Deep Learning

Perceptron, Neural Network, and Bag-of-Visual-Word

Praveen Kumar

Contents

List of Figures

List of Tables

CHAPTER 1

Classification

1.1 Separate the Linearly Separable Classes

In this problem, our objective is to separate the three linearly separable classes with the perceptron algorithm. For this problem, we have provided with a 2-dimensional linearly separable dataset of three classes where each class has 500 data points. Figure 1.1 plots the data for each class.

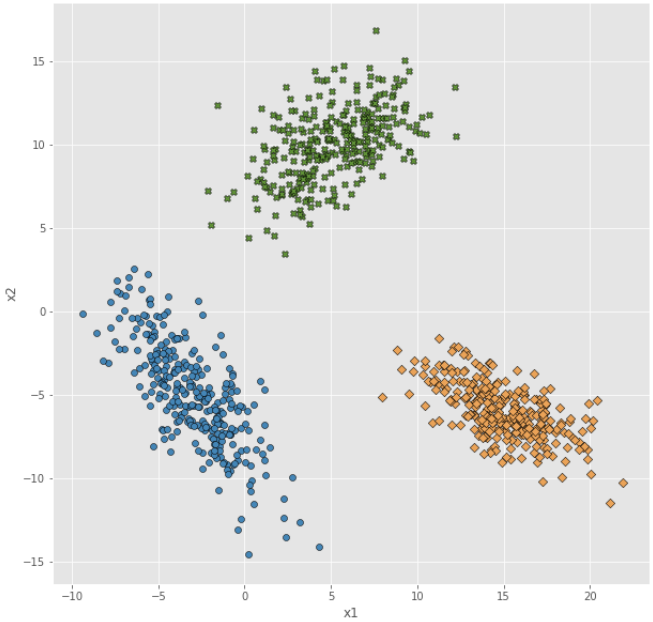


Figure 1.1: Linearly separable data scatter plot.

1.2 Perceptron with a Sigmoidal Activation Function

Develop a perceptron algorithm with a sigmoidal activation function that can be utilized for the classification of the linearly separable data.

Methodology

Data Preprocessing

In this experiment, we used the 2-dimensional linearly separable data where each class has 500 data points. The first 60% of three class data selected for training the algorithm, 20% of the data from each class as validation data, and remaining 20% of data from each class as test data. The input vector is augmented by adding 1 as the first element of the vector. The class labels are One-Hot encoded to distinguish from each other.

$$\begin{aligned}\hat{X} = [[1,\ x_1,\ x_2,\ 1,\ 0,\ 0,\ ]\\ [1,\ x_1,\ x_2,\ 0,\ 1,\ 0,\ ]\\ [1,\ x_1,\ x_2,\ 0,\ 0,\ 1,\ ]]\end{aligned}$$

For the hyperbolic tangent activation function, the class label One-Hot encoded by the -1, and 1 as follows:

$$\begin{aligned}\hat{X} = [[1,\ x_1,\ x_2,\ 1,\ -1,\ -1,\ ]\\ [1,\ x_1,\ x_2,\ -1,\ 1,\ -1,\ ]\\ [1,\ x_1,\ x_2,\ -1,\ -1,\ 1,\ ]]\end{aligned}$$

The input data and class label in the algorithm could be described as follows:

$$Input\ data(\hat{X}) = \hat{X}[:-3]$$

$$Class\ label = argmax(\hat{X}[-3:])$$

Perceptron

A Perceptron is an algorithm utilized for binary class classification. The perceptron determines whether an input value belongs to a particular class.

In simple terms, a perceptron is a single-layer neural network. They consist of four main parts, including input values represent by a vector, weights and bias, net sum, and an activation function applied on the net sum.

As seen in Figure 1.2, the input values represents by the input vector $[1,\ x_1,\ x_2, \ldots, x_n] \in \mathbb{R}^{d+1}$, where d is the dimension of the input data. Every input value in the input vector associate with a weight. The weights are represents by the symbol $w_0,\ w_1\ w_2 \ldots w_n$ in the figure. The symbol $\sum$ represents the net sum of the inputs multiplied by their associated weights. And the final node represents an activation function applied to the net sum.

$$\begin{aligned}Training\ data\ \hat{X} = [1,\ x_1,\ x_2, \ldots, x_n] \in \mathbb{R}^{d+1}\\ Weights\ W = [w_0,\ w_1,\ w_2, \ldots, w_n] \in \mathbb{R}^{d+1}\end{aligned}$$

where w_0 is a bias. The separating hyperplane represents by the following equation. The symbol $(\cdot)$ in the equation represent the dot product of two vectors.

$$g(x) = W^T \cdot \hat{X}$$

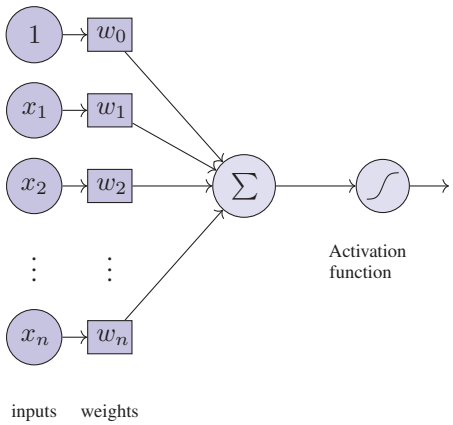


Figure 1.2: A single perceptron with an activation function.

$$= w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Finally, the activation function is applied to the $g(x)$ function or the net sum. The activation function is a non-linear transformation that we use on the net sum. In this experiment, we are using the sigmoidal activation function, and there are two types of sigmoidal activation function which are the logistic and hyperbolic tangent.

Logistic function is a widely used activation function. Figure 1.3 shows the plot of the logistic activation function and its derivative. The range of this function is (0, 1). It is defined as:

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

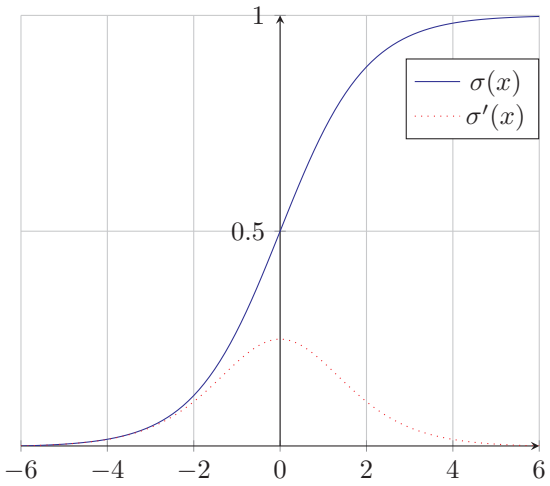


Figure 1.3: The plot of the logistic activation function.

Hyperbolic tangent function is also a widely used activation function. Figure 1.4 shows the plot of the hyperbolic tangent activation function and its derivative. The

range of this function is (-1, 1). It is defined as:

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

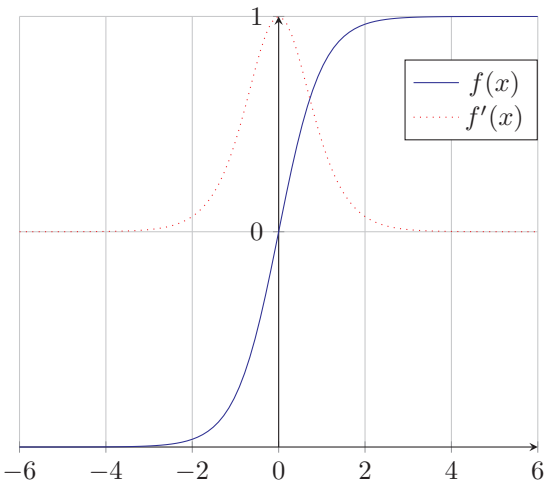


Figure 1.4: The plot of the hyperbolic tangent activation function.

Perceptron Algorithm

A single perceptron can classify the binary class. It can separate the two classes by drawing the decision boundary between them. In the given problem, the three classes are provided to classify. For that, we have developed the one-vs-one multi-class classification approach. The three classes can separate by the three perceptrons in the one-vs-one multi-class classification approach. For example, perceptron one can classify class one and class two, and perceptron two can classify class one and class three. Finally, perceptron three can classify class two and class three. Figure 1.5 shows the multi-class classification, where three perceptrons classify the three classes.

Every perceptron draws a decision boundary between two classes. As shown in Figure 1.6, it will return the positive value if the training example is above the decision boundary and return the negative if below the decision boundary. It will return zero if the training example lies on the decision boundary.

We can decide that the provided input training example belongs to the which class. The activation function is applied to every perceptron output. The activation function transforms the perceptron output value into the specific range. If the activation function output is greater than a 50 threshold value, it belongs to class one, otherwise class zero. The decision rule of the perceptron defined as follows:

$$\hat{y} = \begin{cases} 1, & \text{if f(x) > t,} \\ 0, & \text{otherwise.} \end{cases}$$

where $\hat{y}$ is a predicted class, $f(x)$ is a activation function output, and t represents the threshold value.

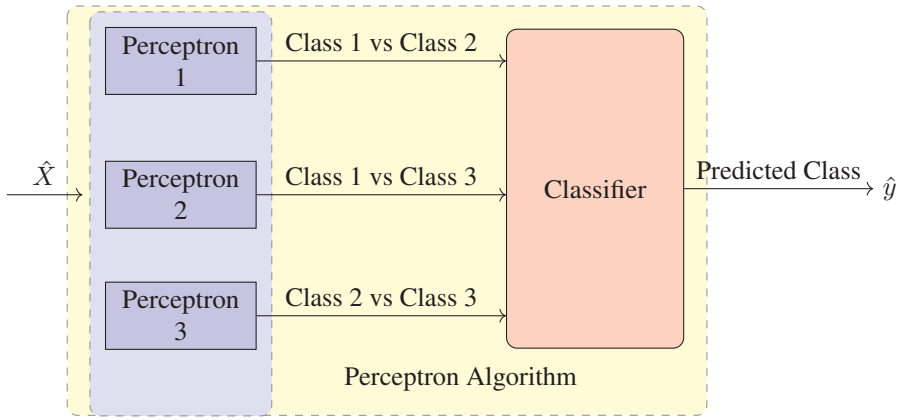


Figure 1.5: The perceptron algorithm for multi-class classification.

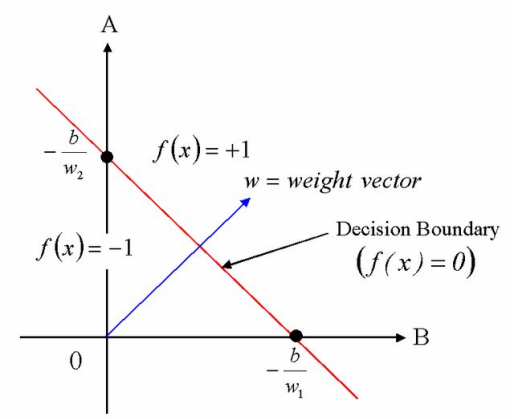


Figure 1.6: Decision boundary of single perceptron.

This experiment developed the three-class classifier with three perceptrons where perceptron returns 0 if data point below the decision boundary otherwise return 1. Bases on the three perceptrons, we have decided the three-class classifier decision rules as follows:

```
if  Perceptron1==0 and Perceptron2==0:
    Class=0
if  Perceptron1==1 and Perceptron3==1:
    Class=1
if  Perceptron3==0 and Perceptron1==0:
    Class=2
```

Weights Update Rule

After creating the classifier with the perceptron, the classifier could correctly classify the class or misclassify the class. If the classifier correctly classified the class, then there is no need to update the weights. Still, if it is misclassified in the class, there is a

need to update the weights. The update rule for the perceptron algorithm defines as follows:

$$W_{new} = W_{old} + (y - \hat{y})\ \eta\ x_i$$

where y is a true class label and $\hat{y}$ is a predicted class label. The symbol η represents the learning rate, and x_i is the input data point. For example, if the true class label is 1 and the predicted class label is also 1, then $1 - 1 = 0$, and there is no update in the weight. If the true class label is 0 and the predicted class label is also 1, then $0 - 1 = -1$ and subtract the $(\eta\ x_i)$ from the old weight.

Results

The newly developed perceptron algorithm runs with 300 numbers of epochs. The learning rate has been set at 0.01. The activation function has been selected logistic function or the hyperbolic tangent. The error was calculated after every epoch and updated the weights after every epoch. The error of every epoch is plotted in Figure 1.7 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

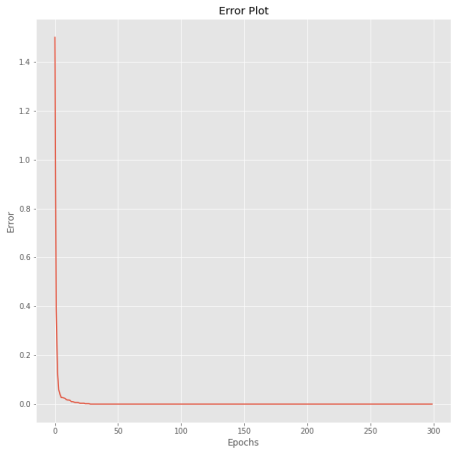


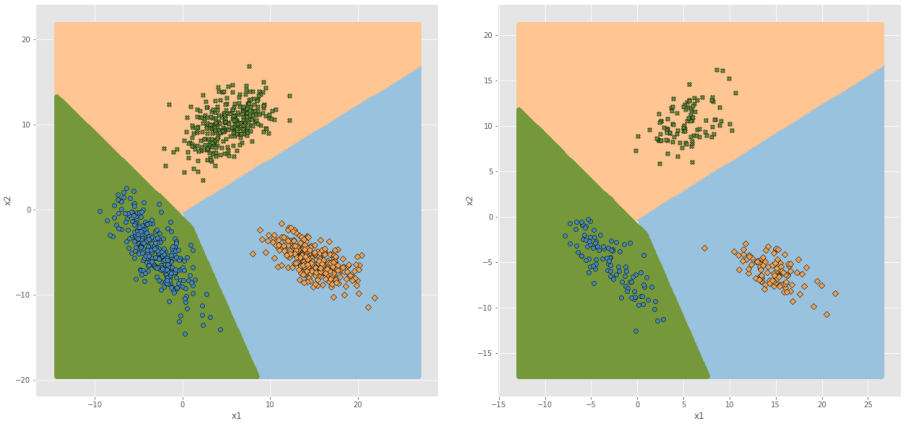
Figure 1.7: Average error of the epochs in the perceptron algorithm.

After trained the perceptron algorithm, we have generated the (500×500) data points to classify with the developed perceptron algorithm to plot the decision region. As illustrated in Figure 1.8, the decision region is represented by three different colors, orange, green, and blue, for the three classes. Figure 1.8a, Figure 1.8b, and Figure 1.8c illustrate the decision region of the three classes for the training, validation and testing, respectively.

The results of the perceptron algorithm in the training, validation, and testing dataset are reported in Table 1.1.

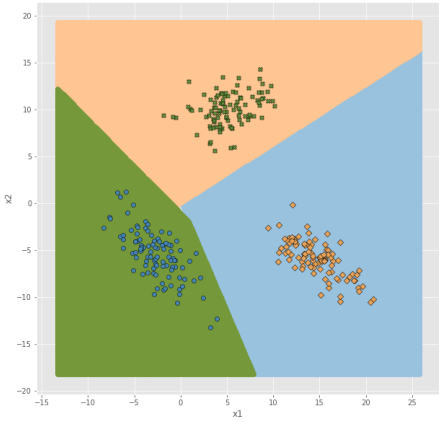
Figure 1.9a, Figure 1.9b, and Figure 1.9c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.

Table 1.2 show the classification accuracy for the training, validation, and testing dataset.



(a) Training data.

(b) Validation data.



(c) Testing data.

Figure 1.8: Decision boundary for the different dataset.

Table 1.1: The accuracy of the perceptron algorithm.

Algorithm	Training	Validation	Testing
Model 1	100%	100%	100%
Model 2	100%	100%	100%
Model 3	100%	100%	100%
Model 4	100%	100%	100%

Table 1.2: The classification accuracy for the different dataset.

Accuracy Measure	Training	Validation	Testing
Average Accuracy (%)	100	100	100
Average Recall (%)	100	100	100
Average Precision (%)	100	100	100
Average F-score (%)	100	100	100

Actual Class \ Predicted Class	1	2	3
1	300	0	0
2	0	300	0
3	0	0	300

(a) Training data.

Actual Class \ Predicted Class	1	2	3
1	100	0	0
2	0	100	0
3	0	0	100

(b) Validation data.

Actual Class \ Predicted Class	1	2	3
1	100	0	0
2	0	100	0
3	0	0	100

(c) Testing data.

Figure 1.9: Confusion matrix for the different dataset.

1.3 Separate the Nonlinearly Separable Classes with Perceptron

In this problem, our objective is to separate the three nonlinearly separable classes with the perceptron algorithm. For this problem, we have provided a 2-dimensional nonlinearly separable dataset of three classes where each class one and class two has 500 data points and class three has 700 data points. Figure 1.10 plots the data for each class.

