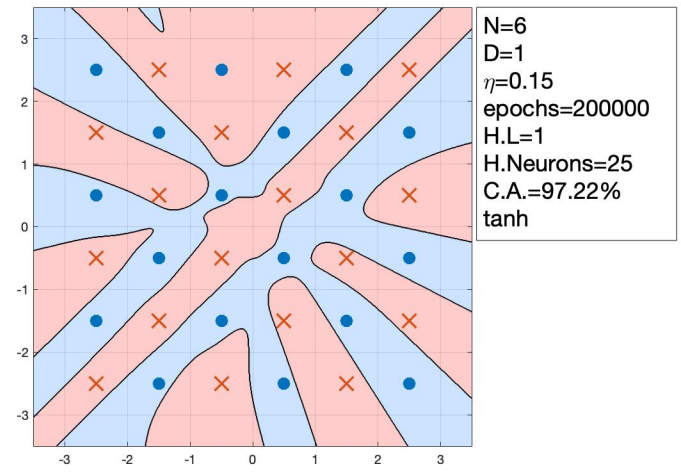

Figure 38

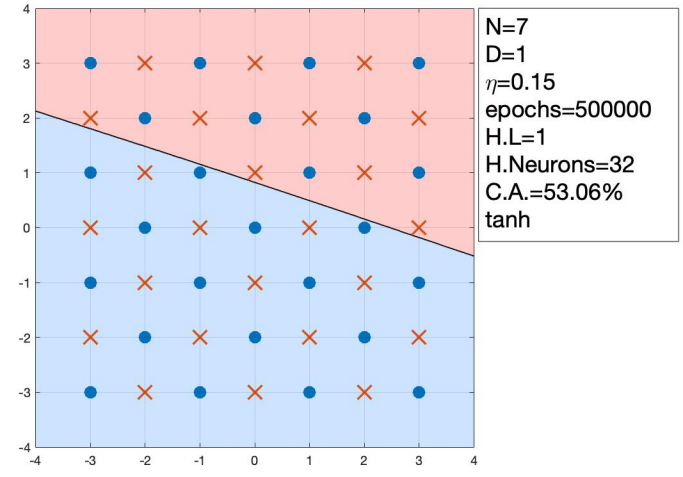

Figure 39

Observations

1. The algorithm is found to perform better when the initial weights are kept between 0 to 1 rather than -0.5 to 0.5.

2. It was expected that the $tanh(.)$ activation function will perform similar to the sigmoid activation function in part (a). But, the sigmoid function seems to be performing better, in terms of the number of neurons required, and the smoothness of the decision boundaries.

3) erf(.) activation function

Activation Function -

$$\varphi(v) = \frac{1}{\sqrt{\pi}} \int_{-v}^{v} e^{-t^2} dt$$

$$\varphi'(v) = \frac{2}{\sqrt{\pi}} e^{-v^2}$$

The class labels were assigned a value of either -1 or 1. The threshold was set to 0.

Determination of Hyperparameters

We keep a single hidden layer. We start with $2N - 2$ neurons and increase the number of neurons until a classification accuracy of 100% is reached. The final results have been plotted below.

