Plymouth University

Back-Propagation and biologically plausible random feedback alignment algorithm

by

Kshitij Limbu

Acknowledgments

I would like to thank my supervisor Dr. Ian Howard for his expertise and guidance in completing this project.

Abstract

Back-propagation is an algorithm that is classed biologically implausible. The feedback alignment algorithm proposed by (Lillycrap, 2016) yet simple, is shown to be as effective as back-propogation and considered to be biologically plausible. The thesis looks to implement a neural network that is able to classify some non-linear patterns using back-propogation algorithm. Then, the feedback alignment algorithm is implemented and similar test conducted on (Lillycrap, 2016) is recreated for comparison. Since (Lillycrap, 2016) uses uniform distribution to generate the random weights, for novelty, the thesis applies normal distribution to generate the random weights instead.

Contents

List of Figures

CHAPTER 1

Introduction

Artificial neural network, a subset in the domain of machine learning is based on how learning takes place in the brain. Its application has grown over the years, in technologies such as image compression, stock market prediction, computer vision, speech recognition, machine translation, board and video games. This thesis evaluates simple neural networks. The classic back-propagation algorithm will be implemented to classifiy various datasets generated from different distribution in MATLAB. The back-propagation algorithm is deem biological implausibility due to its characteristics. A biological plausible algorithm, random feedback, the generalization will be compared to back-propagation.

The thesis is divided into several chapter. Chapter 2 looks into simple single layer networks and implemented in MATLAB with some tests. Chapter 3 discusses multi-layer networks and implemented in MATLAB with some tests. Chapter 4 looks into the biologically plausible algorithms, ’random feedback’ and implemented in MATLAB with some tests. Chapter 5 discusses the results and concludes the thesis.

CHAPTER 2

Single Layer Neural Network

2.1 Biological brain

An intuition on how neurons in the brain works seems appropriate in order to identify and clarify why artificial neural networks is said to be inspired and based on the neurons in the brain. Yet the brain is still said to be an unknown phemonon as neurobiologist are still trying to understand how the brain functions, [1] abstracts the complexities and complications to understanding the brain in terms and concepts that can be connected to artificial neural networks, which is discussed below.

The brain recieves information, perceives it and make decisions, as a wave of forward ad backward transmission. Though the brain processes information at a rather slow rate compared to computer, the billions of neurons and trillions of synapses work in parallel allowing for more efficiemcy. A biological neuron can be visualised as portrayed in Figure 2.1. The neuron consists of a soma, axon and dendrites, where neurons transmit signals through connections known as synapses between dendrite and axom. There are several process involved for transmission. The pre-synaptic process involves difussing transmitter substance across the synaptic junction. The synapse converts a presynaptic electrical signal into chemical signal and then back to a postsynaptic electrical signal. Then, the synapse can impose either excitation or inhibitation, but not both. Action potentials are electrical pulses that propogate across neurons at a constant velocity and amplitude. Action potential needs to reach a threshold at which the neuron fires an output signal. Excitation pushes the action potential towards the threshold, while inhibitation pushes the action potential away from the threshold.

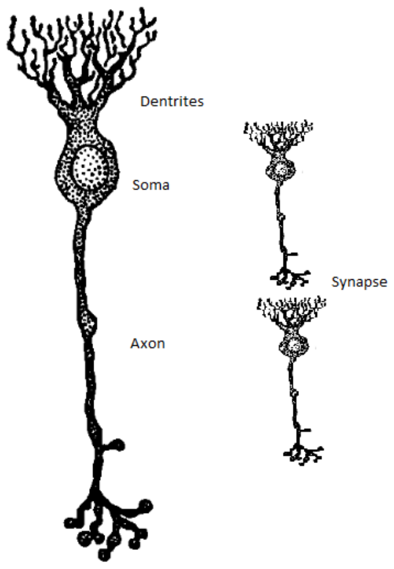


FIGURE 2.1 Biological Neuron.

2.2 Network Architectures

Neural networks have similar architecture to the brain. There are several different classses of network architectures. The thesis will look into feedforward networks of: single and multi layered variations. The generic structure of neural network are in the form of layers. The input layer, the hidden layer/s and the output layer, which consists of neurons/nodes connected through synapses (which are called weights, with some appropriate value assigned to it). The forward pass propogates information from the input layers through the network and outputs some value. The main distinction between single-layer and multilayer network, is the number of hidden layers that exist within the network. Single layer network consist of no hidden layers, for which both networks approach with a different learning methods. Due to additional connections that exist in multilayer network as a result of the additional layers, the network able to overcome the limitations of single layer networks. Information (from the inputs) are propogated forward in the network in a layer by layer basis. A fully connected architecture (where a neuron in connected to every other neuron in the adjacent forward layer) is usually adapted, as it provides a general purpose connection pattern as opposed to sparse connectivity.

2.3 Learning

Intuitively looking at the idea of learning for humans, the same approach is adopted in neural network where learning takes places from its environment and overtimes the performance can be improved. [1] defines learning in the following sequence where the neural network is stimulated by an environment, undergoes changes in its free parameters as a result of this stimulation and responds in a new way to the environment because of the changes that have occured in its internal structure.

In the field of machine learning, there exists a plethora of problems where several categories of learning systems exist. For the purpose of the thesis, supervised learning is the adapted and appropriate approach for the tasks at hand. Supervised learning also refered to as 'learning with a teacher', entails the used of preprocessed input and its desired value of datasets, which is fed to the neural network. Error-correction learning is then used, to make subsequent adjustments to all the weights. The learning involves the output value/s obtained from propagating the input through the network against the desired value for that particular input. The purpose of this learning is to push the output value of the subsequent set of inputs to move closer to the desired value.

2.4 Activation Functions

There are several activation function that can be used. The functions used for this thesis are the threshold function (step-wise) and the sigmoid function. The anti-symmetric version of the threshold function gives the following output depending on the following condition:

$$\varphi(v) = \left\{ \begin{array}{c} \text{1, if } \ v \geq 0 \\ \text{-1, if } \ v < 0 \end{array} \right\}$$

where, a value of 1 is given to v greater or equal to 0, while a value of 0 is given otherwise.

The sigmoid function is defined by:

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

where a determines the steepness of the slope. The output is a range of continuous values from 0 to 1.

2.5 McCulloch Pitts Neuron and simple neuron

The pioneers of aritifical neuron networks, Warren McCulloch and Walter Pitts in 1943 created the first mathematical model of a biological neuron. This model is discussed in most text-book on artificial neural network, and one such is [6]. The neuron Figure 2.2 is able to perform simple Boolean logic operations. The composition of E as excitory inputs, and I as inhibitory inputs would result in 1 (active) or 0 (inactive). An active inhibitory input would result in an output of 0, where as to get an output of 1, all inhibitory inputs need to be inactive while the sum of the excitory inputs need to reach a threshold.

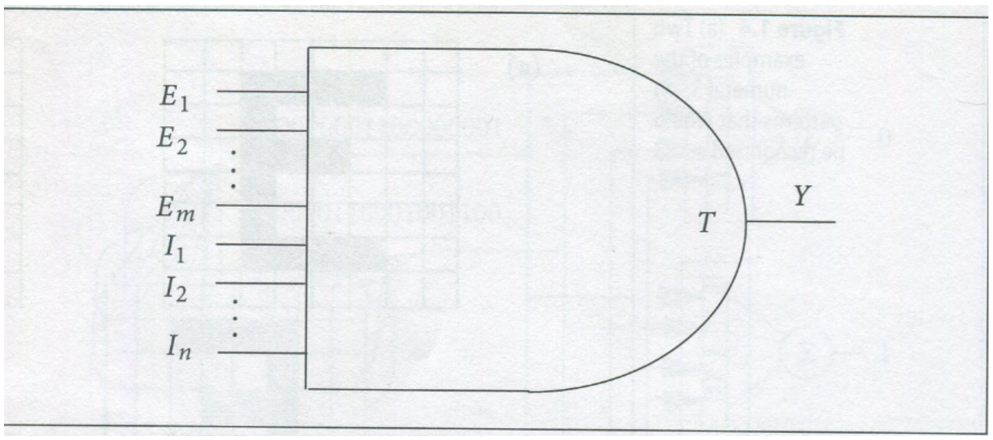


FIGURE 2.2 First Neuron

Several modification made to the original as it had been noted that the neurons were unable to generalize, there have been a number of models with variations. With this new form Figure 2.3, the model permitted a set of inputs with corresponding weight values with a bias value. The linear combiner summates the product of input values and weights to transform via some activation function at the output, which then give some value.