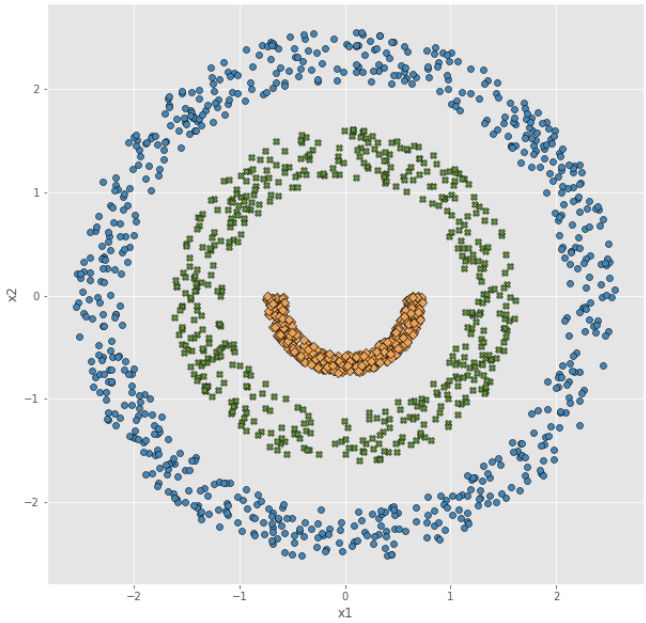


Figure 1.10: Nonlinearly separable data scatter plot.

Results

The newly developed perceptron algorithm in the previous section runs with 200 numbers of epochs. The learning rate has been set at 0.1. The activation function has been selected logistic function or the hyperbolic tangent. The error was calculated after every epoch and updated the weights after every epoch. The error of every epoch is plotted in Figure 1.11 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch. As can be seen in the Figure 1.11 the error in every epoch was fluctuating.

After trained the perceptron algorithm, we have generated the (500×500) data points to classify with the developed perceptron algorithm to plot the decision region. As illustrated in Figure 1.12, the decision region is represented by three different colors, orange, green, and blue, for the three classes. Figure 1.12a, Figure 1.12b, and Figure 1.12c illustrate the decision region of the three classes for the training, validation and testing, respectively.

Figure 1.13a, Figure 1.13b, and Figure 1.13c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.

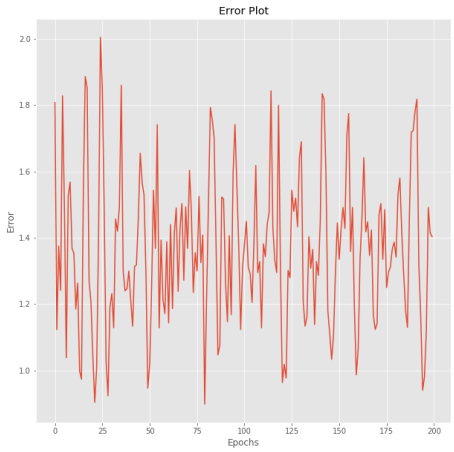
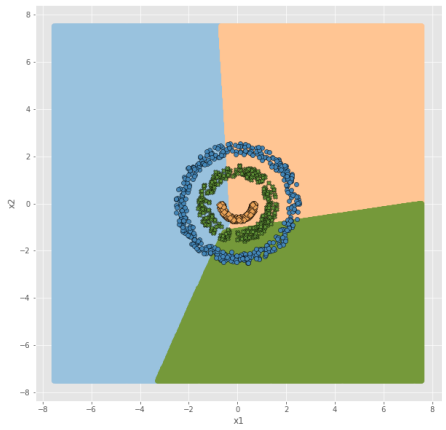


Figure 1.11: Average error of the epochs in the perceptron algorithm.

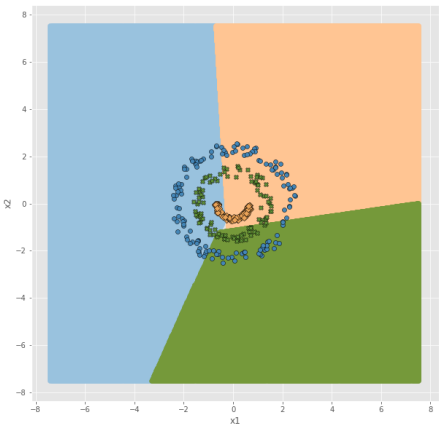
Table 1.3 show the classification accuracy for the training, validation, and testing dataset.

Table 1.3: The classification accuracy for the different dataset.

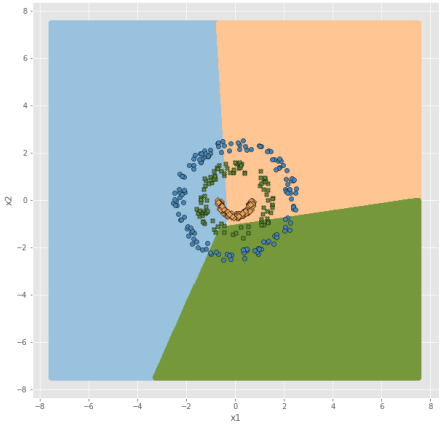
Accuracy Measure	**Training**	**Validation**	**Testing**
Average Accuracy (%)	32	32	29
Average Recall (%)	30	31	27
Average Precision (%)	23	24	20
Average F-score (%)	25	26	23



(a) Training data.



(b) Validation data.



(c) Testing data.

Figure 1.12: Decision boundary for the different dataset.

		Predicted Class		
		1	2	3
Actual Class	1	0	203	97
	2	12	147	141
	3	32	213	175

(a) Training data.

		Predicted Class		
		1	2	3
Actual Class	1	0	72	28
	2	7	49	44
	3	8	71	61

(b) Validation data.

		Predicted Class		
		1	2	3
Actual Class	1	0	68	32
	2	1	38	61
	3	8	71	61

(c) Testing data.

Figure 1.13: Confusion matrix for the different dataset.

1.4 Separate the Nonlinearly Separable Classes with Multilayer Feed-forward Neural Networks

In this problem, our objective is to separate the three nonlinearly separable classes with the multilayer feed-forward neural network algorithm. For this problem, we have provided a 2-dimensional nonlinearly separable dataset of three classes where each class one and class two has 500 data points and class three has 700 data points. Figure 1.14 plots the data for each class.

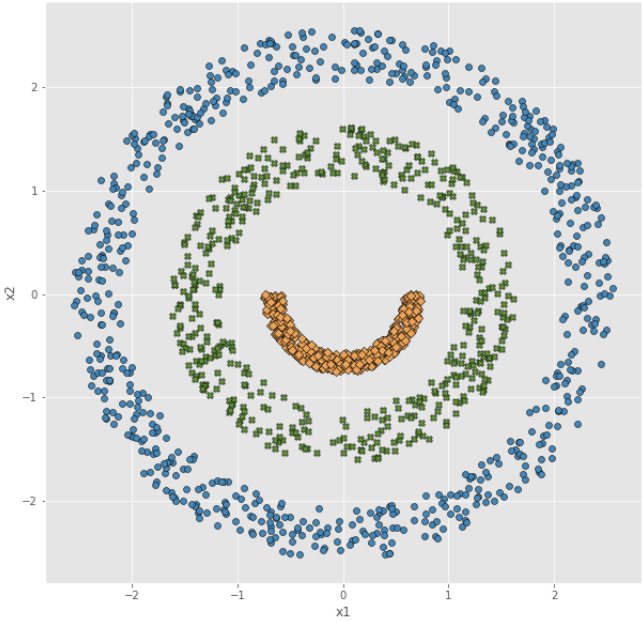


Figure 1.14: Nonlinearly separable data scatter plot.

Methodology

Data Preprocessing

In this experiment, we used the 2-dimensional nonlinearly separable data where first two class have 500 data points and third class has the 700 data points. The first 60% of three class data selected for training the algorithm, 20% of the data from each class as validation data, and remaining 20% of data from each class as test data. The input vector is augmented by adding 1 as the first element of the vector. The class labels are One-Hot encoded to distinguish from each other and append to the augmented input vector. For example, the train example for the three classes shown as follows:

$$\begin{aligned}\hat{X} = [[1,\ x_1,\ x_2,\ 1,\ 0,\ 0,\ ]\\ [1,\ x_1,\ x_2,\ 0,\ 1,\ 0,\ ]\\ [1,\ x_1,\ x_2,\ 0,\ 0,\ 1,\ ]]\end{aligned}$$

For the hyperbolic tangent activation function, the class label One-Hot encoded by the -1, and 1 as follows:

$$\hat{X} = [[1,\ x_1,\ x_2,\ 1,\ -1,\ -1,\ ]$$
$$[1,\ x_1,\ x_2,\ -1,\ 1,\ -1,\ ]$$
$$[1,\ x_1,\ x_2,\ -1,\ -1,\ 1,\ ]]$$

The input data and class label in the algorithm could be described as follows:

$$Input\ data(\hat{X}) = \hat{X}[\,:-3]$$

$$Class\ label = argmax(\hat{X}[-3:])$$

Figure 1.15 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

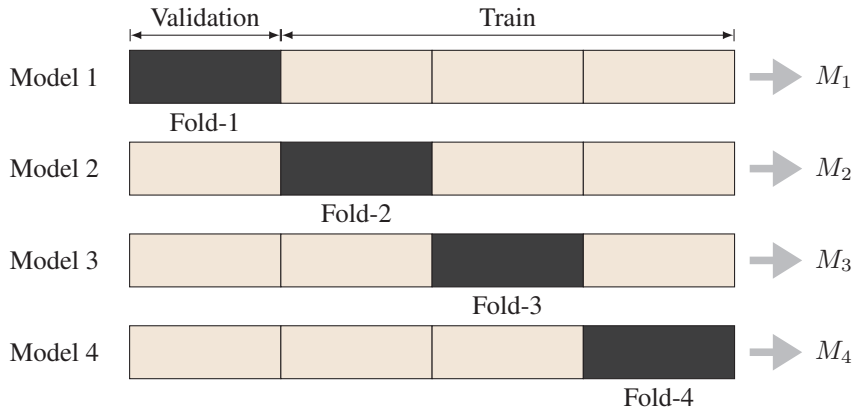


Figure 1.15: 4-cross validation of the training data

Multilayer Feed-forward Neural Networks

A multilayer feed-forward neural networks (MLFFNN) is a feed-forward artificial neural network. At a minimum, it has three layers: an input layer, a hidden layer, and an output layer. Each node of one layer is connected to all other nodes of the next layer, except the input nodes. Every layer uses an activation function (see Figure 1.16). It applies a supervised learning technique called backpropagation for training. The connection between every two nodes of MLFFNN has a certain weight. The weights between all the nodes are updated based on the error (the difference between the target value and predicted value), which is done through backpropagation.

At output node j error in n^{th} training point is represented by:

$$e_j(n) = d_j(n) - y_j(n) \tag{1.1}$$

Where d is the actual value and y is the predicted value. The node weights are changed such that the error in the entire output, which is defined as:

$$\epsilon(n) = \frac{1}{2}\sum_j e_j^2(n) \tag{1.2}$$

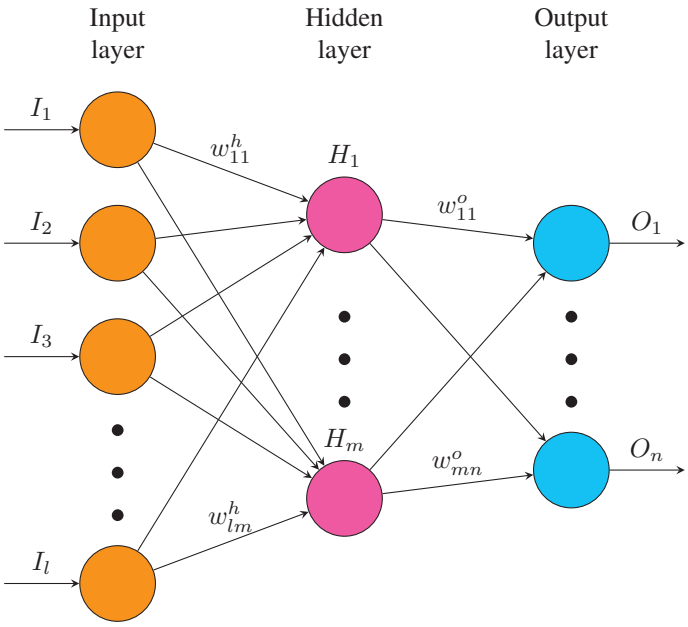


Figure 1.16: Architecture graph of an MLFFNN

Using the gradient descent method, the change in each weight is:

$$\Delta W_{ji}(n) = -\eta \frac{\delta \epsilon(n)}{\delta v^j(n)} y_i(n) \tag{1.3}$$

where y_i is the output of the previous neuron and 'η' is the learning rate of MLFFNN, which is selected such that the weights converge without any oscillations. The derivative $\frac{\delta \epsilon(n)}{\delta v^j(n)}$ at output layer is a function of the error term (Eq. 1.2) and derivative of the activation function.

Accuracy Measure

The multilayer feed-forward neural networks predict a floating value for the every class at the output nodes. Thus, an error could be calculated between the actual class and the predicted class values. We computed the mean squared error (MSE), which measured the average deviation of the actual data points from the predicted data points. The MSE of the algorithm could be calculated with the following formula:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Actual\ class - Predicted\ class)^2 \tag{1.4}$$

where, the actual class is the actual class from the training examples, the predicted class is the predicted class by the algorithm, and n is the total number of training examples.

Backpropagation

To update the weights of the MLFFNN, the backpropagation algorithm used that feed the error in a backward direction to update the weights in every layer. Our goal with

backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

For example, We want to know how much a change in w_{11}^o affects the total error (E_{total}). To find out the participation of the w_{11}^o in the total error E_{total} we could calculate the partial derivative of the E_{total} with respect to w_{11}^o that can be defined as $\frac{\partial E_{total}}{\partial w_{11}^o}$.

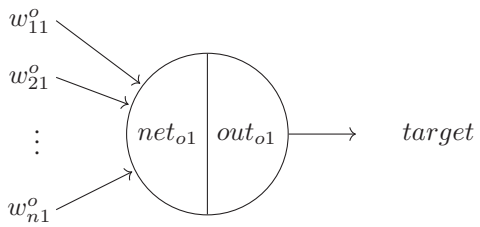


Figure 1.17: Backpropagation at the output layer node

$$\frac{\partial E_{total}}{\partial w_{11}^o} = \frac{\partial E_{total}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}} \cdot \frac{\partial net_{o1}}{\partial w_{11}^o}$$

$$Error(E) = \frac{1}{2}(target - out)^2$$

$$E_{total} = E_{o1} + E_{o2} + \cdots + E_{on}$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(tareget_{o1} - out_{o1})$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) \; for \; the \; logistic \; function$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = 1 - tanh(out_{o1})^2 \; for \; the \; tanh \; function$$

$$\frac{\partial net_{o1}}{\partial w_{11}^o} = out_{h1}$$

$$\frac{\partial E_{total}}{\partial w_{11}^o} = -(tareget_{o1} - out_{o1}) \cdot out_{o1}(1 - out_{o1}) \cdot out_{h1}$$

The $-(tareget_{o1} - out_{o1}) \cdot out_{o1}(1 - out_{o1})$ is always fixed for a neuron in a layer. It could be defined as a δ_o.

$$\delta_{o1} = -(tareget_{o1} - out_{o1}) \cdot out_{o1}(1 - out_{o1}) \cdot out_{h1}$$

$$\frac{\partial E_{total}}{\partial w^o_{11}} = \delta_{o1} \cdot out_{h1}$$

For every neuron in the output layer we calculated the δ_o and define a delta (δ_o) vector for the output layer. For our problem we have three class classification so, the we have the three value of the delta δ_{o1}, δ_{o2}, and δ_{o3}.

$$\delta_o = \begin{bmatrix} \delta_{o1} \\ \delta_{o2} \\ \vdots \\ \delta_{on} \end{bmatrix}$$

The weight w^o_{11} could be updated as follow:

$$w^o_{11} = w^o_{11} - \eta \cdot \frac{\partial E_{total}}{\partial w^o_{11}}$$

where η is the learning rate, and it should be defined between the range (0,1).

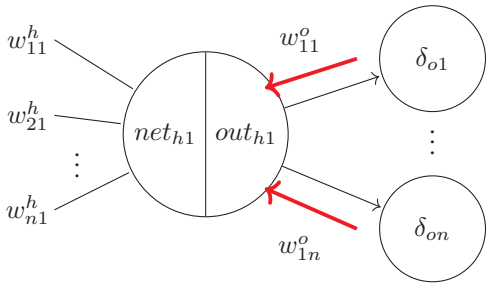


Figure 1.18: Backpropagation at the hidden layer node

Now we are updating the weights of the hidden layer. For example, We want to know how much a change in w^h_{11} affects the E_{total}. To find out the participation of the w^h_{11} in the total error E_{total} we could calculate the partial derivative of the E_{total} with respect to w^h_{11} that can be defined as $\frac{\partial E_{total}}{\partial w^h_{11}}$.

$$\frac{\partial E_{total}}{\partial w^h_{11}} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial net_{h1}} \cdot \frac{\partial net_{h1}}{\partial w_{11^h}}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} + \cdot + \frac{\partial E_{on}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} \cdot \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}}$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_{11}^{o}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) \; for \; the \; logistic \; function$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = 1 - tanh(out_{h1})^2 \; for \; the \; tanh \; function$$

$$\frac{\partial net_{h1}}{\partial w_{11}^{h}} = x_1$$

$$\frac{\partial E_{total}}{\partial w_{11}^{h}} = (\sum_{o} \delta_o \cdot w_{ho}^{h}) \cdot out_{h1}(1 - out_{h1}) \cdot x_1$$

The $(\sum_o \delta_o \cdot w_{ho}^h) \cdot out_{h1}(1 - out_{h1})$ is always fixed for a neuron in a hidden layer. It could be defined as a δ_h. The x_1 represent the input data associated with the w_{11}^h.

$$\delta_{h1} = (\sum_{o} \delta_o \cdot w_{ho}^{h}) \cdot out_{h1}(1 - out_{h1})$$

$$\frac{\partial E_{total}}{\partial w_{11}^{h}} = \delta_{h1} \cdot x_1$$

For every neuron in the hidden layer we calculated the δ_h and define a delta (δ_h) vector for the hidden layer. For n number of neuron in the hidden layer there were n number of deltas $\delta_{h1}, \delta_{h2}, \ldots \delta_{hn}$.

$$\delta_h = \begin{bmatrix} \delta_{h1} \\ \delta_{h2} \\ \vdots \\ \delta_{hn} \end{bmatrix}$$

The backpropagation algorithm developed like that first, we calculated the delta vector for the output layer. Then we can calculate the deltas for the n number of hidden layers. Finally, we updated all the weights of the neural networks based on the calculated deltas.