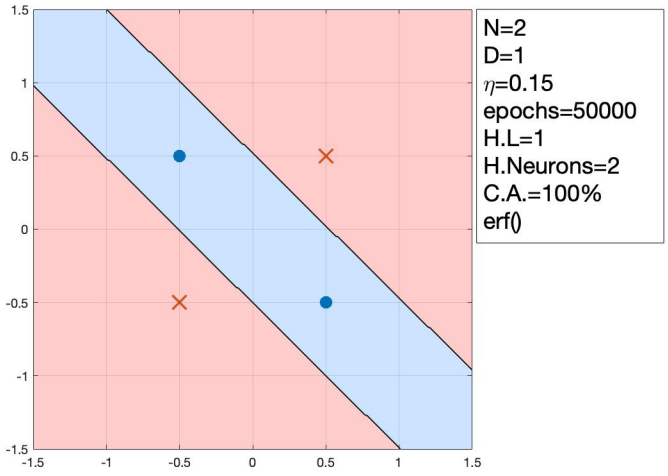

Figure 40

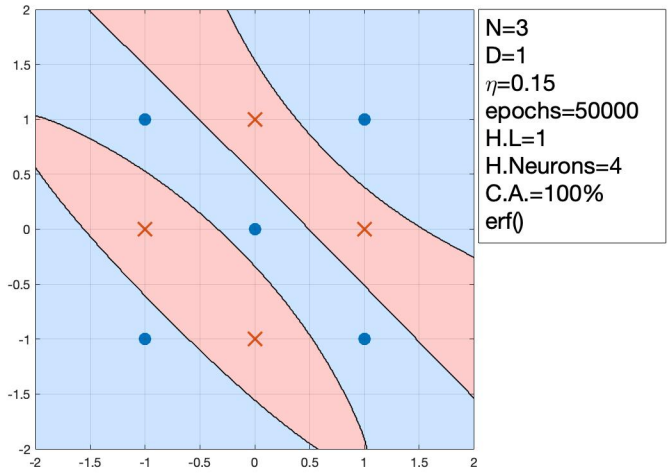

Figure 41

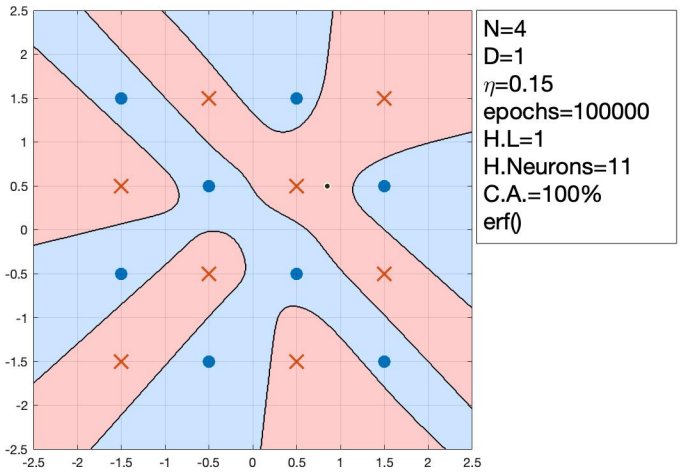

Figure 42

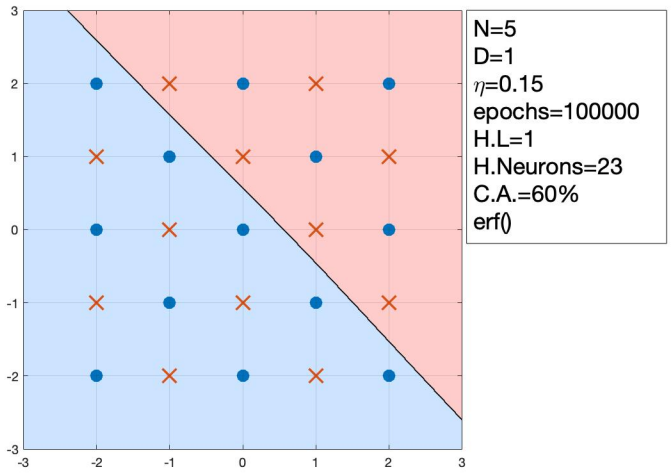

Figure 43

Observations

1. For small values of N, a classification accuracy of 100% was achieved.
2. For large values of N, the program was having difficulty converging.

4) Heavy-side activation function

Activation Function -

$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{otherwise} \end{cases}$$

As the function is discontinuous, and hence not differentiable, we use a logistic function to approximate the heavy-side function.

$$\varphi(v) = \frac{1}{1 + e^{-av}}$$

The larger the value of the parameter a, the closer we approximate the heavy-side function. We keep the value of a 15. The class labels were assigned a value of either 0 or 1. The threshold was set to 0.5.