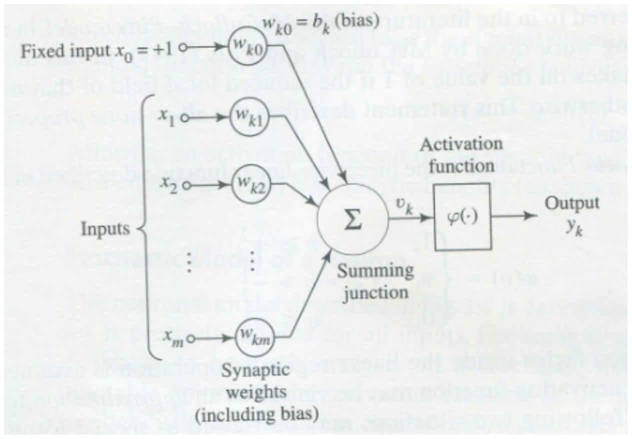


FIGURE 2.3 New Model

2.6 Perceptron

The perceptron was proposed by Frank Rossenblatt in 1958 as a model built around non-linear neuron using supervised learning. Non-linear, as the output value is applied with a threshold function (hard-limiter). Perceptron is a simple form of neural network used for the classification of patterns that are linearly seperable. Linearly seperability is defined, as there is a hyperplane such that all points of the classes are seperated by it.

Perceptron convergence theorem [2] states that for any data set which is linearly seperable, the learning rule is guranteed to find a solution in a finite number of steps.

Incorporating the functionality of the simple neuron mentioned above using the threshold function, the output is either 1 or 0. The weight is initialised to a value other than 0. Inputs values set to 0 or 1. With the perceptron, this output is compared against the provided target. If the target and output does not match, the difference as the error is used to update the weights, given as

$$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n)$$

2.7 Adaline

The ADALINE (adaptive linear neuron) was proposed by Bernard Widrow and Ted Hoff in 1960. The architecture of the network is very similar to that of the perceptron. Training is done via supervised learning. The inputs are propagated through the network, but for error correction, instead of the hard limiter, the linear value is used. Weight are adjusted according to Hebb's rule which states [6]:

”When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.”

Thus, the value of the weights are increased if the output is active when the input associated with that weight is also active (1). Usesd +1 and -1 instead of 0 and 1.

The delta rule or Least Mean square (least mean square) is used to adjust the weights, minimizing the mean square error with ith outputs.

$$e = \sum \frac{1}{2}(d_i - y_i)^2$$

From [2], the derivative with respect to the weight is given by,

$$\frac{\partial e}{\partial w_{ij}} = -(d_i - y_i)x_j \tag{2.1}$$

The update rule is given by,

$$w(n+1) = w(n) + \eta x(n)e(n)$$

2.8 Implementation in MATLAB

The following pseudocode highlights the main functionality of $simple\_network\_train$ function implemented in MATLAB which trains weight parameters on linearly seperable patterns using either perceptron or adaline. Full code is available in the appendix which can be accessed from here.

```
INPUT (inputs, targets, network, learning rate)
```

```
inputs: matrix of inputs values
targets: matrix of targets for the inputs
network: have to choose between perceptron or adaline for learning
learning rate: a value set for the rate of learning

weights initialised by random selection from a zero mean, unit variance

append 1 to the inputs as the bia value

REPEAT
    for n = 1 to number of inputs
        switch
            case 'perceptron'
                computes the linear combiner

                uses the hard limiter to get an output value
                of either 1 or 0;

            case 'adaline'
                computes the linear combiner

                if linear_combiner > 0 output is 1
                else output is 0
    end
target is evaluated to make comparison

if output and target are not a match, correction to the weight is needed,
hence the weights are updated
    switch
        case 'perceptron'
        Error Correction done here using the perceptron rule to the weights

        case 'adaline'
        Error Correction done here using the adaline rule to the weights


        if weights are updated error value is appended with a value so that its greater
        than 0 so the while loop continues;
UNTIL error value is less than 0
```

```
OUTPUTS weights;
```

Once an $optmial\_weight$ is obtained, the $simple\_network\_test$ function uses this weight to generalise. Full code is available in the appendix which can be accessed from here. The pseudocode below provides details on the general functionality.

```
INPUT (weight, testingData, testingTarget)
%testingData is just the input values
%testingTarget is the desired value for the corresponding inputs
append ones to the testingData as the bias value

for n = 1 to ndata to loops through each test data
    compute the linear combined value

    output computed depending on perceptron or adaline

    the output value is compared to the corresponding targets
    if the values doesnt match a value of 1 is added to the
    error

OUTPUT error
```

The $GenerateGaussianDataLS$ function is a matlab file provided by Dr. Ian Howard which generates a two-class dataset in matrix data with 2D data vectors from a Gaussian distribution also labeling each class. The number of sample is the only variable that needs to be entered in the function, where a dataset of input values and its target is generated. The generated data is also linearly seperable. Full code is available in the appendix which can be accessed from here.

To test, $simple\_network\_train$ and $simple\_network\_test$, the script "$test\_simple\_network$" was used. Full code is available in the appendix which can be accessed from here. Below is the pseudocode.

```
generate a linearly seperable dataset with training Data,
training Target, testing Data and testing Target

run the simple_network_train function with the following inputs : (training Data as
input, training Target as targets, network set to either perceptron or adaline,
and some value for learning_rate) to obtain some optimised weights

use the optimised weights in simple_network_test
along with (optimised weight as weights, testing Data,testing Target) to check
```

```
if correct classification are made

create random points within the boundary of the input training and testing data, for
visualising the decision boundary

use simple_network_test with the optimised_weight and the random points.
The simple_network_test has a hidden feature that keeps tracks of all the
classification for the input values in a matrix.

Plot four graphs
One with the training data
One with the testing data
One with the classification made using the simple_network_train and simple_network_test
One with the decision boundary
```

A sample of 10000 training dataset and 1000 testing dataset generated. Learning rate set to 0.2. The script was used to evalute both cases of Perceptron and Adaline. Figure 2.4 and Figure 2.5 shows plots for the perceptron and the adaline. The plots throughout the thesis are provided in the same format. The colours indicate different classes. Training Data is the data used for training. Desired result is the plot of the testing data. Predicted Result is the classification given to the set of input from the testing data. Decision boundary shows how the networks classified random points within the boundary of the testing data using the optimised weights. Since the distribution is linearly seperable, both the perceptron and adaline classifies perfectly.