Model Calibration

The MLFFNN developed like that we could pass the model architecture explicitly and automatically create the number of input and output node, number of hidden layers,

and number of neurons per hidden layer. For example, architecture = [2, 10, 3] defined as the number of input nodes is two. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the three nodes, predicting the output for the three classes. Similarly, architecture = [2, 10, 20, 3] defined as the number of hidden layers are two with the 10 and 20 neurons, respectively. We varied the hyper parameters of the model and the range of the hyper parameters explained the Table 1.4.

Table 1.4: Range of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3, 4
Number of Hidden Layers	1, 2
Nodes in Input Layer	2
Nodes in Output Layer	3
Nodes in Hidden Layers	5, 8,10,15,20
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	logistic, hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9

We developed the grid search algorithm and supply all combinations of the parameters to the model and record the model's accuracy on the given parameters.

Results

The newly developed MLFFNN algorithm runs with ranges of parameters. We select those parameters that were maximize the validation accuracy. The error was calculated after every epoch and updated the weights after every epoch. The error of every epoch is plotted in Figure 1.19 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

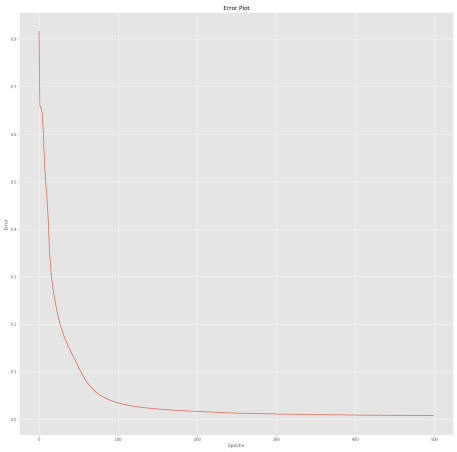


Figure 1.19: Average error of the epochs in the MLFFNN algorithm.

The results of the MLFFNN in the training, validation, and testing dataset are reported in Table 1.5.

Table 1.5: The accuracy of the MLFFNN algorithm.

Algorithm	**Training**	**Validation**	**Testing**
Model 1	100%	100%	100%
Model 2	100%	99%	100%
Model 3	100%	100%	100%
Model 4	100%	100%	100%

Table 1.6 shows the best set of parameters where the MLFFNN algorithm yielded the best accuracy. For example, the MLFFNN generated the best accuracy with the eight neurons in the hidden layer, 500 epochs, and 0.1 learning rate.

Table 1.6: Best set of parameters for the MLFFNN.

Parameter	**Optimised Values**
Number of Layers	3
Number of Hidden Layers	1
Nodes in Input Layer	2
Nodes in Output Layer	3
Nodes in Hidden Layers	8
Number of Epochs	500
Activation Functions	logistic
Input Shuffle	Yes
Learning Rate	0.1

After trained the MLFFNN algorithm, we have generated the (500×500) data points to classify with the developed MLFFNN algorithm to plot the decision region. As illustrated in Figure 1.20, the decision region is represented by three different colors, orange, green, and blue, for the three classes. Figure 1.20a, Figure 1.20b, and Figure 1.20c illustrate the decision region of the three classes for the training, validation and testing, respectively.

Figure 1.21 shows the output of the eight neurons of the hidden layer. The x-axis and y-axis are the two dimensions of the input dataset, and the z-axis is the output of the neuron. As can see from Figure 1.21, every neuron transforms some part of the input data into a higher dimension. In the higher dimension, it is easy for the classifier to classify the class data.

Figure 1.22 shows the output of the three neurons of the output layer. As can see from Figure 1.22, every neuron transforms the class data into a higher dimension. In the higher dimension, one class is separate from the other two classes. So, every neuron in the output layer separates one class from other classes.

Figure 1.23a, Figure 1.23b, and Figure 1.23c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.

Table 1.7 show the classification accuracy for the training, validation, and testing dataset.

(a) Training data.

(b) Validation data.

(c) Testing data.

Figure 1.20: Decision boundary for the different dataset.

Table 1.7: The classification accuracy for the different dataset.

Accuracy Measure	**Training**	**Validation**	**Testing**
Average Accuracy (%)	100	100	100
Average Recall (%)	100	100	100
Average Precision (%)	100	100	100
Average F-score (%)	100	100	100

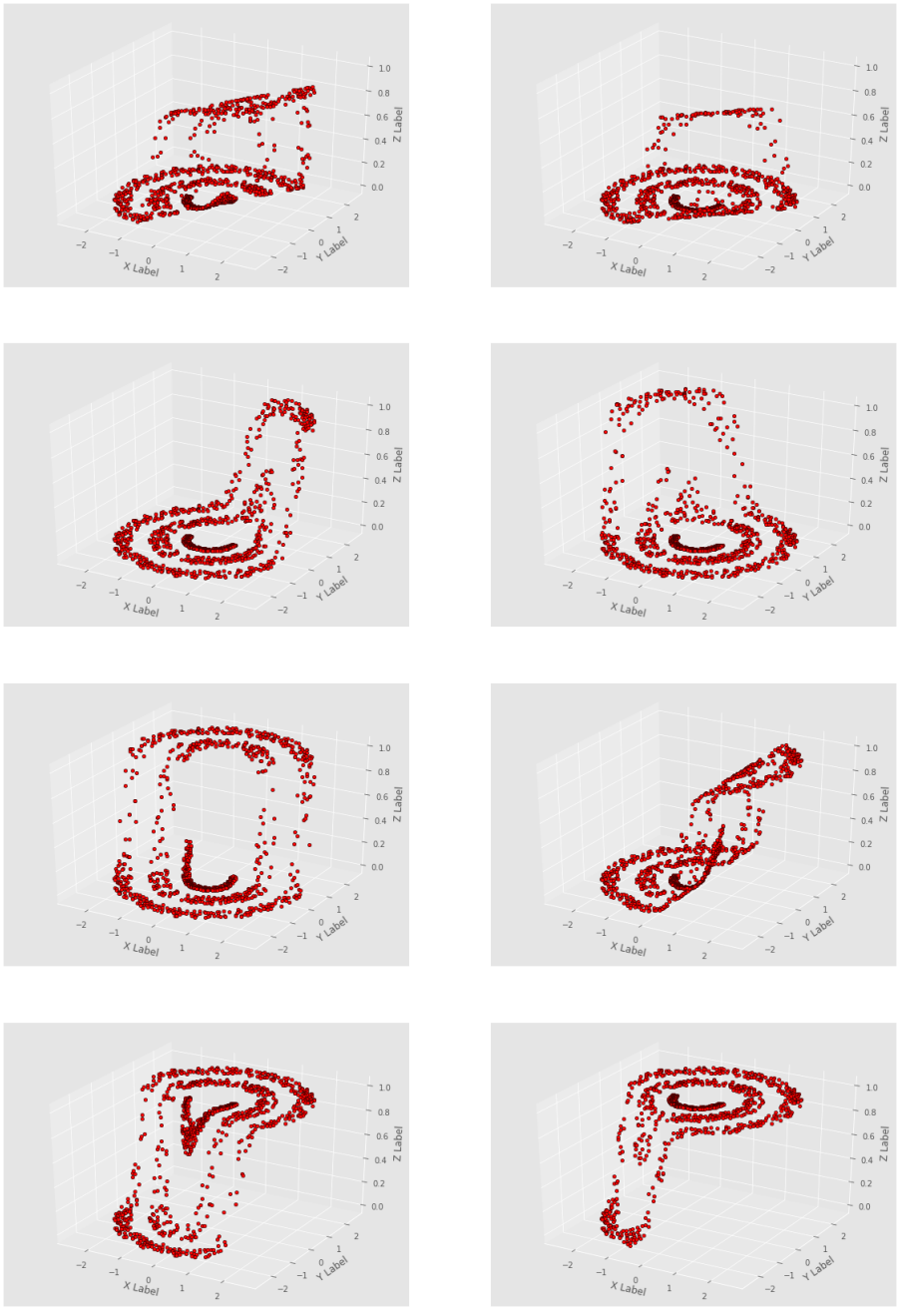


Figure 1.21: Output of the eight neuron in the hidden layer.

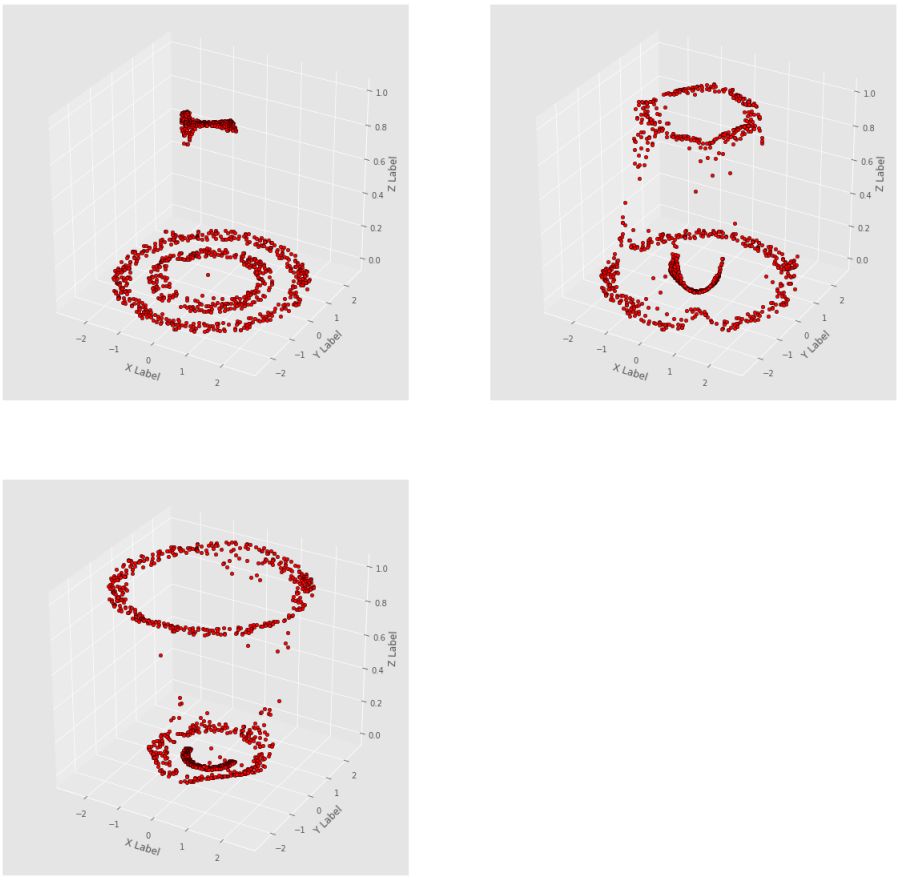


Figure 1.22: Output of three neuron in the output layer.

Actual Class \ Predicted Class	1	2	3
1	300	0	0
2	0	300	0
3	0	0	300

(a) Training data.

Actual Class \ Predicted Class	1	2	3
1	100	0	3
2	0	100	0
3	0	0	100

(b) Validation data.

Actual Class \ Predicted Class	1	2	3
1	100	0	0
2	0	100	0
3	0	0	100

(c) Testing data.

Figure 1.23: Confusion matrix for the different dataset.

1.5 Separate the Linearly Separable Class with the MLFNN

In this problem, our objective is to separate the three linearly separable classes with the MLFNN algorithm. For this problem, we have provided with a 2-dimensional linearly separable dataset of three classes where each class has 500 data points. Figure 1.24 plots the data for each class.

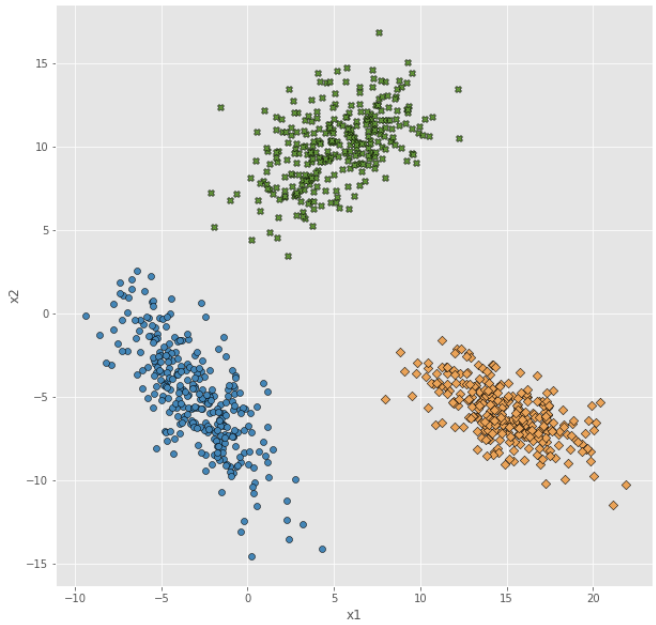


Figure 1.24: Linearly separable data scatter plot.

Model Calibration

The newly developed MLFFNN model in the previous section utilized to classify the linearly separable data we passed the model architecture explicitly and model automatically create the number of input and output node, number of hidden layers, and number of neurons per hidden layer. For example, architecture = [2, 10, 3] defined as the number of input nodes is two. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the three nodes, predicting the output for the three classes. Similarly, architecture = [2, 10, 20, 3] defined as the number of hidden layers are two with the 10 and 20 neurons, respectively. We varied the hyper parameters of the model and the range of the hyper parameters explained the Table 1.8.

Figure 1.25 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

We developed the grid search algorithm and supply all combinations of the parameters to the model and record the model's accuracy on the given parameters.

Table 1.8: Range of parameters in the MLFFNN.

Parameter	Range of Values
Number of Layers	2, 3
Number of Hidden Layers	0, 1
Nodes in Input Layer	2
Nodes in Output Layer	3
Nodes in Hidden Layers	0, 5, 10
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	logistic, hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9

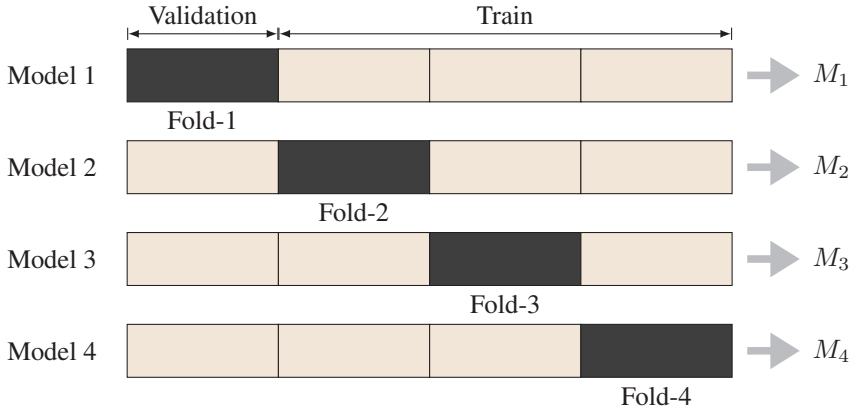


Figure 1.25: 4-cross validation of the training data

Results

The newly developed MLFFNN algorithm runs with ranges of parameters. We select those parameters that were maximize the validation accuracy. The error was calculated after every epoch and updated the weights after every epoch. The error of every epoch is plotted in Figure 1.26 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

The results of the MLFFNN in the training, validation, and testing dataset are reported in Table 1.9.

Table 1.9: The accuracy of the MLFFNN algorithm on the linearly separable data.

Algorithm	Training	Validation	Testing
Model 1	100%	100%	100%
Model 2	100%	100%	100%
Model 3	100%	100%	100%
Model 4	100%	100%	100%

Table 1.10 shows the best set of parameters where the MLFFNN algorithm yielded

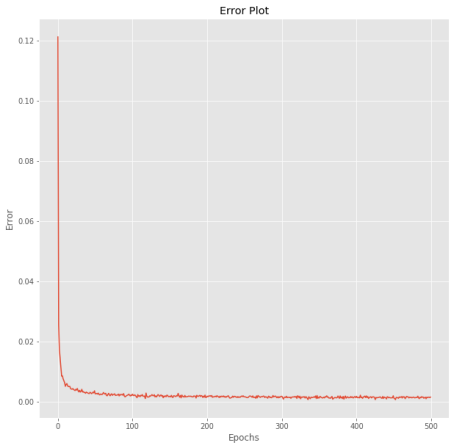


Figure 1.26: Average error of the epochs in the MLFFNN algorithm.

the best accuracy. For example, the MLFFNN generated the best accuracy with the three neurons in the output layer, 500 epochs, and 0.1 learning rate.