The results are plotted below -

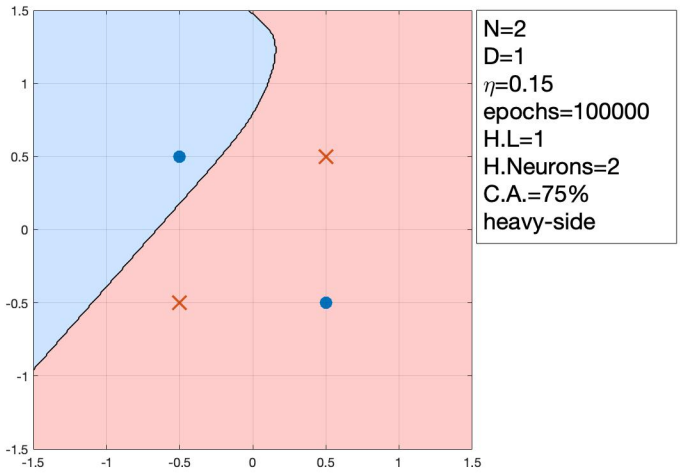

Figure 44

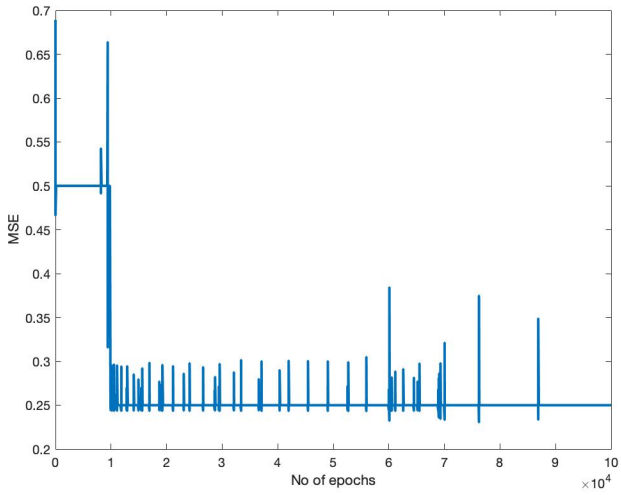

Figure 45: Learning Curve

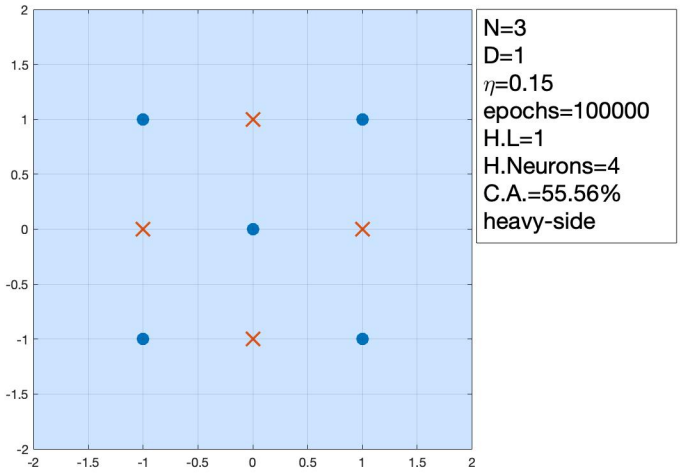

Figure 46

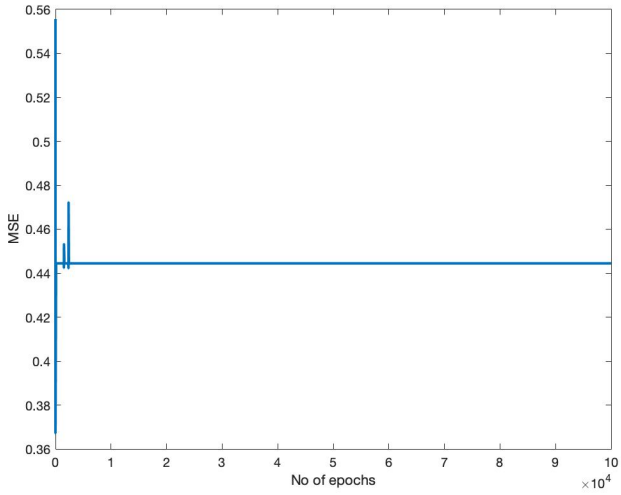

Figure 47: Learning Curve

Observations

1. It is expected that the algorithm with heavy-side activation function will have difficulty converging because of large values of δ near 0.

2. We see from the learning curve that the algorithm keeps on oscillating in order to find the optimum value of weights.

(c) Aim - Repeat (a) with one hidden layer and by varying the value of D from 0.1 to 1 in steps of 0.1.