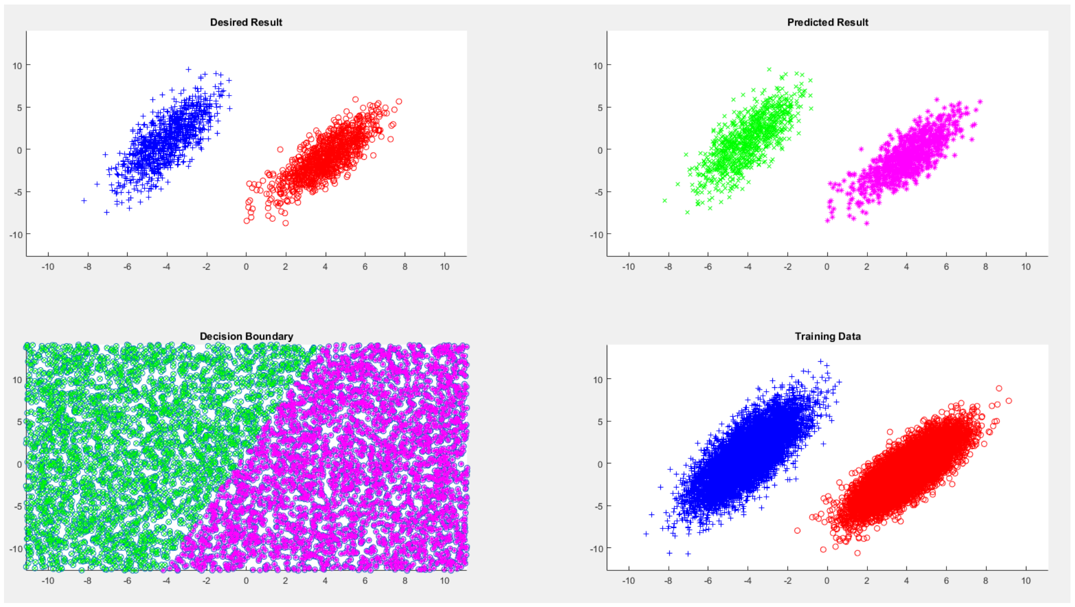


FIGURE 2.4 Perceptron

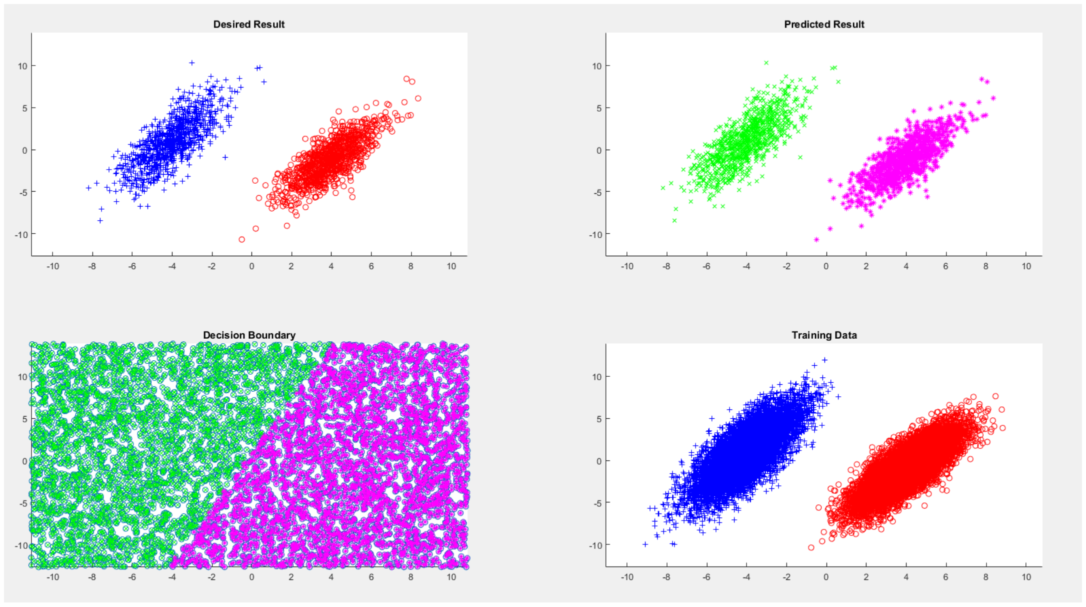


FIGURE 2.5 Adaline

2.9 Limitation of the perceptron

The perceptron convergence theorem only applies to single layered perceptrons on linearly separable datasets. If a hyperplane that can seperate different classes does not exist for a dataset, then the perceptron as well as the adaline is unable to solve. Non-linear function such as the X-OR Figure 2.6(a representation) is one such case where these single layer network are unable to generalize, since the network will never converge.

To overcome these limitations, several multi layer network has been developed post advent of the perceptron. One such network is the multi-layer perceptron which is discussed in the next chapter.

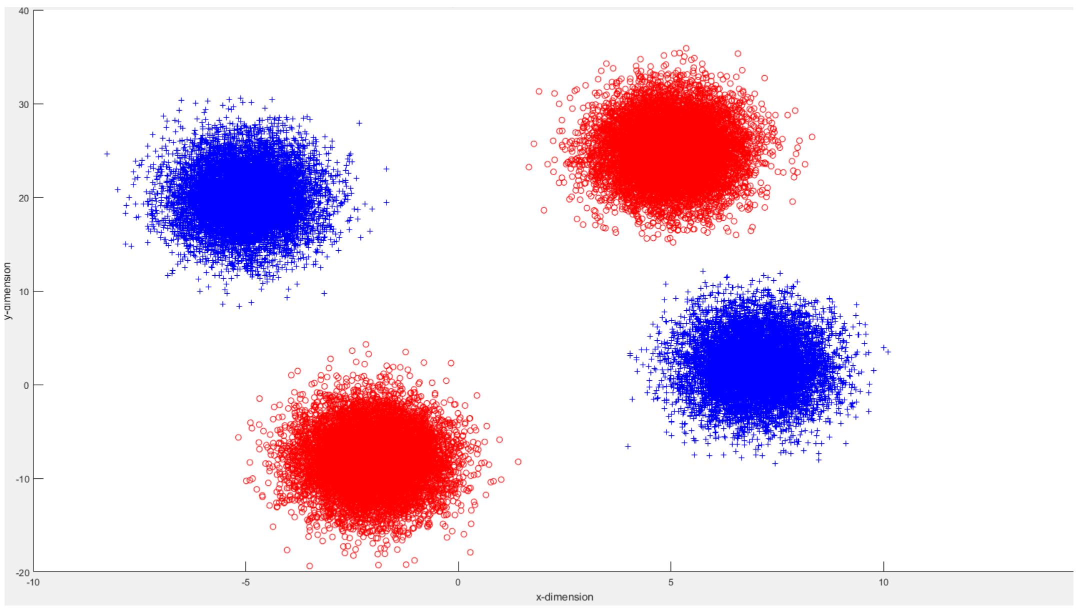


FIGURE 2.6 XOR

CHAPTER 3

Multilayer Neural Network

3.1 Multilayer perceptron

The multilayer perceptron is the enhancement made to the single layer perceptron to overcome its limitations. Unlike the single layer perceptron, multi-layer perceptron consists of one or more hidden layers between the input and output layers of the network and uses the back-propagation algorithm for learning.

The back-propagation consists of two phases: the forward pass, where inputs are propagation through the network, then the backward pass, where the weights are adjusted depending on the error.

A requirement for back-propagation is a differentiable activation function. Non-linear activation functions are generally chosen for multi-layer networks. There are several non-linear activation function available, but the sigmoid function is chosen as the activation for this thesis. The sigmoid function is given by,

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

where the derivative is given by

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

Hence,

$$f'(x) = f(x)(1 - f(x)).$$

It can be noted that the activation functions is continuous and the derivative is simple to compute as well.

Figure 3.1 illustrates a network of a simple multilayer perceptron, with two inputs, x_1 and x_2 with its random corresponding weights connecting to neurons in the adjacent layer (to the right progressing through the network). The bias in the input layer is given by b_1 and b_2, which also has some random corresponding weights. The linear combinor, $\sum w_1 x_1$, is the sum of the product of the inputs and also the bias value with its corresponding weight given by

$$hid(n) = \sum_{i=0}^{m} w_i x_i \tag{3.1}$$

for a neuron in the hidden layer. $i = 0$ is the bias. The activation function is applied to this value which is given by

$$h\_act(n) = \varphi(hid(n)) \tag{3.2}$$

and traversed forward (with some corresponding random weight) in a similar manner to the input value until the output layer. Given by

$$out(n) = \sum_{j=0}^{m} h\_act_j(n) w_j \tag{3.3}$$

$$out\_act_j(n) = \varphi(out(n)) \tag{3.4}$$

where the sigmoid function is used for this case, which gives an output value. This is a summary of the forward pass.

In essence, a multilayer can have finitely many inputs and hidden layers as required. The type of activation function applied in each layer can also be varied.

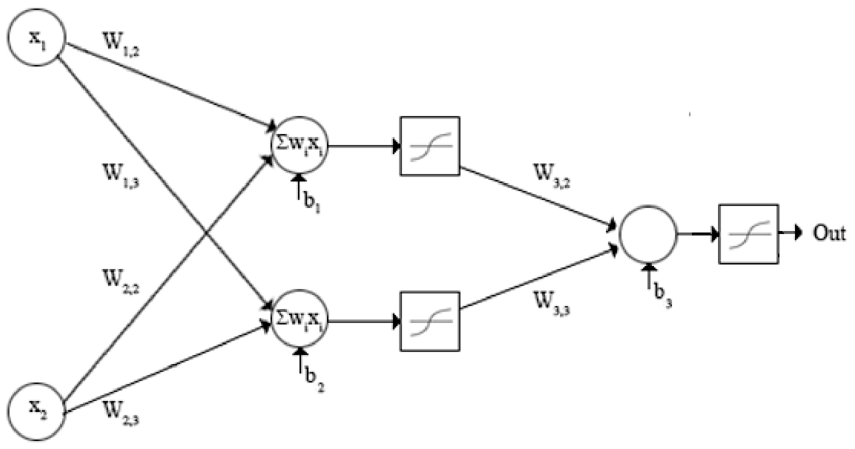


FIGURE 3.1 Multilayer perceptron.