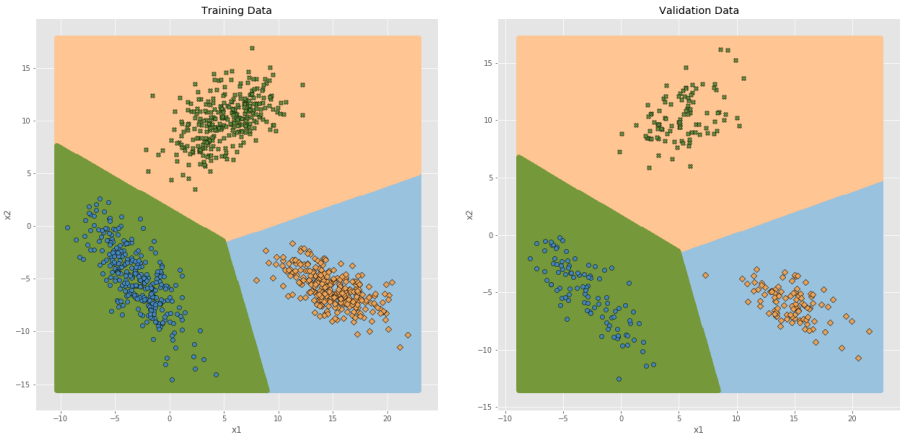
Table 1.10: Best set of parameters for the MLFFNN.

Parameter	**Optimised Values**
Number of Layers	2
Number of Hidden Layers	0
Nodes in Input Layer	2
Nodes in Output Layer	3
Nodes in Hidden Layers	0
Number of Epochs	500
Activation Functions	logistic
Input Shuffle	Yes
Learning Rate	0.1

After trained the MLFFNN algorithm, we have generated the (500×500) data points to classify with the developed MLFFNN algorithm to plot the decision region. As illustrated in Figure 1.27, the decision region is represented by three different colors, orange, green, and blue, for the three classes. Figure 1.27a, Figure 1.27b, and Figure 1.27c illustrate the decision region of the three classes for the training, validation and testing, respectively.

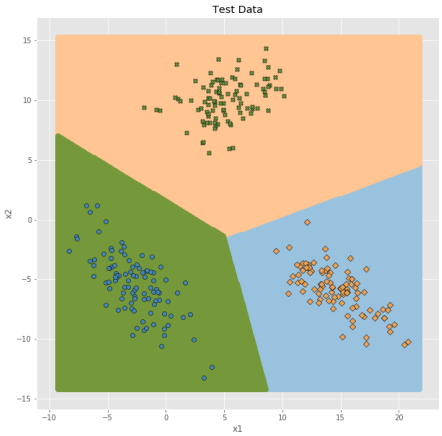
Figure 1.28 shows the output of the three neurons of the output layer. The x-axis and y-axis are the two dimensions of the input dataset, and the z-axis is the output of the neuron. As can see from Figure 1.28, every neuron transforms some part of the input data into a higher dimension. In the higher dimension, one class is separate from the other two classes. So, every neuron in the output layer separates one class from other classes.

Figure 1.29a, Figure 1.29b, and Figure 1.29c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.



(a) Training data.

(b) Validation data.



(c) Testing data.

Figure 1.27: Decision boundary for the different dataset.

Table 1.11 show the classification accuracy for the training, validation, and testing dataset.

Table 1.11: The classification accuracy for the different dataset.

Accuracy Measure	**Training**	**Validation**	**Testing**
Average Accuracy (%)	100	100	100
Average Recall (%)	100	100	100
Average Precision (%)	100	100	100
Average F-score (%)	100	100	100

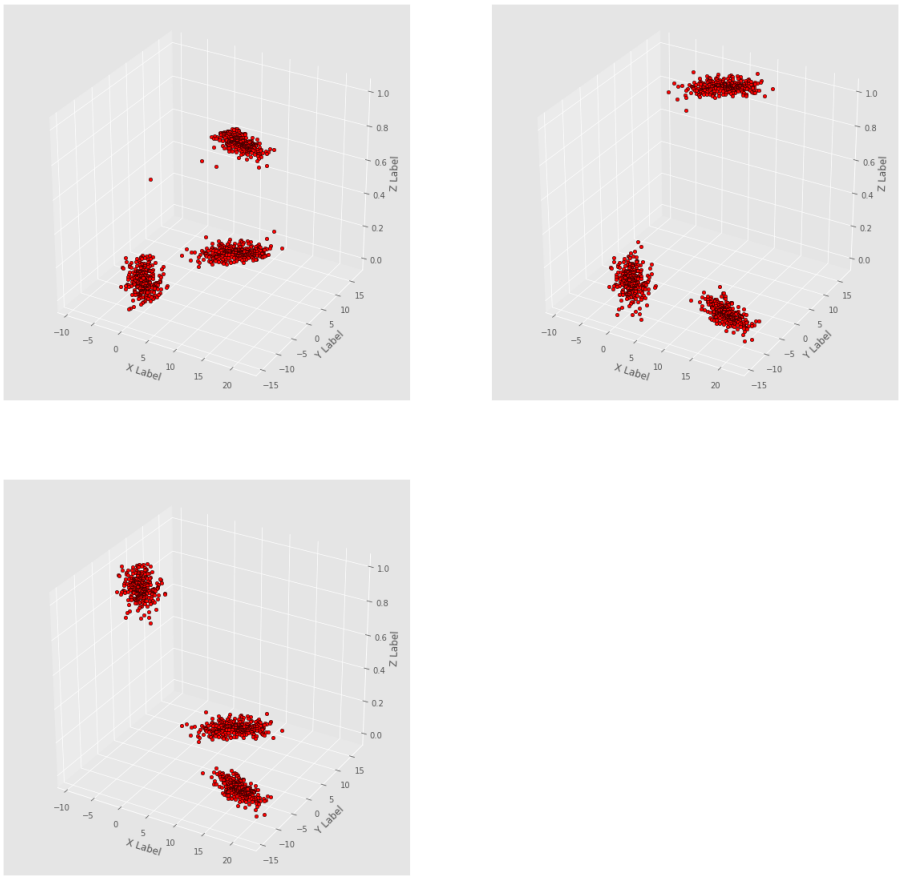


Figure 1.28: Output of the three neuron in the output layer.

Actual Class \ Predicted Class	1	2	3
1	300	0	0
2	0	300	0
3	0	0	300

(a) Training data.

Actual Class \ Predicted Class	1	2	3
1	100	0	0
2	0	100	0
3	0	0	100

(b) Validation data.

Actual Class \ Predicted Class	1	2	3
1	100	0	0
2	0	100	0
3	0	0	100

(c) Testing data.

Figure 1.29: Confusion matrix for the different dataset.

1.6 Classify the Image dataset

In this problem, our objective is to separate the Scene Image Data corresponding to Three different classes with perceptron with sigmoidal Activation function. For this problem we had been provided 40 Images as training dataset and remaining 10 images as validation dataset. Figure 1.30 showing the sample images from the each classes.

(a) Class synagogue outdoor.

(b) Class desert vegetation.

(c) Class auditorium.

Figure 1.30: The sample image from the three different classes.

Methodology

Data Preprocessing

In this experiment, we used the 50 image data provided for training and validation on the data-set. we had classify the image data-set into three classes. class 1 as synagogue outdoor class 2 as desert vegetation and class 3 as auditorium Every image had three-color channel RGB (red, green, blue). Every image from class 1, class 2, and class 3 in the dataset had a different shape, for example, height and width. To train the images in the model, we need to reshape all the images to get every image's fixed dimension to pass as input in the model to train the images. We use the same size of the image for training because the algorithm MLFFNN requires the images to be in the same size for training. We took the different sizes for the images by reshaping the image to determine which size gives the best result in the model training. After we normalized the image data by dividing the images by 255, we flattened the image data to convert it into a one-dimensional vector to feed into the network. The first 40 images from 50 images were selected for training and rest 10 images for validation on the data-set. The

input vector is augmented by adding 1 as the first element of the vector. The data labels are One-Hot encoded to distinguish from each other.

Sample dataset when sigmoid activation function selected as a logistic function.

$$\begin{aligned}\hat{X} = [&[1,\ x_1,\ x_2, \ldots, x_n\ 1,\ 0,\ 0,\ ]\\ &[1,\ x_1,\ x_2, \ldots, x_n\ 0,\ 1,\ 0,\ ]\\ &[1,\ x_1,\ x_2, \ldots, x_n\ 0,\ 0,\ 1,\ ]]\end{aligned}$$

Sample dataset when sigmoid activation function selected as a hyperbolic tangent function.

$$\begin{aligned}\hat{X} = [&[1,\ x_1,\ x_2, \ldots, x_n\ 1,\ -1,\ -1,\ ]\\ &[1,\ x_1,\ x_2, \ldots, x_n\ -,\ 1,\ -1,\ ]\\ &[1,\ x_1,\ x_2, \ldots, x_n\ -1,\ -1,\ 1,\ ]]\end{aligned}$$

Figure 1.31 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

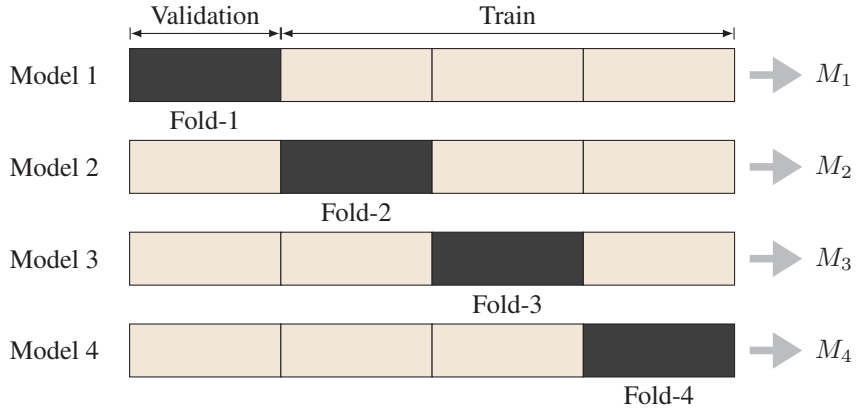


Figure 1.31: 4-cross validation of the training data

Multi-layer feed-forward Neural Network

The newly developed MLFFNN with the backpropagation algorithm in the last section utilized for the classification of the images in this model we are varying the hyper parameters which we are discussing in the next section.

Model Calibration

The newly developed MLFFNN model used to classify the image data. In this model we passed the architecture explicitly. The model automatically creates the number of input, output nodes, and the number of hidden layers, and neurons per hidden layer. For example, architecture = [x, 10, 3] defined as the number of input nodes is x, where x represents the size of flatten the image. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the three nodes, predicting

the three classes' output. Similarly, architecture = [x, 10, 20, 3] defined as the number of hidden layers are two with the 10 and 20 neurons, respectively. We had varied the size of images from 10 to 200 with step size 10. For example, if the shape of the image is (10×10), then flatten image size is $10 \times 10 \times 3$, so the number of input nodes is 300.

Table 1.12: Range of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3, 4
Number of Hidden Layers	2, 3
Nodes in Input Layer	As per the size of image
Nodes in Output Layer	3
Nodes in Hidden Layers	10 to 100 with step size 10
Size of the Image	(x, x) x = 10 to 200 with step size 10
Flatten image Size	$(x \times x \times 3)$
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	logistic, hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 1.12 to the MLFFNN model and record the model's accuracy on the given parameters. We select those parameters that were maximize the validation accuracy. The error of every epoch for the MLFFNN with the best set of parameters is plotted in Figure 1.32 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

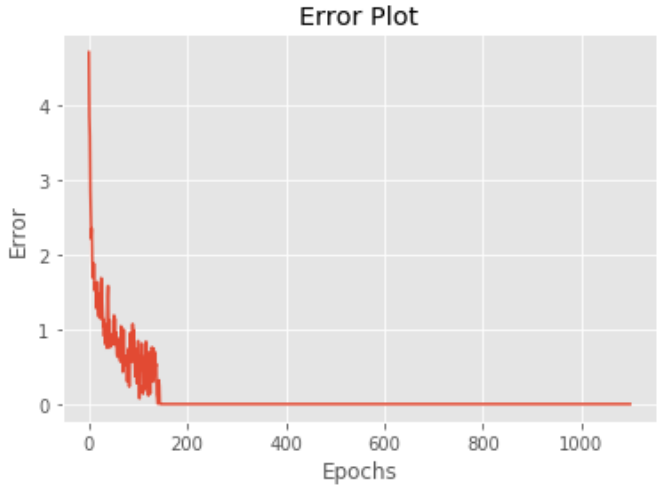


Figure 1.32: Average error of the epochs for the image classification.

The results of the MLFFNN in the training, validation, and testing dataset are reported in Table 1.13.

Table 1.13: The accuracy of the MLFFNN algorithm on the linearly separable data.

Algorithm	**Training**	**Validation**	**Testing**
Model 1	100%	81%	23%
Model 2	100%	78%	25%
Model 3	100%	83%	24%
Model 4	100%	83%	22%

The Table 1.14 showing the best set of parameters where the average error for the MLFFNN algorithm is less.

Table 1.14: Best set of parameters in the MLFFNN for image classification.

Parameter	**Best set of Values**
Number of Layers	4
Number of Hidden Layers	2
Nodes in Input Layer	300
Nodes in Output Layer	3
Nodes in Hidden Layers	10 in hidden layer 1 , 80 in hidden layer 2
Size of the Image	$(10, 10)$
Flatten image Size	300
Number of Epochs	1100
Activation Functions	hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1

Figure 1.33a, Figure 1.33b, and Figure 1.33c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.

Table 1.15 show the classification accuracy for the training, validation, and testing dataset.

Table 1.15: The classification accuracy for the different dataset.

Accuracy Measure	**Training**	**Validation**	**Testing**
Average Accuracy (%)	100	100	100
Average Recall (%)	100	83	22
Average Precision (%)	100	86	23
Average F-score (%)	100	86	23

Actual Class \ Predicted Class	1	2	3
1	36	0	0
2	0	36	0
3	0	0	36

(a) Training data.

Actual Class \ Predicted Class	1	2	3
1	11	1	0
2	2	10	3
3	2	1	9

(b) Validation data.

Actual Class \ Predicted Class	1	2	3
1	29	7	14
2	7	2	41
3	11	37	2

(c) Testing data.

Figure 1.33: Confusion matrix for the different dataset.

CHAPTER 2

Regression

In this problem, our objective is to fit a linear function on the univariate data, for the regression problem we have provided with a 2-dimensional dataset with 1001 data points (see Figure 2.1).

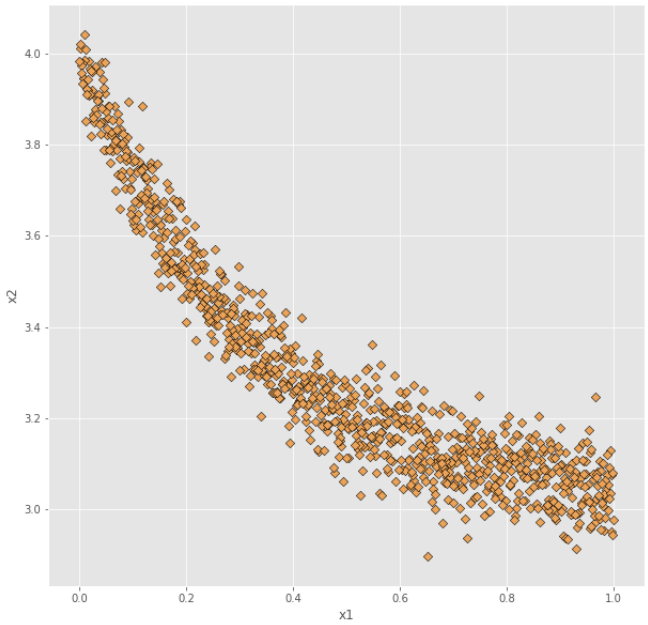


Figure 2.1: The 2-dimensional univariate data scatter plot.

2.1 Perceptron with a Linear Activation Function

Develop a perceptron algorithm with a linear activation function that can be utilized for the regression problem.

Methodology

Data Preprocessing

In this experiment, we used the 2-dimensional data with 1001 data points. The first 60% of three class data selected for training the algorithm, 20% of the data as validation data, and remaining 20% of data for testing the algorithm. The input vector is augmented by adding 1 as the first element of the vector.

$$\hat{X} = [1,\ x_1]$$

where x_1 is a univariate independent variable and we need to predict the dependent variable using this independent variable.

Perceptron for Regression

A Perceptron is an algorithm utilized for regression. The perceptron with the linear activation function predicts a linear combination of the input values and the associated weights.

As seen in Figure 2.2, the input values represents by the input vector $[1,\ x_1,\ x_2, \ldots, x_n] \in \mathbb{R}^{d+1}$, where d is the dimension of the input data. Every input value in the input vector associate with a weight. The weights are represents by the symbol $w_0,\ w_1\ w_2 \ldots w_n$ in the figure. The symbol $\sum$ represents the net sum of the inputs multiplied by their associated weights. And the final node represents a linear activation function applied to the net sum.