The conditions were kept same as part-(a). We started with the number of neurons required for 100% classification accuracy for $D = 1$ in part-(a), and increased the number of neurons as required while decreasing D. The results for N equal to 2, 4, 6 are shown below.

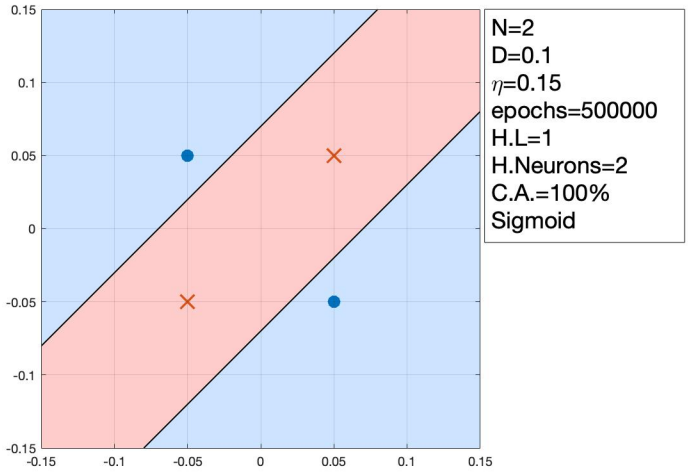

Figure 48

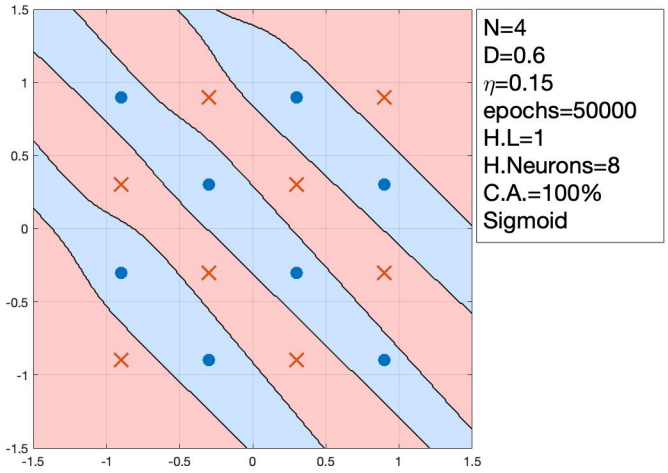

Figure 49

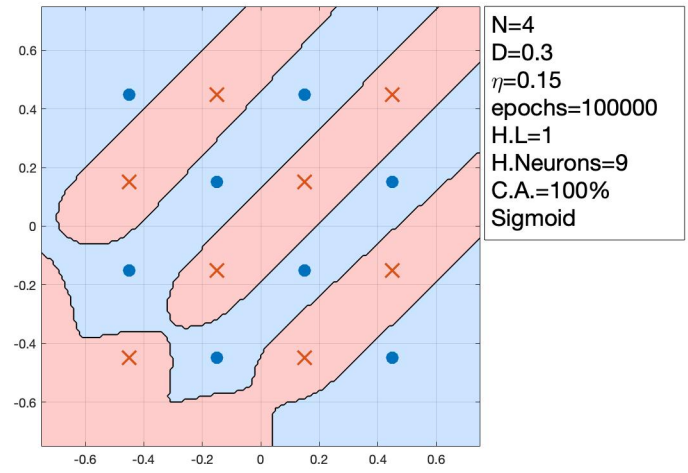

Figure 50

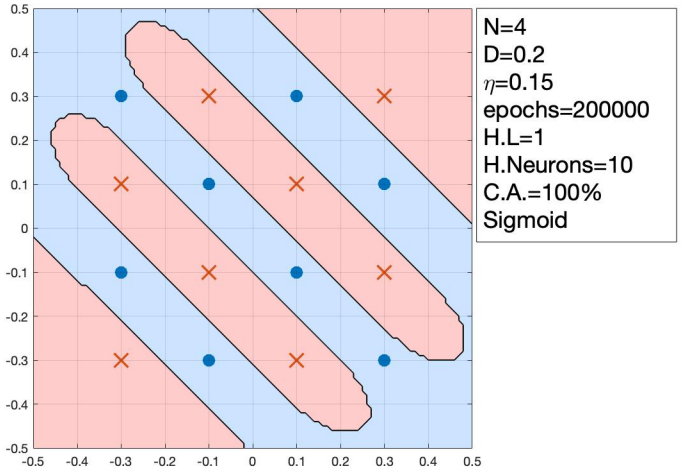

Figure 51

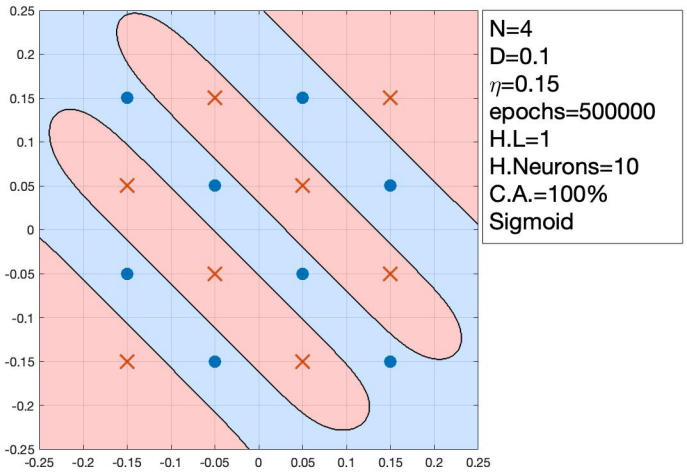

Figure 52