3.2 Back-propagation algortihm

The value obtained at the end of the forward pass (3.4), and the desired value is used to evaluate the error using the sum of square function, given by

$$e = \frac{1}{2}(d-y)^2 \tag{3.5}$$

Considering the case where there is only no hidden layer; isolating the hidden layer and the output layer in Figure 3.1.

E denotes the total error at the output of the network. The above network in the figure repesents a single output network. But, it is also possible to have multiple output network, in which case, e for each neuron in the output layer is summated together for the value of E. Similar to the delta rule, the back-propagation algorithm corrects the weights given by the partial derivative [3]

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial v}\frac{\partial v}{\partial w} \tag{3.6}$$

The partial derivative tells us which direction to move in weight space to reduce the error. Chain rule is a process that allows computation of derivatives of composite functions. A composite function is a function composed of multiple inner or nested functions. The left side of (3.5) is decomposed into composite of four functions to give the right side.

$\frac{\partial E}{\partial w}$ is the derivative of the error with respect to w, the weight in the hidden layer.

$\frac{\partial E}{\partial e}$ represents the derivative of the total error with respect to (3.5), which is the error evaluated at the output neuron.

In a case where multiple output neurons are present in the output layer, the derivate of

$\frac{\partial e}{\partial y}$ can be evaluated using (3.5) and taking the partial derivative with respect to y. In $\frac{\partial y}{\partial v}$, y is the equivalent of (3.2) as the transfomed value, and v is the equivalent of (3.1), a linear combiner. $\frac{\partial v}{\partial w}$ is the derivative of v with respect to the weight, equivalent of taking the derivative of (3.1).

[3] evaluates (3.6) to the equivalent of

$$\frac{\partial E}{\partial w} = -e\varphi'(v)y \tag{3.7}$$

where $\varphi'(v)$ is the derivative of v which has been transformed using the activation function. y is input. Depending on the position of the weight in the network, this value can either be the input fed to the network (if w exist in the first layer) or (3.2)(if w is in the hidden layer). A local gradient [3] δ is defined as the value for $e\varphi'(v)$ in (3.6). So, δ is the equivalent of

$$\frac{\partial E}{\partial v}$$

Observing the right side of (3.6), as

$$\frac{\partial E}{\partial v} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial v} \tag{3.8}$$

it can be observed

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial v}\frac{\partial v}{\partial w} \tag{3.9}$$

For the weights in case a hidden layer is present between the output layer, error from the output of the hidden layer is unknown since a desired output for this layer is not specified. The output error needs to propagates backwards so that an error value can be assigned to the weight in the hidden layer as a share of contribution towards the output value in the network. $\frac{\partial E}{\partial w_h}$ is to be evaluated. The local gradient [3] is derived using the definition of δ which gives

$$\delta_h = \varphi'(v_h)\sum_k e\varphi'(v)w_k \tag{3.10}$$

$\varphi(v_h)$ is the neuron (for this case, this neuron would be in the hidden layer) connected from w_h where the liner combined value is transformed using the activation function. $\varphi'(v_h)$ is just the derivative of $\varphi(v_h)$ with respect to v_h. $e\varphi'(v)$ is δ and w_k is the weight connecting the hidden layer to the output layer. The local gradient combined with a learning parameter and input gives the delta rule (Haykin, 1999),

$$weight\_correction = learning\_rate * local\_gradient * input$$

Depending on the depth of layers from the output of the network, the error would have to be determined recursively in terms of the error signal of all the neurons to which that hidden neuron is directly connected [3].

Now, consider looking to compute the derivative of a weight that has several layers and a single output in output layer.

$$\frac{\partial E}{\partial w} = (-e\varphi'(x)z)\beta(\varphi'(v_{adj})y_{adj}) \tag{3.11}$$

where

$$\beta = \prod_{i=1}^{l}\sum_{n=1}^{m}\varphi'(v_{adj+1})y_{adj+1}$$

e is the error. φv_{adj} and y_{adj} are the adjacent input and corresponding transformed value through the activation function. x is the linear combined value at the output node. z is the weight adjacent to the output node. l is the number of layers between the weight and the output layer. m is the number of nodes in the layer. Essentially, all associated nodes and weights are combined in determining the value for updating the weight.

Once the gradient has been obtained, the respective weights are updated with a suitable learning rate parameter. This parameter is set within the range of 0 to 1, where a value closer to 0 will make no changes to the weights, and 1 makes use of the exact value of the gradient. It is a ratio of the gradient, which determines the rate of change to updating weights.

3.3 Training

Training can be conducted in several ways. An epoch denotes a complete run through the entire dataset. Batch training uses the mean of the error from an epoch, and then only makes updates to the weights. On-line training on the other hand, updates weights on each training set from the dataset. Mini-batch training takes random small batches to train on each epoch and uses the mean error.

A stopping criteria is also necessary as the network doesn not converge. A reasonable number of epochs can be set at which the program stops running. The error can also be used to stop the network when it reaches a certain threshold.

The experiments conducted for the thesis is looking into classification problem. Generally the sum of squares function is used. [5] discusses that for classification problems, it is better to optimise the network outputs to represent the posterior probabilities (the probability of event A occurring given that event B has occurred) of each class. Since the target is binary, a Gaussian noise model (which is used to yield the sum of squares function) does not provide a good description of its distribution. Using Bernoulli random variable (the probability distribution of a random variable which takes the value 1 with probability p and the value 0 with probability q=1-p), the cross-entropy error function is derived.

For a two class problem, the cross-entropy error function is

$$e = -t^n ln y^n + (1-t^n) ln(1-y^n)$$

For a multi-class problem, the cross-entropy error function is

$$e = -t_k^n ln y_k^n$$

3.4 Implementation of 3 layer network in MATLAB

The $three\_layer\_network\_train$ function implemented in MATLAB uses the back-propagation algorithm to train a set of weights. The three layers are the input layer, one hidden layer and output layer. Full code is available in the appendix which can be accessed from here. Below is a pseudocode version.

```
INPUT: epochs, input, targets, no_of_output, no_hidden_neuron, output_function,
error_function, algorithm, learning,momentum, learning_rate, error_val,
mini_batch, mb_value

%plot_error: acumulates all the error from each epoch
%epochs: number of iterations
```

```
%output_function: sigmoid or softmax depending on the number of outputs
%error_function: cross_entropy for single or multiple output
%algorithm: back-prop or random feedback

Initialise weights for the two layers

add bias value of 1 to the input

batch = 1;

while batch < epochs as the stopping criteria
    %choose between mini-batch or batch, 1 for mini-batch, 0 for batch or
    %online
    switch mini_batch
        case 0
            for n=1:ndata;
                forward propagation of input value to the hidden neurons

                application of transfer function

                add bias value of 1 to this value to propagate forward

                switch output_function

                    case 'sigmoid'
                        propagate forward

                        use the sigmoid function as the output function
                        ( y = 1./(1 + exp(-forward))) to get an output

                    case 'softmax'
                        propagate forward

                        apply the output function, for softmax

                end

                switch out_funct
                    case 'single'
                        switch algorithm
                case ’back-prop’
                use back-prop to evaluate the  delta value to update the
                weights connecting the hidden layer to output layer

                use back-prop to evaluate the delta delta value to update
                the weights connecting the evaluate the input layer to
                the hidden layer

                 switch learning
                        case ’batch’
                          for batch learning acumulating the errors

                        case ’online’
                            for on-line learning update the weights

        case ’multi’

            switch algorithm

                case ’back-prop’
                back propagation is done using the same method
                (redundant information)


    end

    switch error_function
           evaluate cross entropy error function for two class

           or, evaluate cross entropy error function for multi class depending
           on the output of the network




case 1
    For mini-batch, mb_value is the number of inputs retrieved randomly
    from the inputs

    for m=1:mb_value;
```

```
                Backpropagation is done in the same way from here forth



        if weights are to be updated using batch learning

        weights in the second layer is updated

        weights in the first layer is then updated


    %batch error
    if batch was selected, error function is calculated here

    for n=1:ndata

        the forward pass of back-prop is evaluated to give an output value

        switch error_function

                    cross entropy error function for 2 class is evaluated and the
                    error is accumulated

                    or, cross entropy error function for multi class is evaluated and
                    the error is accumulated

    evaluated cross entropy error are stored in a matrix to create plots later on

    if eval_error < error_val is the stopping criteria
        break

OUTPUT:(both sets of weights; updated weights between the input and the hidden layer,
and the weights between the hidden layer and the output layer, evaluated cross entropy
error over the batch)
```

The optimised weights can be used to classify some testing data using the *three_layer_network_test* function. Full code is available in the appendix which can be accessed from here.