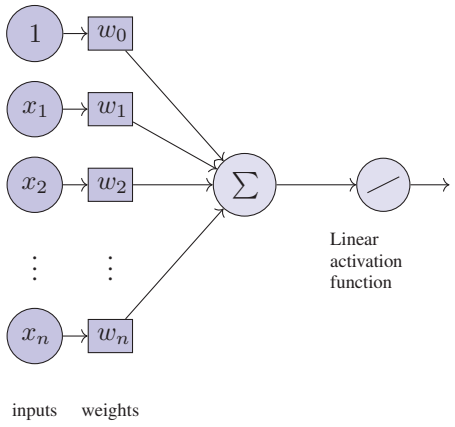


Figure 2.2: A single perceptron with an activation function.

$$Training\ data\ \hat{X} = [1,\ x_1,\ x_2, \ldots, x_n] \in \mathbb{R}^{d+1}$$
$$Weights\ W = [w_0,\ w_1,\ w_2, \ldots, w_n] \in \mathbb{R}^{d+1}$$

where w_0 is a bias. The separating hyperplane represents by the following equation. The symbol $(\cdot)$ in the equation represent the dot product of two vectors.

$$g(x) = W^T \cdot \hat{X}$$

$$= w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Finally, the linear activation function is applied to the $g(x)$ function or the net sum. In this experiment, we are using the linear activation function. The identity function is a widely used linear activation function. The identity function returns the outputs that are provided as input.

Figure 2.3 shows the plot of the identity activation function and its derivative. The range of this function is $(-\infty,\ \infty)$. It is defined as:

$$f(x) = x$$

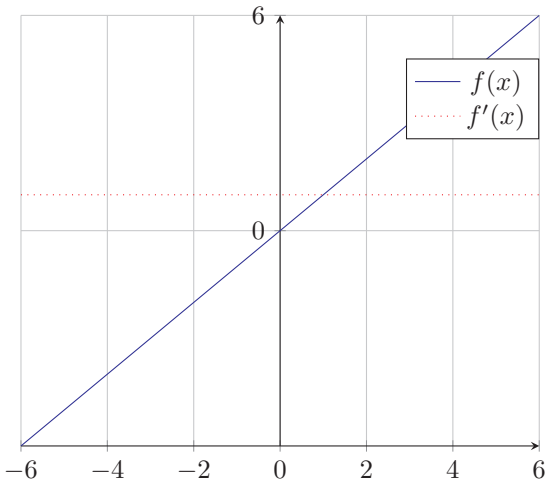


Figure 2.3: The plot of the identity activation function.

Weights Update Rule

After developing the perceptron for regression, the perceptron predicts the output nearby the target value. The error could be calculated by difference between target value and predicted value. The cost function for the regression problem in the perceptron algorithm defined as follows:

$$Cost = \frac{1}{m}\sum(\hat{y} - y)^2$$

The derivative of the cost function is following:

$$= -2 \cdot (\hat{y} - y)$$

The derivative of the error with respect to weight in the perceptron algorithm is defined as follows:

$$\frac{\partial Error}{\partial w_i} = -2 \cdot (\hat{y} - y) \cdot w_i$$

The weights update rule for the perceptron algorithm defines as follows:

$$W_{new} = W_{old} - \eta \frac{\partial Error}{\partial w_i}$$

where y is a true value and $\hat{y}$ is a predicted value by perceptron algorithm. The symbol η represents the learning rate, and x_i is the input data point.

Model Calibration

After developing the perceptron algorithm, we need to tune some of the parameters to reduce the error. Table 2.1 shows the different range of parameters to be varied in the algorithm.

Table 2.1: Range of parameters in the MLFFNN.

Parameter	Range of Values
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	Identity activation function
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9
Accuracy Measure	Mean Square Error (MSE)

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 2.4 to the perceptron algorithm and record the model's error on the given parameters. We select those parameters that were reducing the validation error. The error of every epoch for the algorithm with the best set of parameters is plotted in Figure 2.4 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

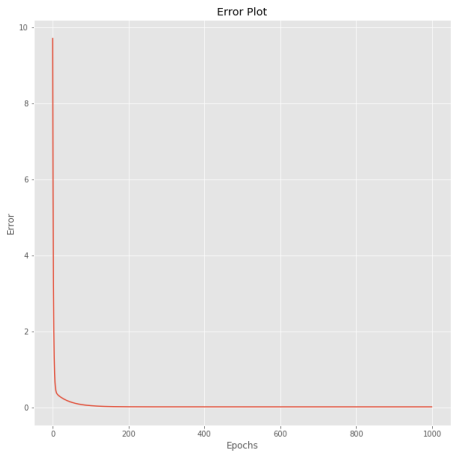


Figure 2.4: Average error of the epochs for the univariate regression.

The results of the perceptron algorithm in the training, validation, and testing dataset are reported in Table 2.2.

Table 2.2: The error (MSE) of the perceptron algorithm for the univariate dataset.

Algorithm	**Training**	**Validation**	**Testing**
Perceptron for univariate regression	0.012	0.012	0.011

The Table 2.3 showing the best set of parameters where the average error for the MLFFNN algorithm is less.

Table 2.3: Best set of parameters for the perceptron algorithm.

Parameter	**Best set of values**
Number of Epochs	1000
Activation Functions	Identity activation function
Input Shuffle	Yes
Learning Rate	0.1
Accuracy Measure	Mean Square Error (MSE)

Figure 2.5 shows the fitted regression line on the univariate dataset. The single perceptron with the linear activation function only can fit a line to the univariate data.

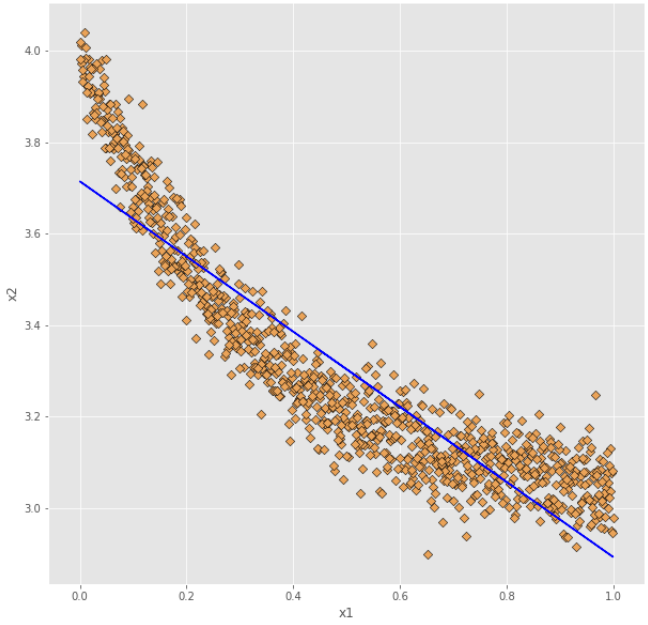
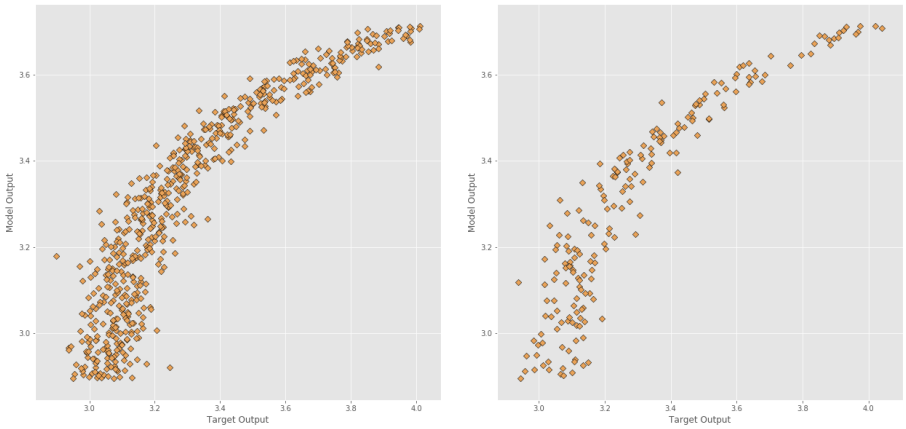


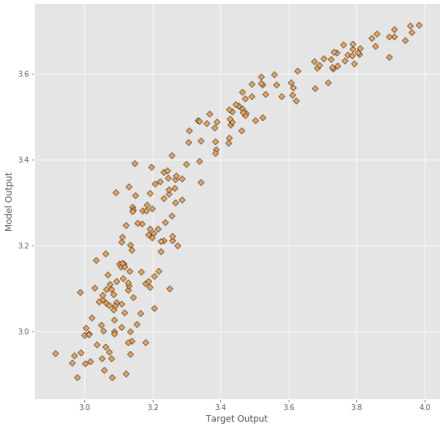
Figure 2.5: Fitted regression line on the univariate dataset.

Figure 2.6 shows the model output and target out scatter plot, where the x-axis is the target output, and the y-axis is the model output. If the model shows less error, then the scatter plot is the diagonal.



(a) Training data.

(b) Validation data.



(c) Testing data.

Figure 2.6: Model output and target output scatter plot for the different data.

2.2 MLFFNN Algorithm for the Regression on the Univariate Data Set

A multilayer feed-forward neural networks (MLFFNN) is a feed-forward artificial neural network. At a minimum, it has three layers: an input layer, a hidden layer, and an output layer. Each node of one layer is connected to all other nodes of the next layer, except the input nodes. Every layer uses an activation function (see Figure 2.7). It applies a supervised learning technique called backpropagation for training. The connection between every two nodes of MLFFNN has a certain weight. The weights between all the nodes are updated based on the error (the difference between the target value and predicted value), which is done through backpropagation.

For the regression problem the MLFFNN has one neuron in the output layer to predict the one variable. The activation function in the output neuron should be linear activation function for prediction the dependent variable range.

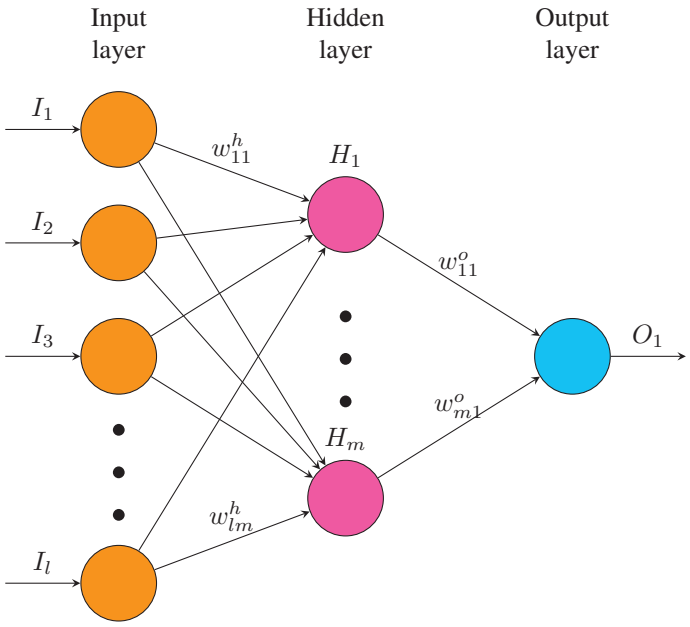


Figure 2.7: Architecture graph of an MLFFNN for the regression.

At output node j error in n^{th} training point is represented by:

$$e_j(n) = d_j(n) - y_j(n) \tag{2.1}$$

Where d is the actual value and y is the predicted value. The node weights are changed such that the error in the entire output, which is defined as:

$$\epsilon(n) = \frac{1}{2}\sum_j e_j^2(n) \tag{2.2}$$

Using the gradient descent method, the change in each weight is:

$$\Delta W_{ji}(n) = -\eta \frac{\delta\epsilon(n)}{\delta v^j(n)} y_i(n) \tag{2.3}$$

where y_i is the output of the previous neuron and 'η' is the learning rate of MLFFNN, which is selected such that the weights converge without any oscillations. The derivative $\frac{\delta\epsilon(n)}{\delta v^j(n)}$ at output layer is a function of the error term (Eq. 2.2) and derivative of the activation function.

Accuracy Measure

The multilayer feed-forward neural networks predict a floating value at the output nodes. Thus, an error could be calculated between the actual value and the predicted value. We computed the mean squared error (MSE), which measured the average deviation of the actual data points from the predicted values. The MSE of the algorithm could be calculated with the following formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Actual\ value - Predicted\ vlaue)^2 \tag{2.4}$$

where, the actual value is the actual value from the training examples, the predicted value is the predicted value by the algorithm, and n is the total number of training examples.

Backpropagation

To update the weights of the MLFFNN, the backpropagation algorithm used that feed the error in a backward direction to update the weights in every layer. Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

For example, We want to know how much a change in w^o_{11} affects the total error (E_{total}). To find out the participation of the w^o_{11} in the total error E_{total} we could calculate the partial derivative of the E_{total} with respect to w^o_{11}. The output layer has one node and the activation function is the linear activation (identity function) in the output node here. The derivative of the identity function is the constant.

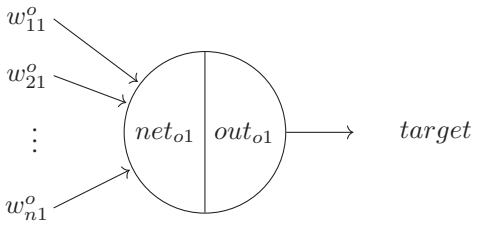


Figure 2.8: Backpropagation at the output layer node

$$\frac{\partial E_{total}}{\partial w^o_{11}} = \frac{\partial E_{total}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}} \cdot \frac{\partial net_{o1}}{\partial w^o_{11}}$$

$$E_{total} = \frac{1}{n}\sum_{i=1}^{n}(target - out)^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -2(tareget_{o1} - out_{o1})$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = 1 \ \ (for\ the\ identity\ function)$$

$$\frac{\partial net_{o1}}{\partial w^o_{11}} = out_{h1}$$

$$\frac{\partial E_{total}}{\partial w^o_{11}} = \frac{-2}{n}(tareget_{o1} - out_{o1}) \cdot 1 \cdot out_{h1}$$

The $\frac{-2}{n}(tareget_{o1} - out_{o1})$ is always fixed for a neuron in a layer. It could be defined as a δ_o.

$$\delta_{o1} = \frac{-2}{n}(tareget_{o1} - out_{o1})$$

$$\frac{\partial E_{total}}{\partial w^o_{11}} = \delta_{o1} \cdot out_{h1}$$

The weight w^o_{11} could be updated as follow:

$$w^o_{11} = w^o_{11} - \eta \cdot \frac{\partial E_{total}}{\partial w^o_{11}}$$

where η is the learning rate, and it should be defined between the range (0,1).

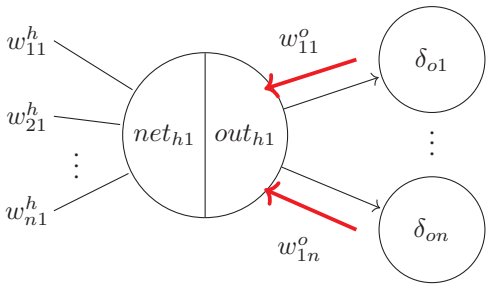


Figure 2.9: Backpropagation at the hidden layer node

Now we are updating the weights of the hidden layer. For example, We want to know how much a change in w^h_{11} affects the E_{total}. To find out the participation of the w^h_{11} in the total error E_{total} we could calculate the partial derivative of the E_{total} with respect to w^h_{11} that can be defined as $\frac{\partial E_{total}}{\partial w^h_{11}}$.

$$\frac{\partial E_{total}}{\partial w_{11}^{h}} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial net_{h1}} \cdot \frac{\partial net_{h1}}{\partial w_{11^{h}}}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{total}}{\partial net_{o1}} \cdot \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{total}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}}$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_{11}^{o}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1})\ for\ the\ logistic\ function$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = 1 - tanh(out_{h1})^2\ for\ the\ tanh\ function$$

$$\frac{\partial net_{h1}}{\partial w_{11}^{h}} = x_1$$

$$\frac{\partial E_{total}}{\partial w_{11}^{h}} = (\sum_{o} \delta_o \cdot w_{ho}^{h}) \cdot out_{h1}(1 - out_{h1}) \cdot x_1$$

The $(\sum_o \delta_o \cdot w_{ho}^{h}) \cdot out_{h1}(1 - out_{h1})$ is always fixed for a neuron in a hidden layer. It could be defined as a δ_h. The x_1 represent the input data associated with the w_{11}^{h}.

$$\delta_{h1} = (\sum_{o} \delta_o \cdot w_{ho}^{h}) \cdot out_{h1}(1 - out_{h1})$$

$$\frac{\partial E_{total}}{\partial w_{11}^{h}} = \delta_{h1} \cdot x_1$$

For every neuron in the hidden layer we calculated the δ_h and define a delta (δ_h) vector for the hidden layer. For n number of neuron in the hidden layer there were n number of deltas $\delta_{h1}, \delta_{h2}, \ldots \delta_{hn}$.

$$\delta_h = \begin{bmatrix} \delta_{h1} \\ \delta_{h2} \\ \vdots \\ \delta_{hn} \end{bmatrix}$$

The backpropagation algorithm developed like that first, we calculated the delta vector for the output layer. Then we can calculate the deltas for the n number of

hidden layers. Finally, we updated all the weights of the neural networks based on the calculated deltas.

Model Calibration

The newly developed MLFFNN model used to fit the regression curve on the univariate data. In this model we passed the architecture explicitly. The model automatically creates the number of input, output nodes, and the number of hidden layers, and neurons per hidden layer. For example, architecture = [1, 10, 1] defined as the number of input nodes is 1, where 1 represents the one input variable. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the one nodes, predicting the one value.