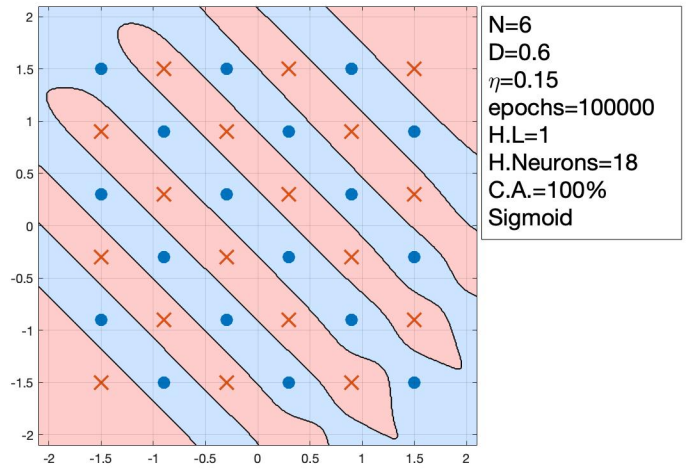

Figure 53

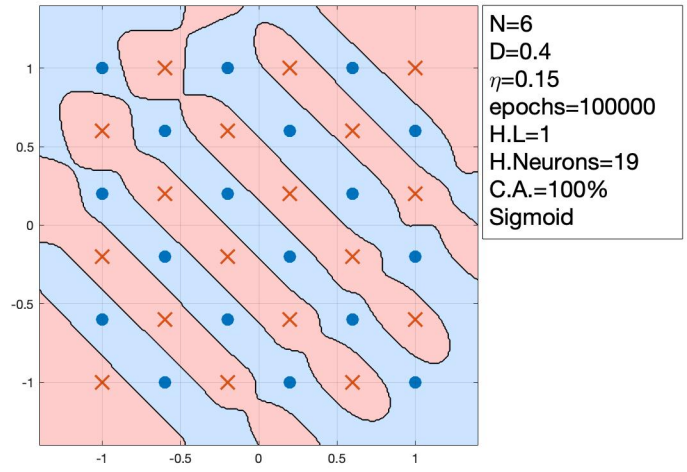

Figure 54

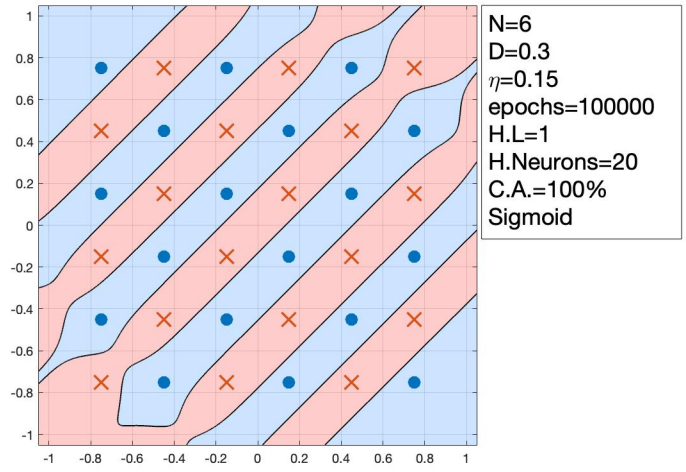

Figure 55

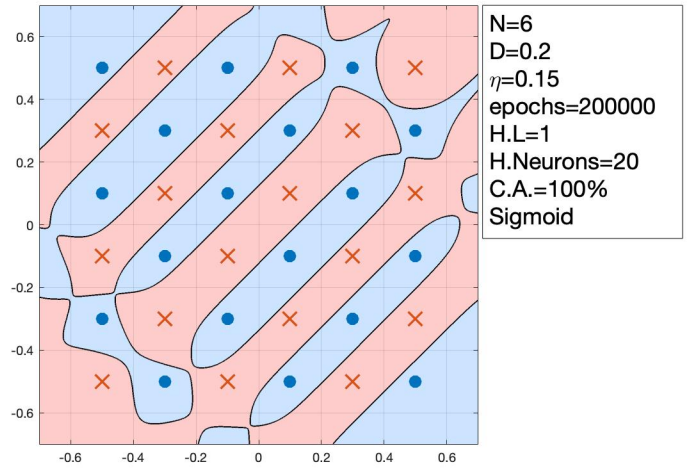

Figure 56

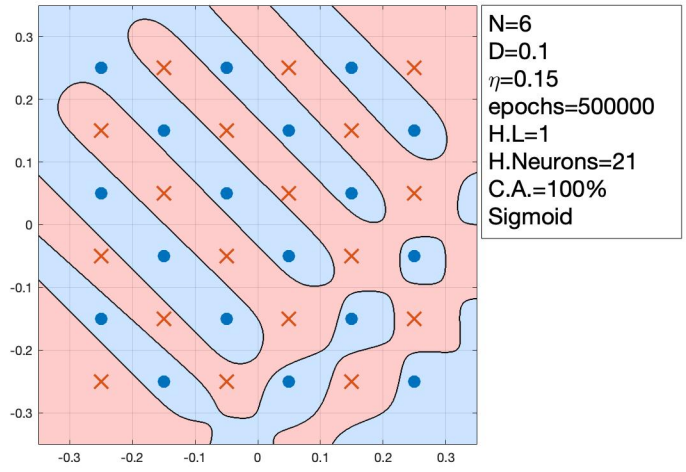

Figure 57

Observations

1. A single hidden layer is enough to achieve a 100% classification accuracy even upto small values of D and large values of N. This is in accordance with the universal approximation theorem.

2. Except for the simple case of a 2×2 grid, as the value of D gets smaller, the number of neurons