Below is a pseudocode version of the main aspect of the code.

```
INPUT: (both sets of optimised weights, testingData, testingTarget)

incorrect_prediction = 0;

    for n= 1 to no of inputs in testingData

        if two class problem

            forward propagation using optimised weights

            sigmoid function applied at the output function

            if this value  < 0.5
                output = 0;
            else
                output = 1;

        if multi class problem

            forward propagation

            softmax function applied at the output function

            the index of the maximum is used as the output
            as with multi class problems, there are multiple
            values at the output


        if target ~= output
            append 1 to number of incorrect_prediction

    error = incorrect_pred;
```

$GenerateNonLinear$ is a version of $GenerateGaussianDataLS$ using different parameter to create a slightly complex non-linear function. Full code is available in the appendix which can be accessed from here.

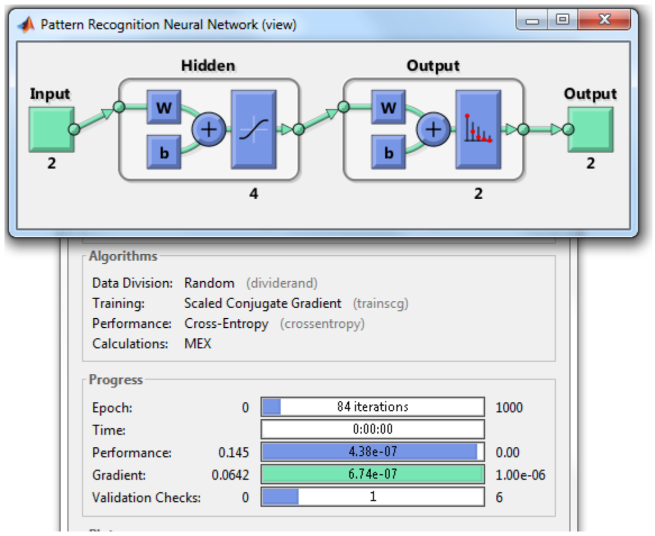
As there are several training modes available; batch training, online-training or mini-batch, it only seems appropriate to decide the best training mode before progressing any further.

For mini-batch, choosing 200 random samples.

Since MATLAB has a built in neural network toolkit, for purpose of making comparison to ensure that the implementation of back-propagation is not absolutely incorrect and to get some initial configuration for the network, the MATLAB neural network toolkit is utilised. The network set with 1 hidden layer of 4 nodes with the sample of 30000 training data and 6000 testing data generated from $GenerateNonLinear$.

In Figure 3.2, it can be observed that the MATLAB neural network, took 84 iteration trained using Scaled Conjugate gradient and the performance measured using cross-entropy error function which gave an error of 0.004.

To test the back-propagation implementation, similar parameter will be set. Learning rate is set to 0.2, 4 nodes used in the hidden layer, error set to 0.005, maximum number of epochs set to 80. The same sample was used for training and testing in the three different learning modes.



Algorithms

Data Division:	Random (dividerand)
Training:	Scaled Conjugate Gradient (trainscg)
Performance:	Cross-Entropy (crossentropy)
Calculations:	MEX

Progress

Epoch:	0	84 iterations	1000
Time:		0:00:00	
Performance:	0.145	4.38e-07	0.00
Gradient:	0.0642	6.74e-07	1.00e-06
Validation Checks:	0	1	6

FIGURE 3.2 Matlab neural network

Batch learning failed and was unable to generalise. Moving on, Figure 3.2 is the result of using on-line learning. Mini-batch online learning was also classify with similar results to on-line learning. Figure 3.4 show the error over 35 iterations during training. $Script1$ (Full code is available in the appendix which can be accessed from here) is used for this task. It is a procedure that makes use of $Three\_layer\_network\_train$ and $Three\_layer\_network\_test$, and makes plots in similar way to $simple\_network\_test$. For further training, a combination of mini-batch on-line and on-line learning will be used. $Three\_layer\_network\_train$ function is able to classify well so far, but when a slighlty complex non-linear function is presented, such as Figure 3.5. It is unable to generalise.

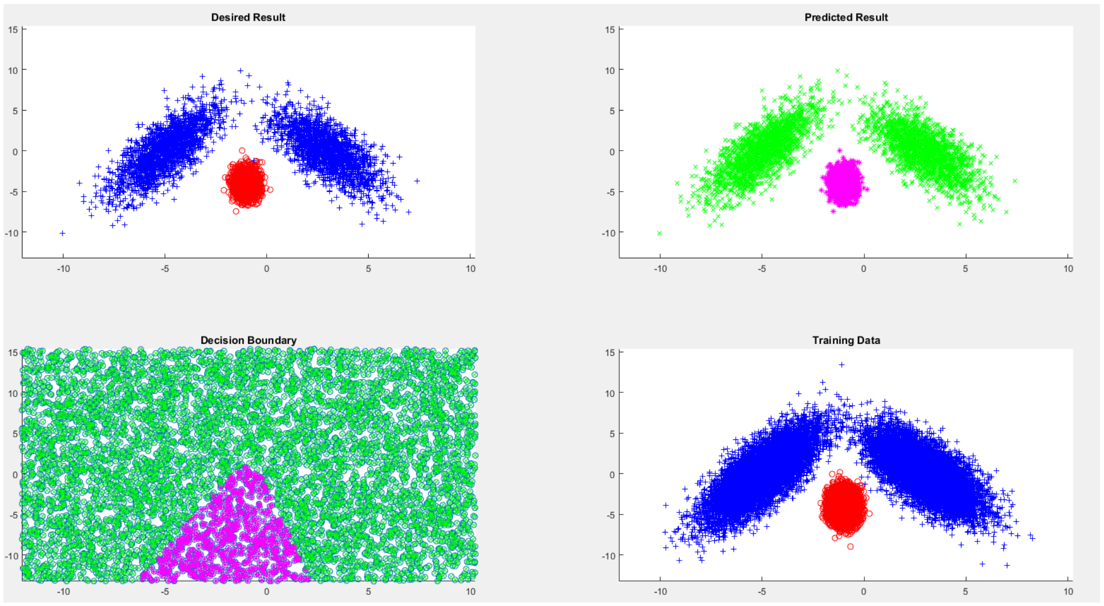


FIGURE 3.3 On-line Learing on non-linear function generated from GenerateNonLinear

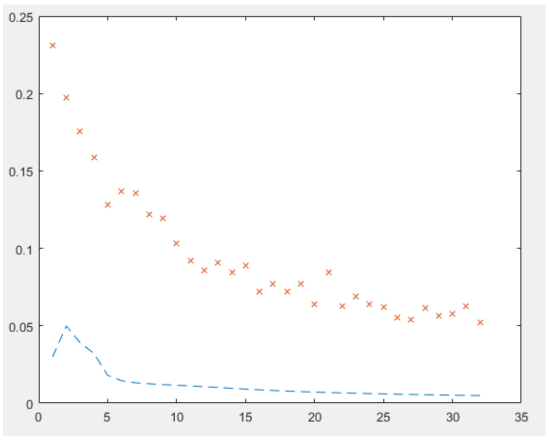


FIGURE 3.4 (- - -) represents on-line learning, (x x x) represents mini-batch on-line learning

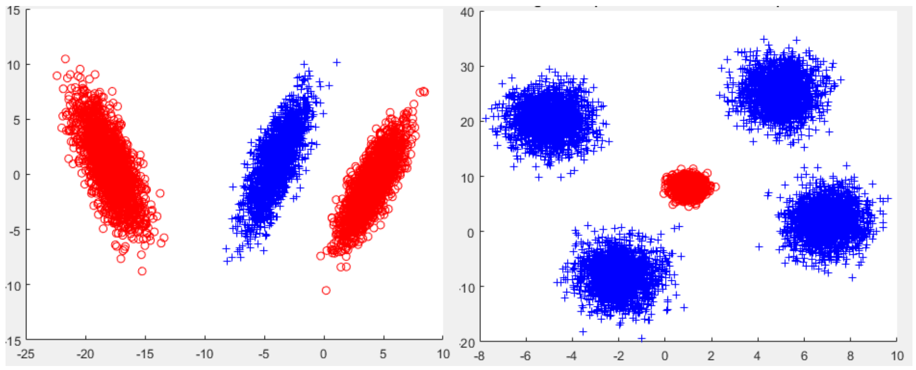


FIGURE 3.5 (left) generated from GenerateNonLinear3. (right) generated from GenerateNonLinear2.