Figure 2.10 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

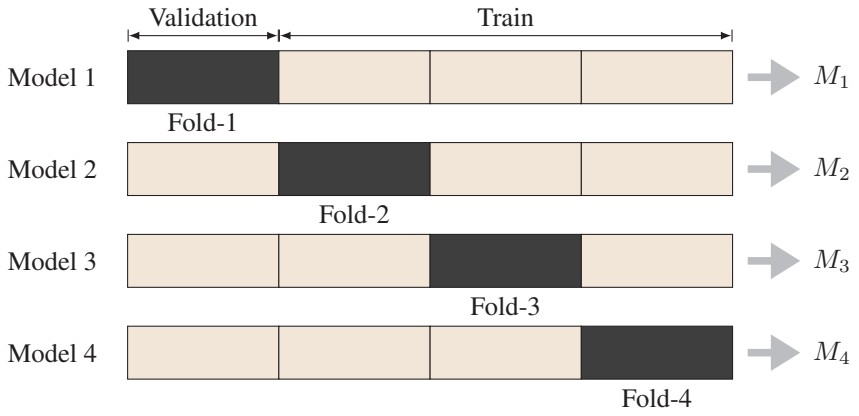


Figure 2.10: 4-cross validation of the training data

Table 2.4 presenting the different range of parameters varied in the MLFFNN.

Table 2.4: Range of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3, 4
Number of Hidden Layers	1, 2
Nodes in Input Layer	1
Nodes in Output Layer	1
Nodes in Hidden Layers	3 to 100 with step size 10
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	logistic, hyperbolic tangent, Identity (Output layer)
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 2.4 to the perceptron algorithm and record the model's error on the given parameters. We select those parameters that were reducing the validation error. The error of every epoch for the algorithm with the best set of parameters is plotted in Figure 2.11 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch. Figure 2.12 showing the MSE of the MLFFNN model with different complexity, where x-axis is the number of nodes in the hidden layer.

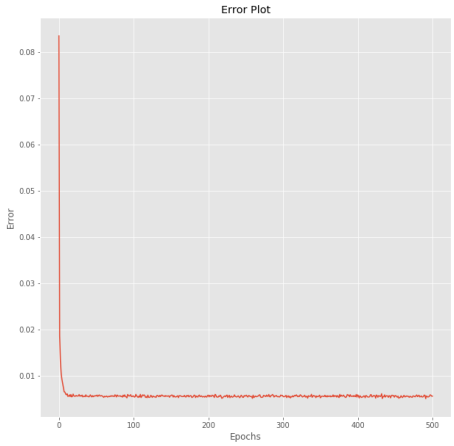


Figure 2.11: Average error of the epochs for the univariate regression in MLFFNN.

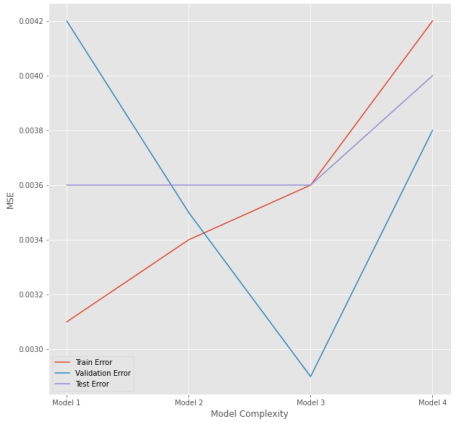


Figure 2.12: MSE of the MLFFNN model with different complexity.

The results of the MLFFNN algorithm in the training, validation, and testing dataset are reported in Table 2.5.

The Table 2.6 showing the best set of parameters where the average error for the MLFFNN algorithm is less.

Figure 2.13 shows the fitted regression curve on the univariate dataset. The MLFFNN with the linear activation function on the output node can fit a curve to

Table 2.5: The error (MSE) of the perceptron algorithm for the univariate dataset.

Algorithm	Training	Validation	Testing
Model 1	0.0046	0.003	0.003
Model 2	0.0056	0.004	0.005
Model 3	0.0036	0.003	0.004
Model 4	0.0036	0.004	0.004

Table 2.6: Best set of parameters for the perceptron algorithm.

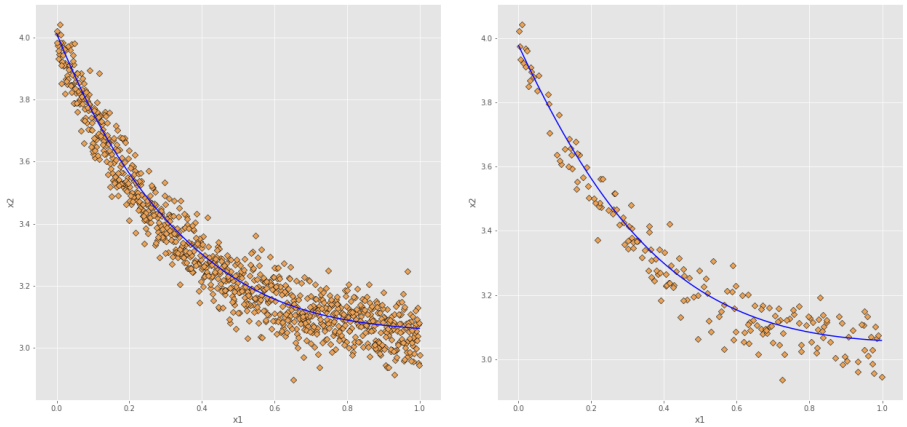
Parameter	Best set of values
Number of Layers	3
Number of Hidden Layers	1
Nodes in Input Layer	1
Nodes in Output Layer	1
Nodes in Hidden Layers	3
Number of Epochs	50
Activation Functions	hyperbolic tangent, Identity (Output layer)
Input Shuffle	Yes
Learning Rate	0.1

the univariate data to minimize data error. As seen in Figure 2.13, the MLFFNN model output draw with a blue line, where the x-axis is the model input data, and the y-axis is the model output and target value.

Figure 2.14 shows the model output and target out scatter plot, where the x-axis is the target output, and the y-axis is the model output. If the model shows less error, then the scatter plot is the diagonal.

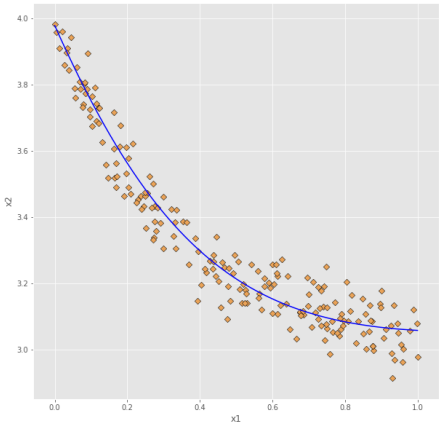
Figure 2.15, Figure 2.16, and Figure 2.17 shows the output of the hidden layer for training, validation, and testing dataset.

Figure 2.18 shows the output of the single node at output layer for training, validation, and testing dataset.



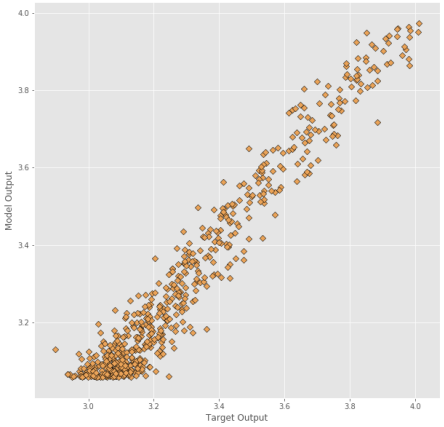
(a) Training data.

(b) Validation data.

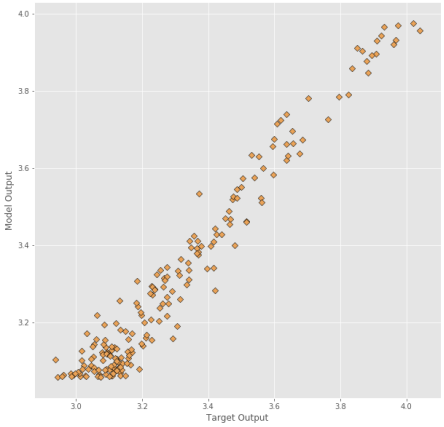


(c) Testing data.

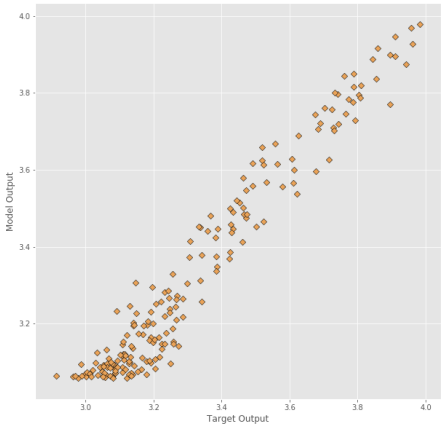
Figure 2.13: Plot of the Model output and the actual output scatter plot for the different data.



(a) Training data.



(b) Validation data.



(c) Testing data.

Figure 2.14: Model output and target output scatter plot for the different data.

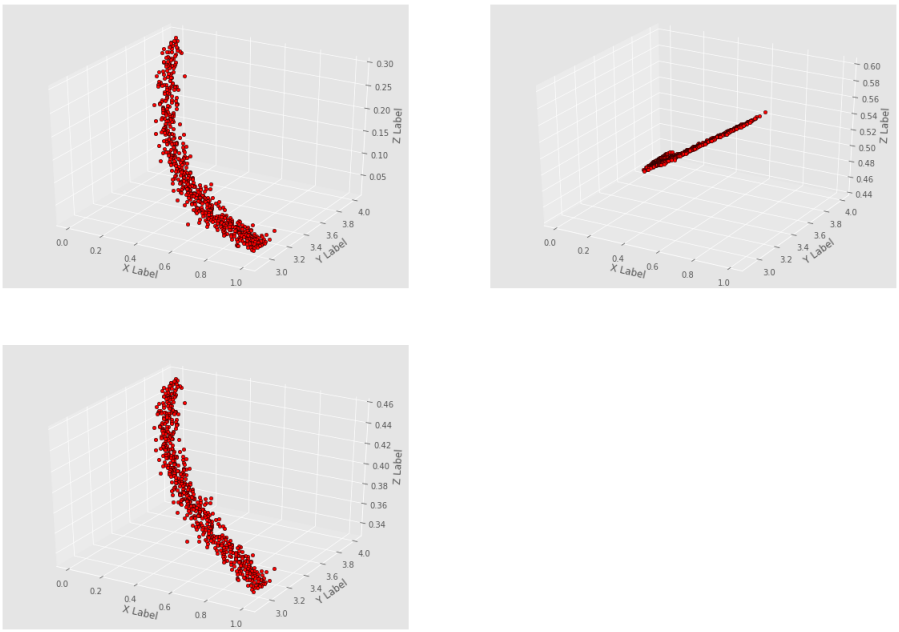


Figure 2.15: Output of the three nodes of the hidden layer for training data.

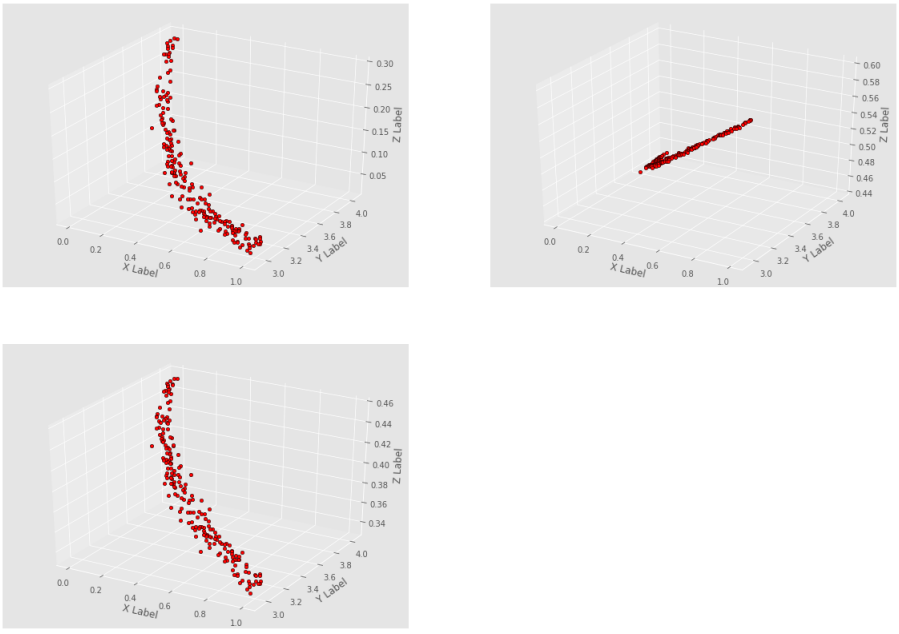


Figure 2.16: Output of the three nodes of the hidden layer for validation data.

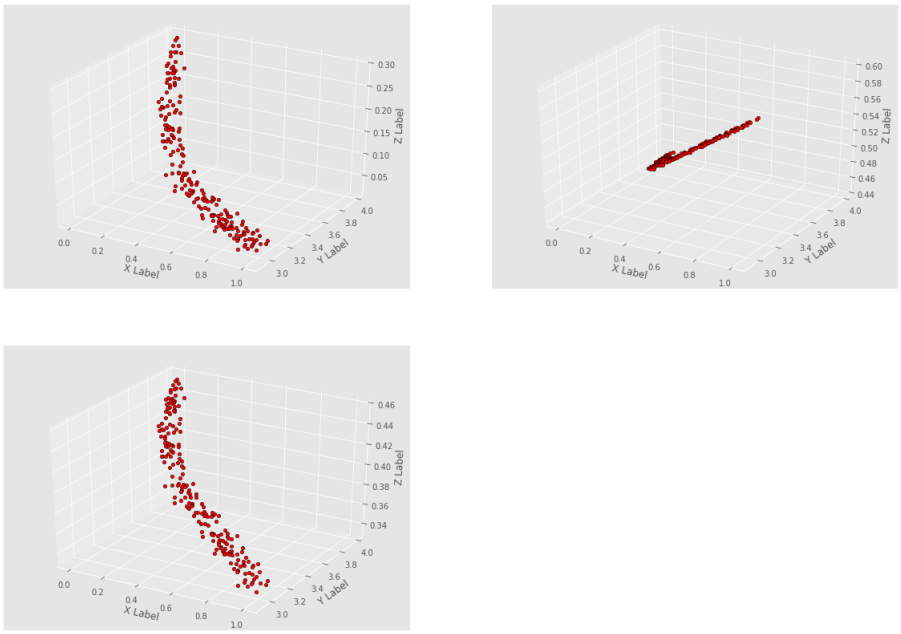
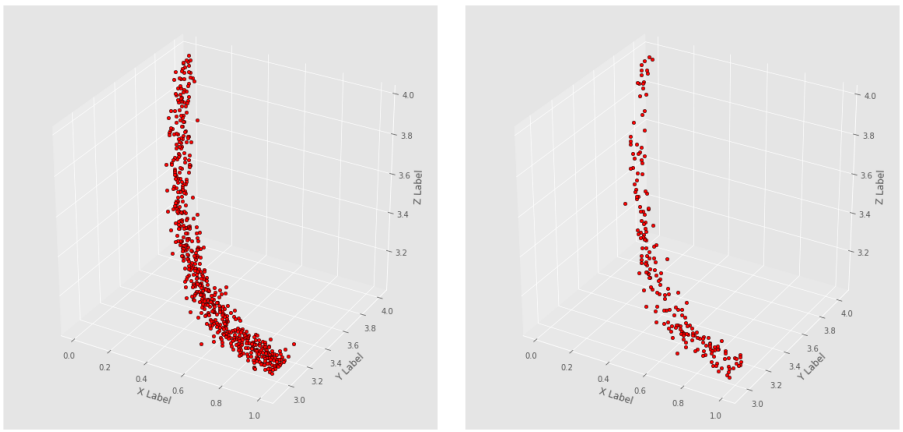
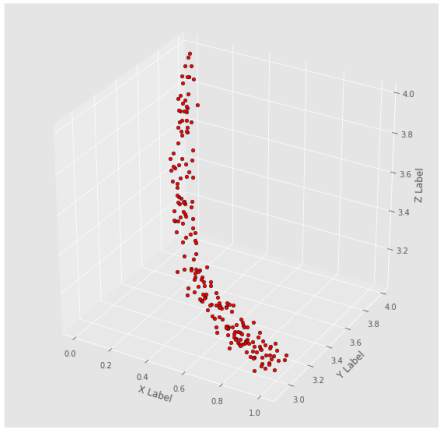


Figure 2.17: Output of the three nodes of the hidden layer for testing data.



(a) Training data.

(b) Validation data.



(c) Testing data.

Figure 2.18: Output of the single nodes of the output layer for the different data.

2.3 Bivariate Data for the Regression

In this problem, our objective is to fit a linear function on the bivariate data, for the regression problem we have provided with a 3-dimensional dataset with 10201 data points (see Figure 2.19).

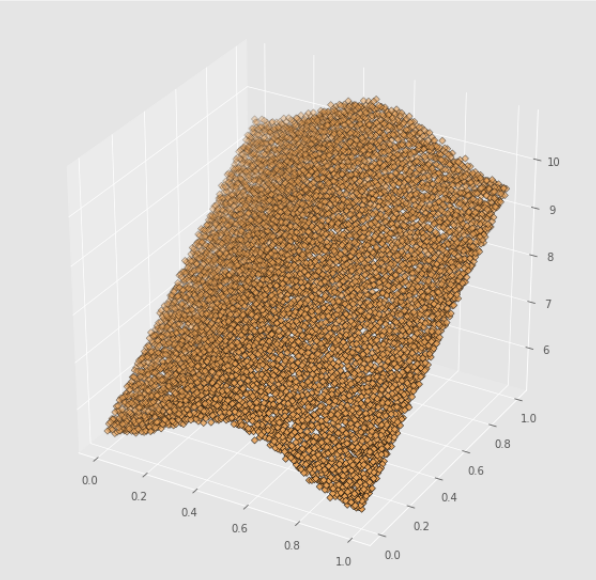


Figure 2.19: The 3-dimensional bivariate data scatter plot.