required for 100% classification accuracy increases. This is because the decision boundaries need to bend more, leading to higher computational complexity.

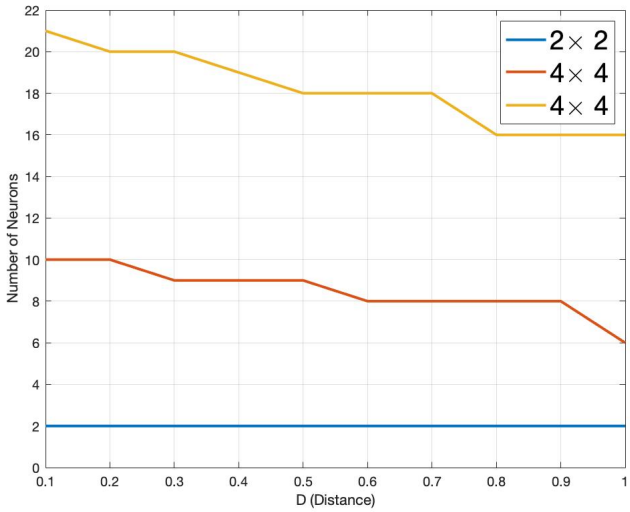

Figure 58: Total Number of Neurons vs. Distance

(d) Aim - Repeat (a) by pruning the network for complexity by employing the following criteria: upon completion of training, remove those neurons from the network for which the L2 norm of the associated weight vector is less than an $\epsilon = 10^{-3}$ and comment on the classification accuracy.