3.5 Implementation of multiple hidden layers in MATLAB

The $multi\_layer\_network\_train$ function is able to train a network with multiple hidden layers with multiple nodes. Full code is available in the appendix which can be accessed from here. Below is a psuedocode of the cruical part of the code

```
INPUT: (epochs,input, targets, no_of_output, net_architecture, output_function,
error_function, algorithm, learning,momentum, learning_rate, error_val, mb_value)
% epochs: max number of iterations
% input: the input dataset
% targets: corresponding target values for the inputs
% no_of_output: total number of nodes in the output layer
% net_architecture: the hidden architecture is presented with this value in a column array
% output_function: depending on the number of output, either sigmoid or softmax is applied
% error_function: either sigmoid or softmax, works out the relevant error

no_of_hidden_layers is the number of columns of the net_architecture

no_of_weight_matrix between layers is the no_of_layers + 1;

for n= 1:no_of_weight_matrix

        randomises and generates weight

        stores the weight in a structural array
end

bias_appended_input = [ones(ndata, 1), input];

iter = 1;
while iter < epochs

    %mini batch
    for m = 1:mb_value;

        n=random(m);

        for layer = 1:no_of_layers

            forward propagation through each layer with the corresponding weights
            retrieved from the structural array storing all weights while storing
            value of propagated value in each layer in a structural array

        end

        switch output_function

            compute sigmoid or softmax depending on the number of classes at the output

        end

        switch out_funct
            case 'single'

                switch algorithm

                    case 'back-prop'

                        depending on the number of hidden layers, the delta values for
                        all the weights between output layer and the adjacent hidden
                        layer to the left is evaluated. Also required to retrieve
                        the corresponding values from the structural array of weights
                        between each layer and forward propagated values at each layer

                        Then the delta values for all the weights across the hidden layer
                        to the left is evaluated.

                        This process is continued until the weights right of the input
                        layer is evaluated.

                        %%% The remaining steps is similar to the
                        Three_layer_network_Train
```

The testing function $multiple\_layer\_network\_test$ (Full code is available in the appendix which can be accessed from here) is very similar in the way it functions to $three\_layer\_network\_test$.

Max epoch set to 80, 2 hidden layer with 20 nodes on each layer, minimum cross entropy error set to 0.005 and using mini batch of 200, training and testing was done on a variety of non-linear distributions. The configuration set in $Script3$ (Full code is available in the appendix which can be accessed from here) for full code) for a sample of 3000 training and 600 testing dataset.

Mini-batch on-line learning is used as it provides a random sequence every epoch which allows the network gets exposed to a varied sequence of data. From the results below, it can be observed that the $multiple\_layer\_network\_test$ function is able to generalize non-linear distribution of varied complexity, including the XOR (Full code is available in the appendix which can be accessed from here.).

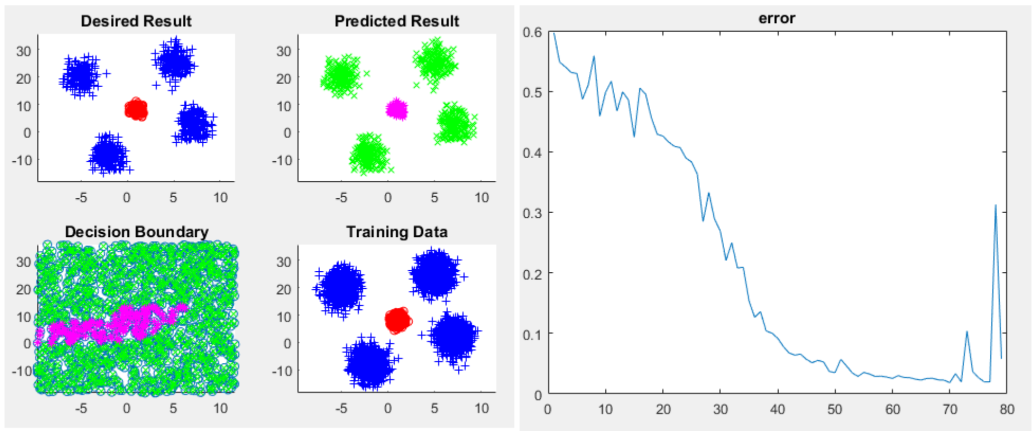


FIGURE 3.6 Classification of Distribution generated using GenerateNonLinear2

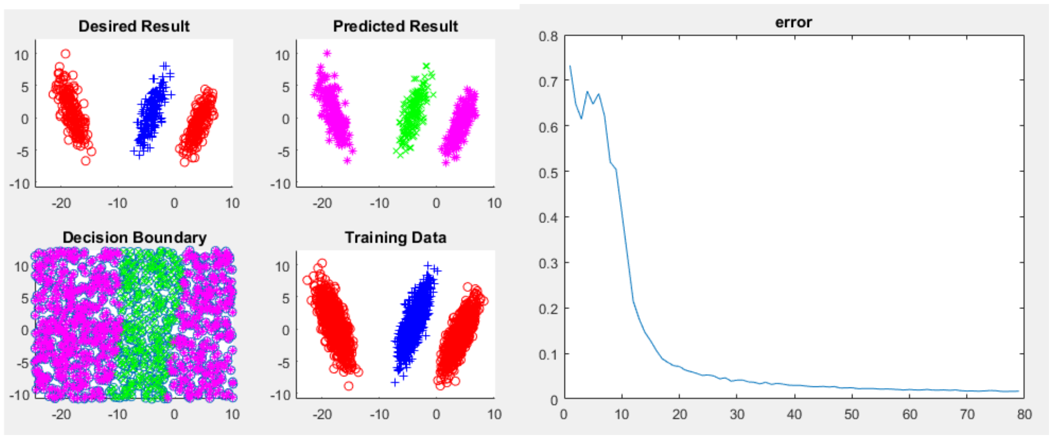


FIGURE 3.7 Classification of Distribution generated using GenerateNonLinear3

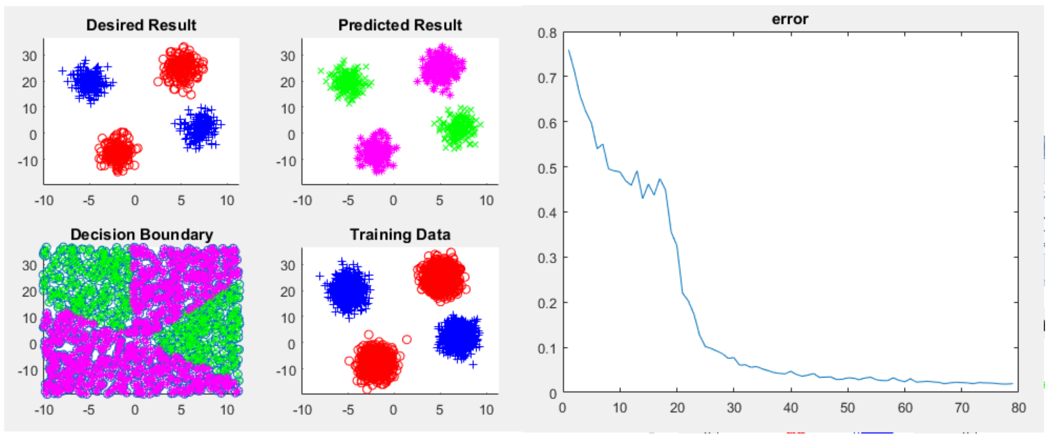


FIGURE 3.8 Classification of Distribution generated using XOR

3.6 Limitation of back-propagation

The figure below is a plot of the the learning rate set to different values, but using the same parameter and the same dataset. It shows that the gradient descent algorithm is slow and requires small learning rates for stable learning

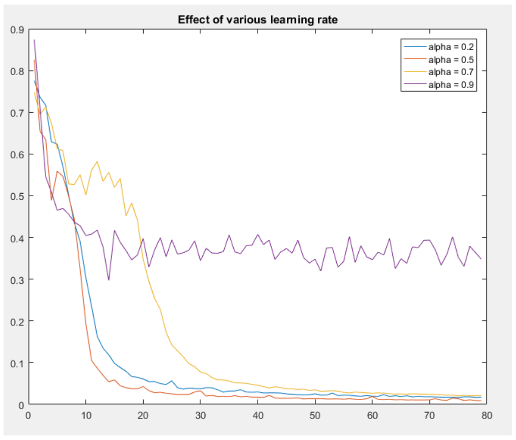
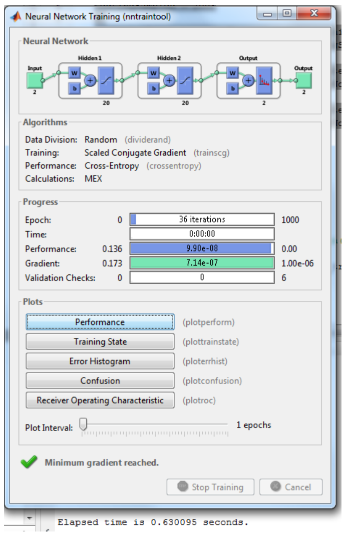


FIGURE 3.9 Various learning rates used on same sample

When dealing with non convex functions, there is that risk of getting stuck at a local minima. A momentum rate resolves this problem. A rule can be implemented for updating weights is,

$$\delta w = \eta w(n-1) + \alpha error$$

where η is the momentum rate and α is the learning rate. But this is even slow.