Perceptron for the Bivariate Regression

The previously developed perceptron algorithm for the regression can also be used for the bivariate regression, where the input variable is two so, the number of node in the input layer is two.

Model Calibration

After developing the perceptron algorithm, we need to tune some of the parameters to reduce the error. Table 2.7 shows the different range of parameters to be varied in the algorithm.

Table 2.7: Range of parameters in the MLFFNN.

Parameter	Range of Values
Number of Epochs	50, 100, 200, 500, 1000
Activation Functions	Identity activation function
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9
Accuracy Measure	Mean Square Error (MSE)

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 2.7 to the perceptron algorithm and record the model's error on the given parameters. We select those parameters that were reducing the validation error. The error of every epoch for the algorithm with the best set of parameters is plotted in Figure 2.20 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

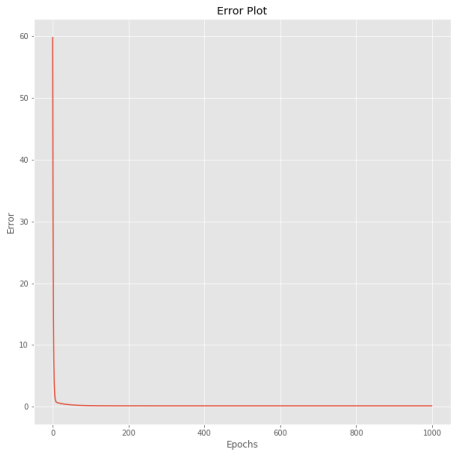


Figure 2.20: Average error of the epochs for the biivariate regression.

The results of the perceptron algorithm in the training, validation, and testing dataset are reported in Table 2.8.

Table 2.8: The error (MSE) of the perceptron algorithm for the bivariate dataset.

Algorithm	**Training**	**Validation**	**Testing**
Perceptron for univariate regression	0.129	0.128	0.129

The Table 2.9 showing the best set of parameters where the average error for the perceptron algorithm is less.

Table 2.9: Best set of parameters for the perceptron algorithm.

Parameter	**Best set of values**
Number of Epochs	1000
Activation Functions	Identity activation function
Input Shuffle	Yes
Learning Rate	0.1
Accuracy Measure	Mean Square Error (MSE)

Figure 2.21 shows the fitted regression plane on the bivariate dataset. The perceptron with the linear activation function on the output node can fit a plane to the bivariate data to minimize the error of the data.

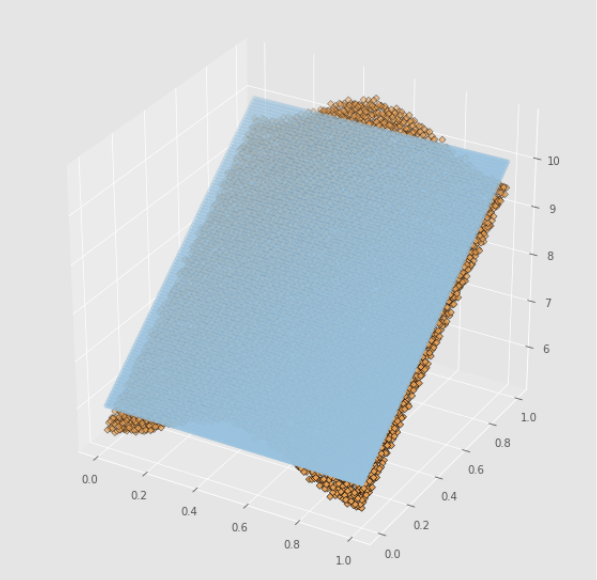
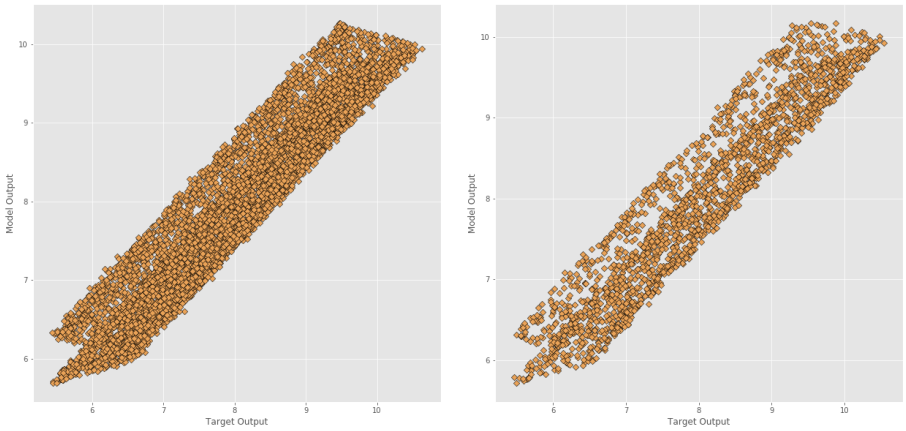


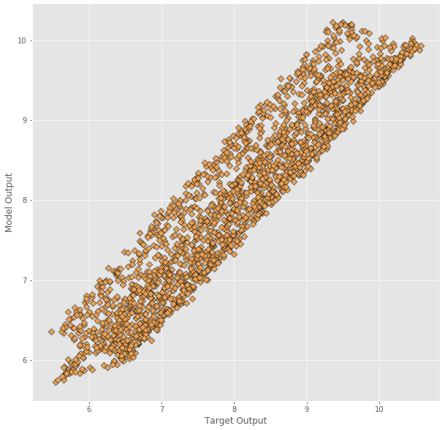
Figure 2.21: Fitted regression plane on the bivariate dataset.

Figure 2.22 shows the model output and target out scatter plot, where the x-axis is the target output, and the y-axis is the model output. If the model shows less error, then the scatter plot is the diagonal.



(a) Training data.

(b) Validation data.



(c) Testing data.

Figure 2.22: Model output and target output scatter plot for the different data.

MLFFNN for the Bivariate Regression

The previously developed MLFFNN with the backpropogation algorithm for the regression can also be used for the bivariate regression, where the input variable is two so, the number of node in the input layer is two.

Model Calibration

The newly developed MLFFNN model used to fit the regression curve on the bivariate data. In this model we passed the architecture explicitly. The model automatically creates the number of input, output nodes, and the number of hidden layers, and neurons per hidden layer. For example, architecture = [2, 10, 1] defined as the number of input nodes is 2, where 2 represents the one input variable. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the one nodes, predicting the one value.

Figure 2.23 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

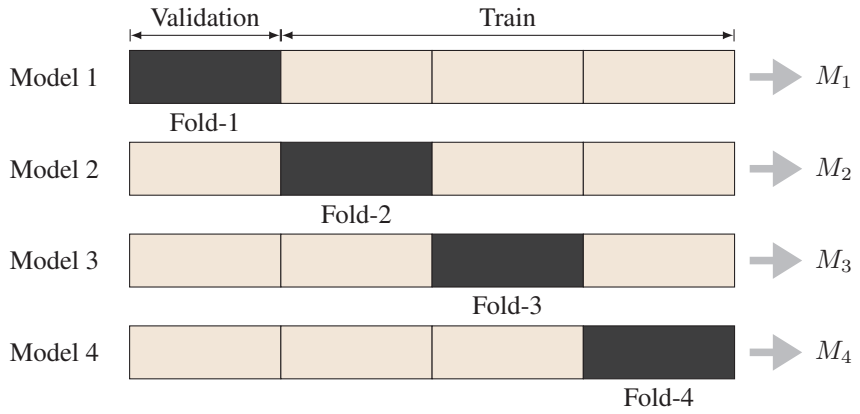


Figure 2.23: 4-cross validation of the training data

Table 2.10 presenting the different range of parameters varied in the MLFFNN.

Table 2.10: Range of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3, 4
Number of Hidden Layers	1, 2
Nodes in Input Layer	2
Nodes in Output Layer	1
Nodes in Hidden Layers	5 to 100 with step size 5
Number of Epochs	10 to 100 with step size 10
Activation Functions	logistic, hyperbolic tangent, Identity (Output layer)
Input Shuffle	Yes
Learning Rate	0.1, 0.3, 0.5, 0.7, 0.9

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 2.4 to the perceptron algorithm and record the model's error on the given parameters. We select those parameters that were reducing the validation error. The error of every epoch for the algorithm with the best set of parameters is plotted in Figure 2.24 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch. Figure 2.25 showing the MSE of the MLFFNN model with different complexity, where x-axis is the number of nodes in the hidden layer.

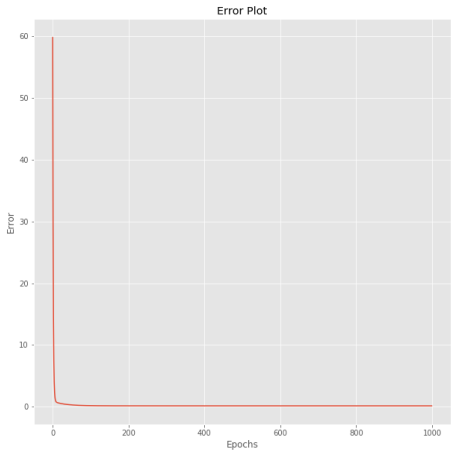


Figure 2.24: Average error of the epochs for the bivariate regression in MLFFNN.

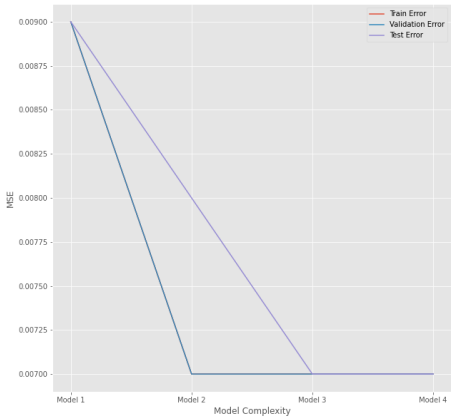


Figure 2.25: MSE of the MLFFNN model with different complexity.

The results of the MLFFNN algorithm in the training, validation, and testing dataset are reported in Table 2.11.

The Table 2.12 showing the best set of parameters where the average error for the MLFFNN algorithm is less.

Figure 2.26 shows the fitted regression plane on the bivariate dataset. The MLFFNN with the linear activation function on the output node can fit a plane to the bivariate

Table 2.11: The error (MSE) of the perceptron algorithm for the univariate dataset.

Algorithm	Training	Validation	Testing
Model 1	0.007	0.007	0.007
Model 2	0.007	0.007	0.007
Model 3	0.007	0.007	0.008
Model 4	0.009	0.009	0.009

Table 2.12: Best set of parameters for the perceptron algorithm.

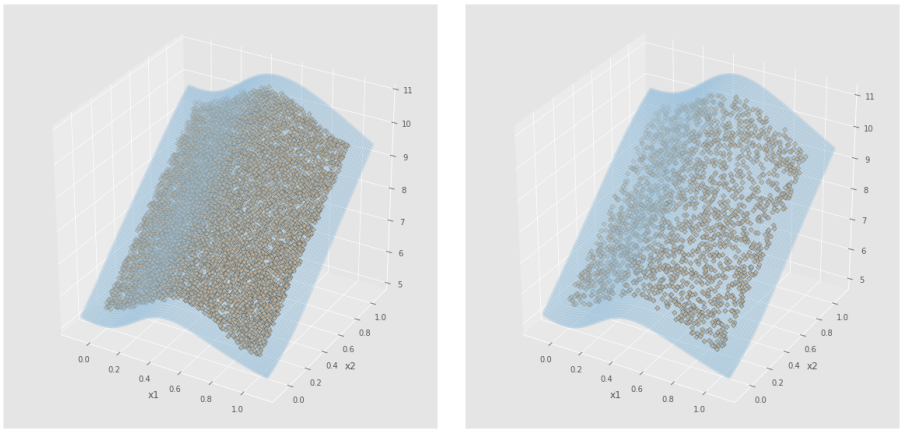
Parameter	Best set of values
Number of Layers	3
Number of Hidden Layers	1
Nodes in Input Layer	1
Nodes in Output Layer	1
Nodes in Hidden Layers	4
Number of Epochs	10
Activation Functions	logistic, Identity (Output layer)
Input Shuffle	Yes
Learning Rate	0.1

data to minimize the error of the data.. As seen in Figure 2.26, the MLFFNN model output draw with a blue hyperplane, where the x-axis and y-axis is the model input data, and the z-axis is the model output and target value.

Figure 2.27 shows the model output and target out scatter plot, where the x-axis is the target output, and the y-axis is the model output. If the model shows less error, then the scatter plot is the diagonal.

Figure 2.28, Figure 2.29, and Figure 2.30 shows the output of the hidden layer for training, validation, and testing dataset.

Figure 2.31 shows the output of the single output node for training, validation, and testing dataset.

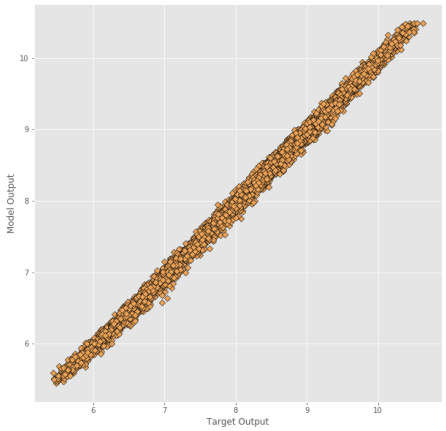


(a) Training data.

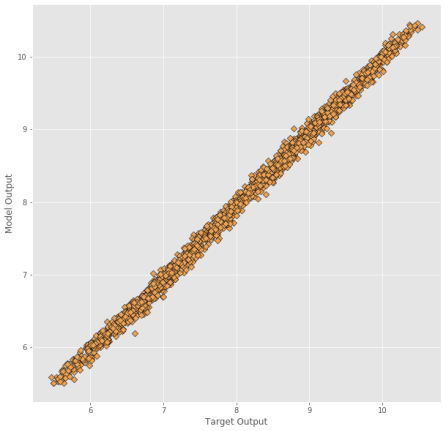
(b) Validation data.

(c) Testing data.

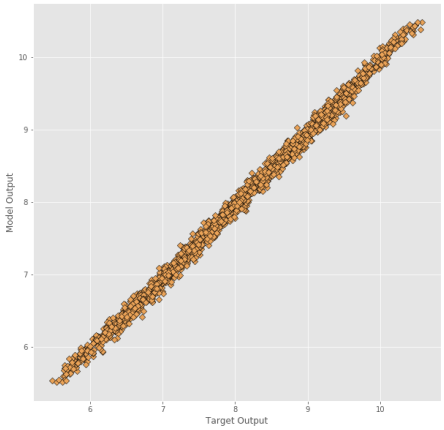
Figure 2.26: Plot of the Model output and the actual output scatter plot for the different data.



(a) Training data.



(b) Validation data.



(c) Testing data.

Figure 2.27: Model output and target output scatter plot for the different data.

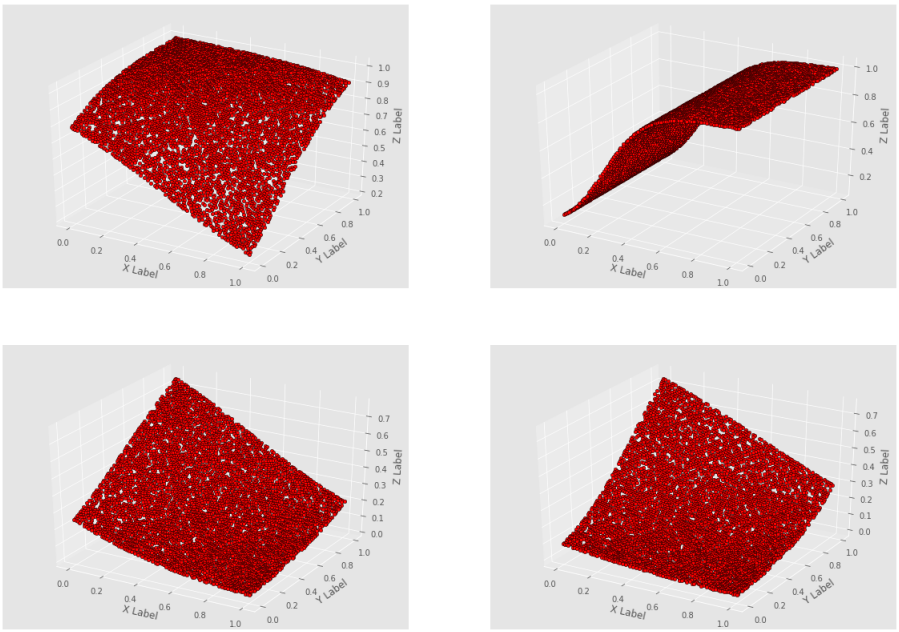


Figure 2.28: Output of the four neuron for the training data.