Methodology

To prune the network, we took our trained B and W, and calculated the norms corresponding to the weight vectors of all the neurons. Rather than comparing the absolute value of this norm with ϵ, we actual compare a relative value. The intuition is that if the norm of the weight vector of a neuron is ϵ times smaller than the maximum norm in the same layer, then this neuron is insignificant relatively. To get a measure of this maximum norm, we take the norm of the full weight matrix Wl corresponding to the neurons of a particular layer-l. Therefore, we make the following comparision -
let $A = [B\{l\}\ W\{l\}]$
if

$$\frac{norm(A(i,:))}{norm(A)} < \epsilon$$

then remove the i^{th} neuron in layer-l.

Observations

1. We started our experiments in part-a by estimating the minimum number of neurons that would be required for classification, which was $2N-2$ neurons for a $N \times N$ grid. In cases where $2N-2$ neurons were not sufficient for 100% classification accuracy, the number of neurons in the hidden layer were increased one by one until they proved to be sufficient. Because of this methodology, the number of

neurons that were used in every case were the minimum number of neurons that were required for complete classification. Therefore, our network didn't have any redundancy. Because of this, when the pruning algorithm was applied on the trained network, no neurons were removed. So, all the results after pruning are the same as before pruning. In summary, *pruning doesn't have any effect on the results of part-(a).*

2. To see the effect of algorithm, we took our 2×2 network and added a large number of neurons to it for redundancy. This network was then pruned for complexity. The results have been plotted below.

3. For this example, the classification accuracy remains the same.

4. The number of neurons that were pruned are only 2, leaving us with a network consisting of 100 hidden neurons. This number is quite large considering that only 2 hidden neurons are required for correct classification in the $N \times N$ case. This implies that our present pruning routine is inefficient.

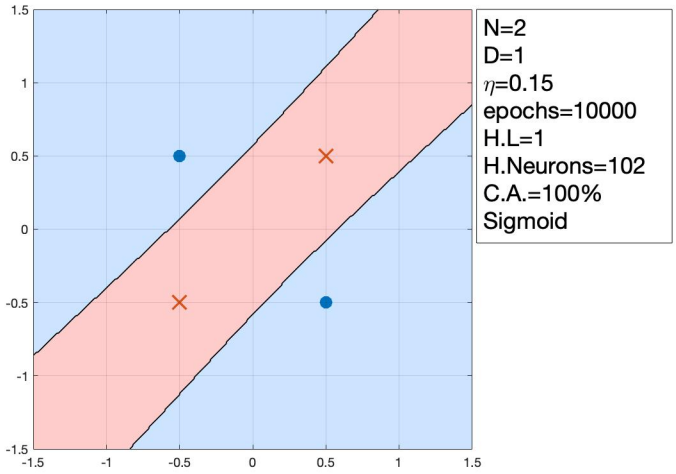

Figure 59: Original Network

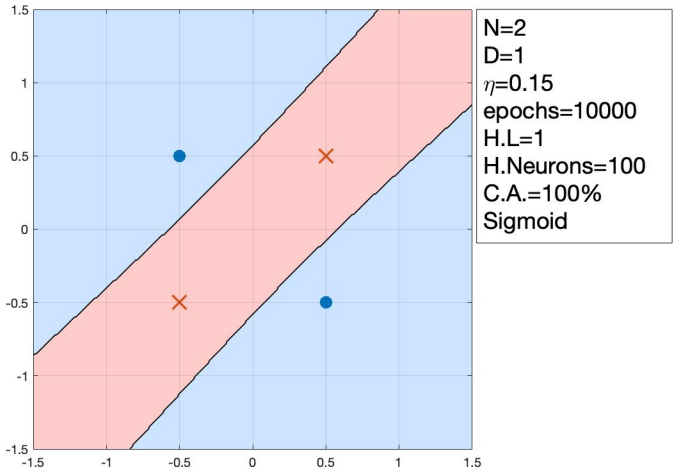

Figure 60: Pruned Network