Neural Network Training (nntraintool)

Neural Network

Hidden 1
Hidden 2
Output
Input
Output
W
b
2
20
20
2
2

Algorithms

```
Data Division:  Random  (dividerand)
Training:       Scaled Conjugate Gradient  (trainscg)
Performance:    Cross-Entropy  (crossentropy)
Calculations:   MEX
```

Progress

```
Epoch:                 0   36 iterations   1000
Time:                      0:00:00
Performance:       0.136   9.90e-08        0.00
Gradient:          0.173   7.14e-07        1.00e-06
Validation Checks:     0   0               6
```

Plots

Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Confusion (plotconfusion)
Receiver Operating Characteristic (plotroc)
Plot Interval: 1 epochs
Stop Training
Cancel

✓ Minimum gradient reached.

```
Elapsed time is 0.630095 seconds.
```

FIGURE 3.10 Various learning rates used on same sample

MATLAB neural network is able to train a network in 0.63 seconds and using scaled conjugate gradient. The back-propagation on the other hand, took 236.470638 seconds. It seems obvious that the basic gradient descent requires optimizing.

CHAPTER 4

Random Feedback alignment

4.1 Introduction

One of the major criticism to the back-propagation alogrithm is that the algorithm is biologically implausible. [7] discuses consideration to back-propagation for biological implementation. The error must be calculated explicitly and separately from the fundamental neurons (the neuron where the linear combiner). During training, the error signal must be backpropagated along physical axons separate from the feed-forward ones. The learning mechanism for the bottom-up(afferent) pathways, must be the same (or nearly the same) as that for the top-down (efferent) pathways as required by the symmetry relation in backpropagation.

The weight transport problem refers to the weights that is used to propagate the error from the output layer backwards. The weights are said to be transported antidromically in back-propagation. [4] proposes the random feedback algorithm which provides a solution to the weight transport problem and shows that it is possible to do backpropagation with random weights and still obtain significant results on various benchmarks. The algorithm computes in similar manner to back-propagation, where random weights are used instead of the weights. Recall the local gradient in back-propagation given by

$$\delta_h = \varphi'(v_h) \sum_k \delta_k w_k \tag{4.1}$$

In random feedback algorithm, w_k is replaced with random weights of the same size.

The algorithm is based on the intuition that any matrix B will suffice,so long as on average, $e^T W B e > 0$, where e is the error in the network's output [4]. Geometrically, this means that $e^T W$ and Be, lies within 90° of each other.

$$\theta = cos^{-1}(||a^T b||/(||a||.||b||)) \tag{4.2}$$

where a and b represent two vectors. The two matrices $e^T W$ and Be were 'flatten' into vectors to compute the angle.

As a consequence, B pushes the network in roughly the same direction as W in back-propagation. [4] randomises the weight uniformly distributed over the range of $[0.5, 0.5]$.

4.2 Implementing Feedback alignment in MATLAB

Since random feedback alignment algorithm is almost identical to back-propagation apart from the random weights, simply adding some random weights and a case for random feedback in both $three\_layer\_network\_train$ and $multi\_layer\_network\_train$ should suffice.

In the $three\_layer\_network\_train$, a case for feedback alignment is also added. Below is the pseudocode explaining the addition.

```
create random weights with uniform distribution in the range [□0.5,0.5] with
the same size as the weights

case 'feedback_alignment'

        the weights in the layer between the hidden layer and the output layer is
        exactly the same as how it was done for back-prop

        %the weights connecting the input layer and the hidden layer
        forward propagation is the same as for back-prop

        during back-propagation , instead of the transpose weight in the network,
        the random weight matrix is used with is the same size as the transpose weight
```

In $multiple\_layer\_network\_train$ function a case for random feedback alignment is also added. Below is the pseudocode explaining the addition.

```
for n= 1:no_of_weight_matrix

        randomises and generates weight from a uniform distribution over -0.5 and 0.5

        stores the weight in a structural array strictly used only for back-propagation
end

case 'feedback_alignment'
        depending on the number of hidden layers, the delta values for all the weights between
        Then the delta values for all the weights across the hidden layer to the left is evalua

        This process is continued until the weights right of the input layer is evaluated.
```

$Script4$ (Full code is available in the appendix which can be accessed from here) sets the exact configuration as $Script1$ except the random feedback alignment algorithm case is used for training. Using the $three\_layer\_network\_train$ function which consists of 1 hidden layer consisting of 4 nodes, error set at 0.005, maximum epochs set to 80 and learning rate set to 0.2. The results are as follows

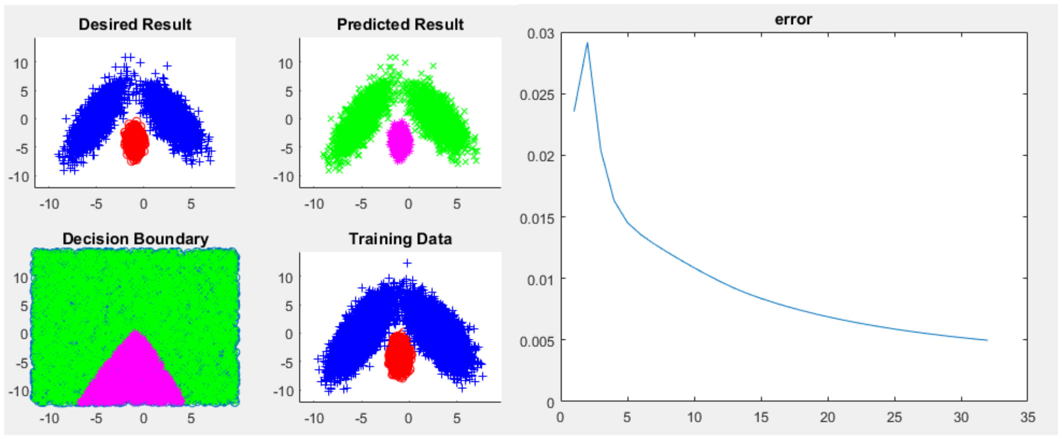


FIGURE 4.1 Feedback alignment generalization on distribution created by GenerateNonLinear

$Script5$ (Full code is available in the appendix which can be accessed from here) sets the exact same parameter as $Script3$ but uses the feedback alignment algorithm for $multiple\_layer\_network\_train$ which consistz of 2 hidden layer of 20 nodes each, max epoch of 80, learning rate set to 0.2, min error set to 0.005, mini-batch of 200 inputs.