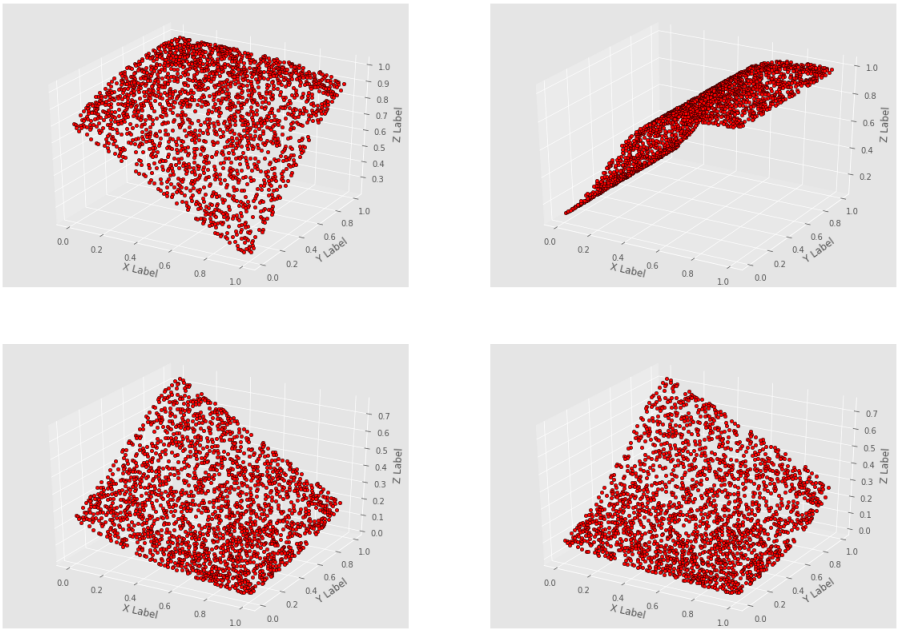


Figure 2.29: Output of the four neuron for the validation data.

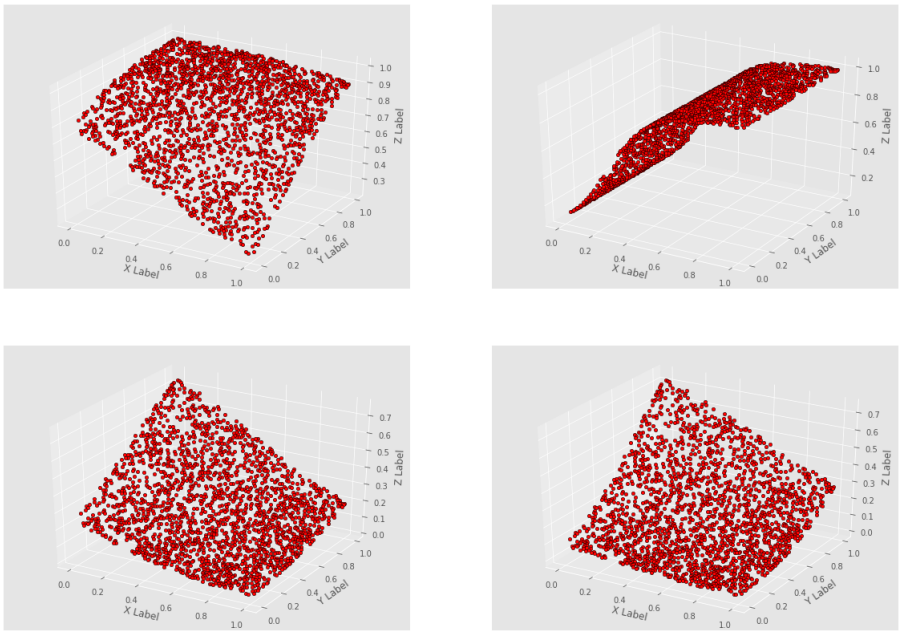
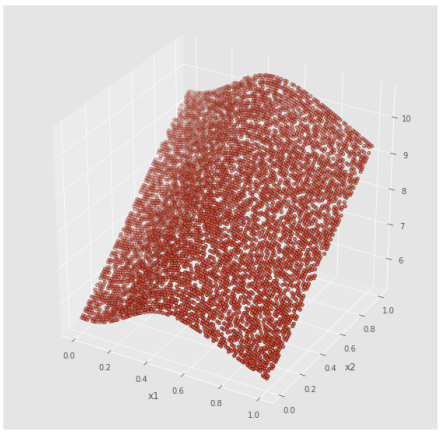
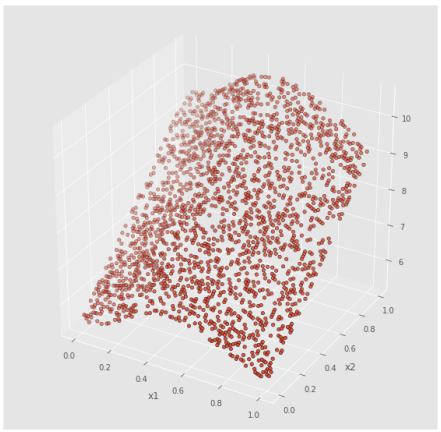


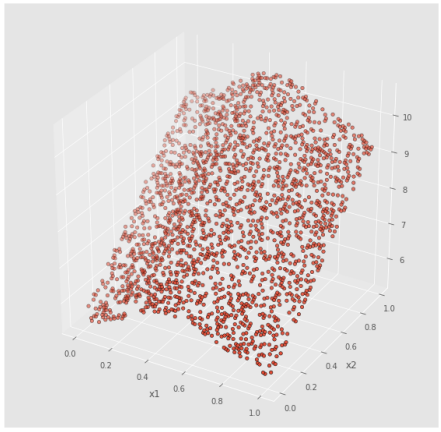
Figure 2.30: Output of the four neuron for the testing data.



(a) Training data.



(b) Validation data.



(c) Testing data.

Figure 2.31: Output of the single nodes of the output layer for the different data.

CHAPTER 3

Bag-of-visual-words (BoVW)

3.1 Classify the Image dataset with Bag-of-visual-words (BoVW)

In this problem, our objective is to separate the Scene Image Data corresponding to Three different classes with MLFFNN with sigmoidal Activation function. For this problem we had been provided 40 Images as training dataset and remaining 10 images as validation dataset. The feature of the each images should be extracted with the Bag-of-visual-words (BoVW). Figure 3.1a, Figure 3.1b, and Figure 3.1c showing the sample images from the each classes.

Figure 3.1 showing the sample images from the each classes.

(a) Class synagogue outdoor.

(b) Class desert vegetation.

(c) Class auditorium.

Figure 3.1: The sample image from the three different classes.

Bag-of-visual-words (BoVW

The bag-of-words model can be utilized for image classification. In the BoVW model, every image feature is treated as words. The features could be extracted from the patches of the image by a vector. In this vector count, the number of pixel frequency in a patch. This feature vector is called a feature descriptor. The final step for the BoVW model is to convert vector-represented patches to codewords. A codeword can be considered as a representative of several similar patches via the k-means clustering algorithm

Methodology

Data Preprocessing

In this experiment, we used the 50 image data provided for training and validation on the data-set. we had classify the image data-set into three classes. class 1 as synagogue outdoor class 2 as desert vegetation and class 3 as auditorium Every image had three-color channel RGB (red, green, blue).

Image Patches

In every image, we extracted the nonoverlapping patches of 32×32 size from the training and test dataset. All the images in the training and testing dataset are different, so we have padded some edges pixel at the end of the image to make the patches size 32×32.

Figure 3.2 shows the one patch with size 32×32 of the image.

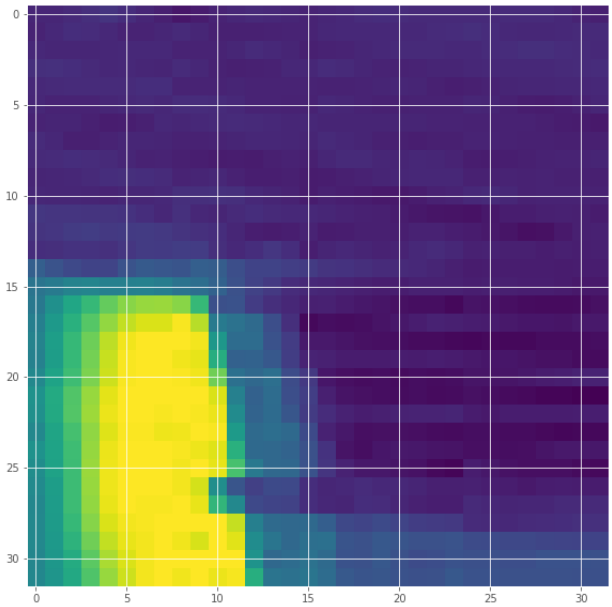


Figure 3.2: A 32×32 size patch of the image.

Color Histogram

The ever patch in the image has three color channels red, green, and blue. The pixels in the patches are from 0 to 255 range, so we have divided this range into eight equal bins. Count the number of pixels falling into each bin. This results in a vector with eight elements. From the three color channels of the patch, we have the three vectors. Finally, we concatenated these three vectors and created a single vector with twenty-four elements.

Figure 3.3 shows the color histogram of the one patch of the image.

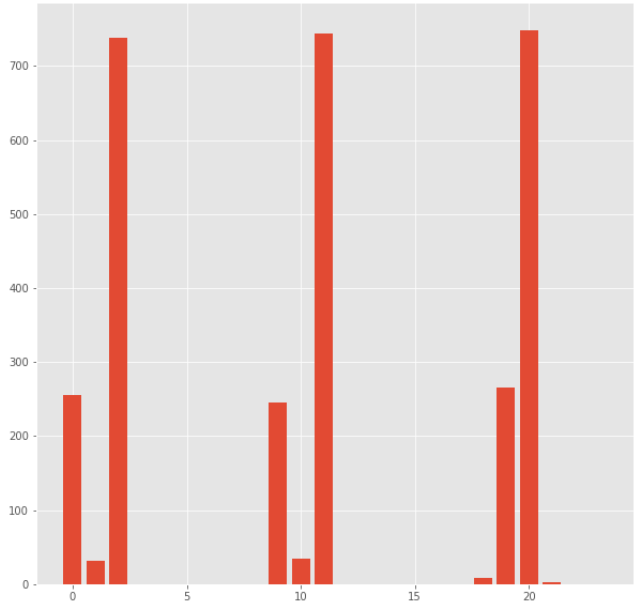


Figure 3.3: A color histogram of the single patch.

K-means Clustering

Some of the patches in the images represent the same features in the same image or the other images in the classes. These features belong to the same category so that we can group them into the same category. The k-means clustering developed for the group the same feature. The 32 clusters were created for grouping the same features. The k-means algorithm applied to the training data and returned the 32 means value of the 32 clusters. The dimension of the vector was 32×24.

Codebook

The final step of the BoVW mode was to generate the codebook of the images. The color histogram of every image assign to the 32 clusters, and count the number of the color histogram of this image belongs to a particular cluster. Finally, we come up with a 32-dimensional vector for every image. This vector is referred to as a codebook of that image. The codebook vector was generated for all the images in the training and testing images.

The number of patches in the images was different due to the different sizes of the images. To normalize the data, we divided the codebook vector with the number of patches in that image.

Figure 3.4 shows the histogram of the codebook of the image.

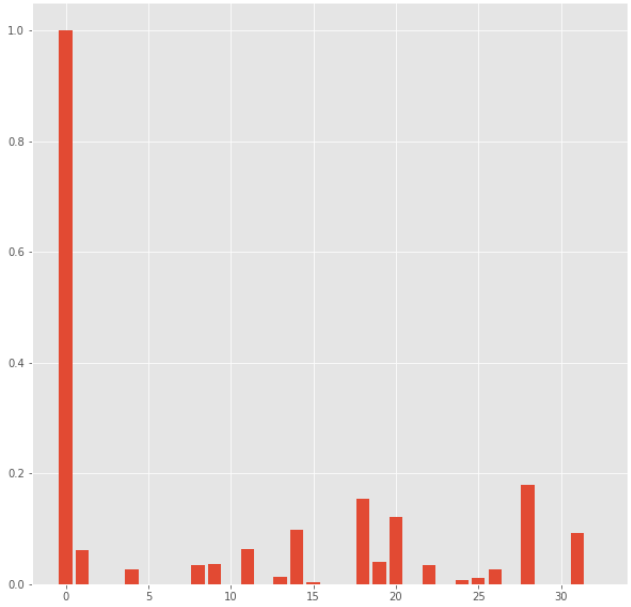


Figure 3.4: A histogram of the codebook.

Cross-validation

Figure 3.5 showing the input for the four different MLFFNN models, where the 20% validation chosen by the cross fold validation methods.

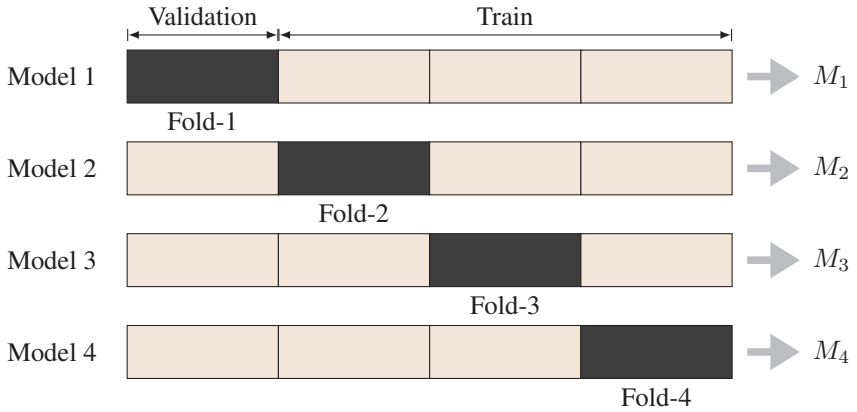


Figure 3.5: 4-cross validation of the training data

Multi-layer feed-forward Neural Network

We have developed MLFFNN with the backpropagation algorithm utilized for the classification of the images. In this model, we pass the codebook of the images as input and predict the classes of the images. We varied the hyperparameters for the MLFFNN model, which we are discussing in the next section.

Model Calibration

The newly developed MLFFNN model used to classify the image data. In this model we passed the architecture explicitly. The model automatically creates the number of input, output nodes, and the number of hidden layers, and neurons per hidden layer. For example, architecture = [32, 10, 3] defined as the number of input nodes is 32. The number of the hidden layer is one with ten neurons. The final output layer is the output layer with the three nodes, predicting the three classes' output. Similarly, architecture = [32, 10, 20, 3] defined as the number of hidden layers are two with the 10 and 20 neurons, respectively.

Table 3.1: Range of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3, 4
Number of Hidden Layers	1, 2
Nodes in Input Layer	32
Nodes in Output Layer	3
Nodes in Hidden Layers	10, 20, 30
Number of Epochs	100 to 1000 with step size 100
Activation Functions	logistic, hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1, 0.2, 0.3, 0.4, 0.5

Results

We developed the grid search algorithm and supply all combinations of the parameters discussed in Table 3.1 to the MLFFNN model and record the model's accuracy on the given parameters. We select those parameters that were maximize the validation accuracy. The error of every epoch for the MLFFNN with the best set of parameters is plotted in Figure 3.6 where the x-axis is the number of epochs, and the y-axis is the average error in the epoch.

The results of the MLFFNN in the training, validation, and testing dataset are reported in Table 3.2. From Table 3.2, we can see that model 3 has the best validation accuracy.

The Table 3.3 showing the best set of parameters where the average error for the MLFFNN algorithm is less.

Figure 3.7a, Figure 3.7b, and Figure 3.7c, show the confusion matrix of the training, validation, and test data of all the three classes from the dataset, respectively. The diagonal elements in the matrix represent the number of correctly classified classes.

Table 3.4 show the classification accuracy for the training, validation, and testing dataset.

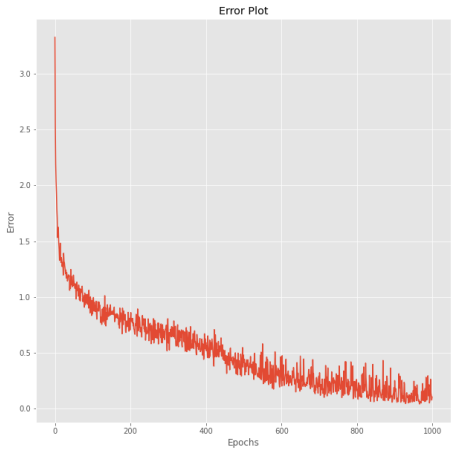


Figure 3.6: Average error of the epochs for the image classification by BoVW method.

Table 3.2: The accuracy of the MLFFNN algorithm on the linearly separable data.

Algorithm	**Training**	**Validation**	**Testing**
Model 1	100%	67%	31%
Model 2	100%	70%	27%
Model 3	100%	75%	30%
Model 4	100%	67%	32%

Table 3.3: Best set of parameters in the MLFFNN.

Parameter	**Range of Values**
Number of Layers	3
Number of Hidden Layers	1
Nodes in Input Layer	32
Nodes in Output Layer	3
Nodes in Hidden Layers	10
Number of Epochs	1000
Activation Functions	hyperbolic tangent
Input Shuffle	Yes
Learning Rate	0.1

Table 3.4: The classification accuracy for the different dataset.

Accuracy Measure	**Training**	**Validation**	**Testing**
Average Accuracy (%)	100	75	30
Average Recall (%)	100	75	30
Average Precision (%)	100	77	33
Average F-score (%)	100	76	31

Actual Class \ Predicted Class	1	2	3
1	36	0	0
2	0	36	0
3	0	0	36

(a) Training data.

Actual Class \ Predicted Class	1	2	3
1	9	0	3
2	0	9	3
3	2	1	9

(b) Validation data.

Actual Class \ Predicted Class	1	2	3
1	4	7	39
2	3	26	21
3	30	5	15

(c) Testing data.

Figure 3.7: Confusion matrix for the different dataset.