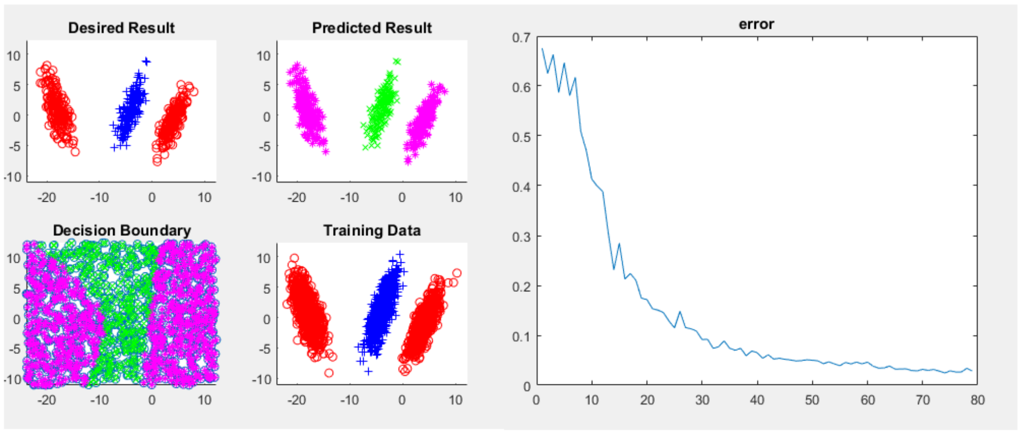


FIGURE 4.2 Feedback alignment generalization on distribution created by GenerateNonLinear3

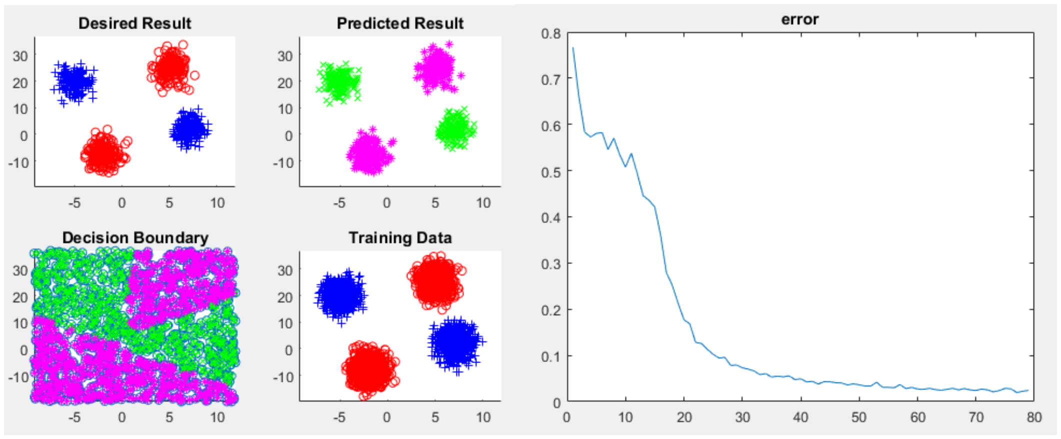


FIGURE 4.3 Feedback alignment generalization on distribution created by XOR

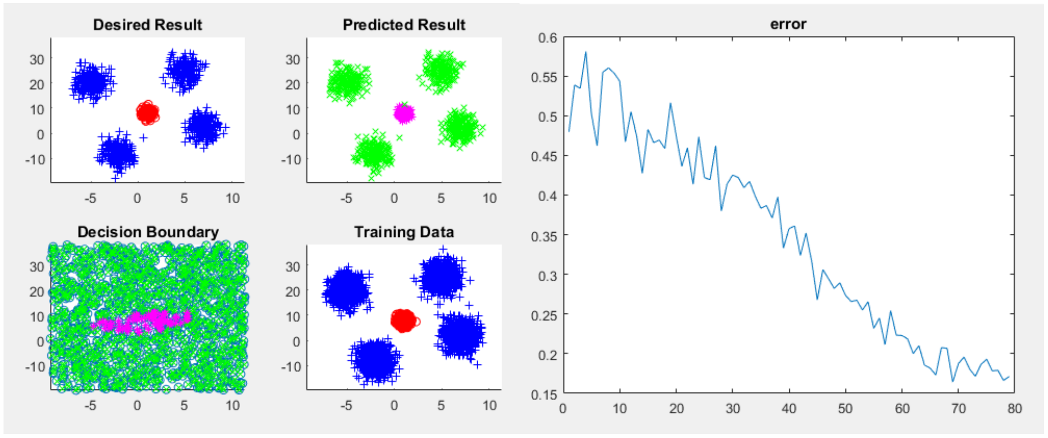


FIGURE 4.4 Feedback alignment generalization on distribution created by GenerateNonLinear2

It can be be observed from Figure 4.1, 4.2, 4.3 and 4.4, that the random feedback algorithm does indeed work.

4.2.1 Comparison of back-propagation and random feedback alignment

Since, the results of Figure 3.6, 3.7 and 3.8 is evaluated using back-propagation and Figure 4.1, 4.2, 4.3 and 4.4 is evaluated using random feedback with the exact same network configuration. The errors has been extracted for analysis.

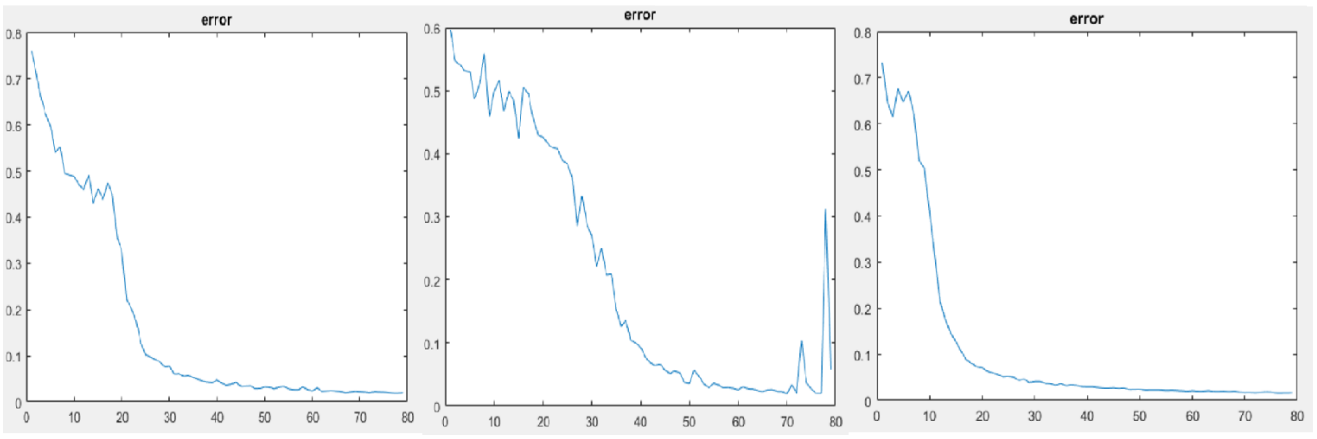


FIGURE 4.5 Cross entropy error function over epochs for XOR, GenerateNonLinear2 and GenerateNonLinear3 (also presented in the same order above) using back-propagation

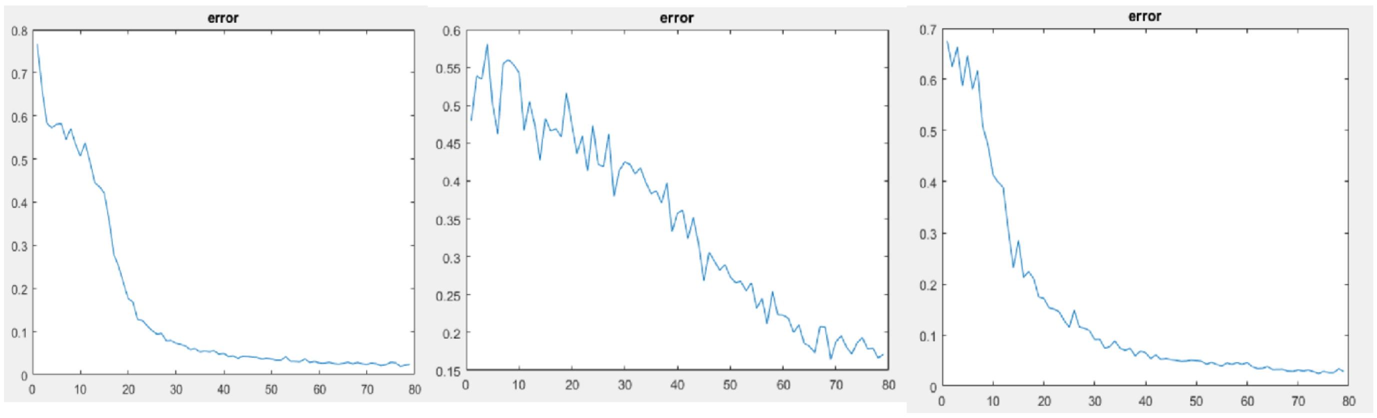


FIGURE 4.6 Cross entropy error function over epochs for XOR, GenerateNonLinear2 and GenerateNonLinear3 (also presented in the same order above) using random feedback

It can be observed that the error for back-propagation tends to converge quicker than random feedback. Error from back-propagtion also gave a smoother graph than random feedback.

Using $three\_layer\_network\_train$ function with the following network configuration:max epoch at 80, hidden layer with 8 neuron, learning rate of 0.2, min error of 0.005 and mini-batch of 200 inputs.

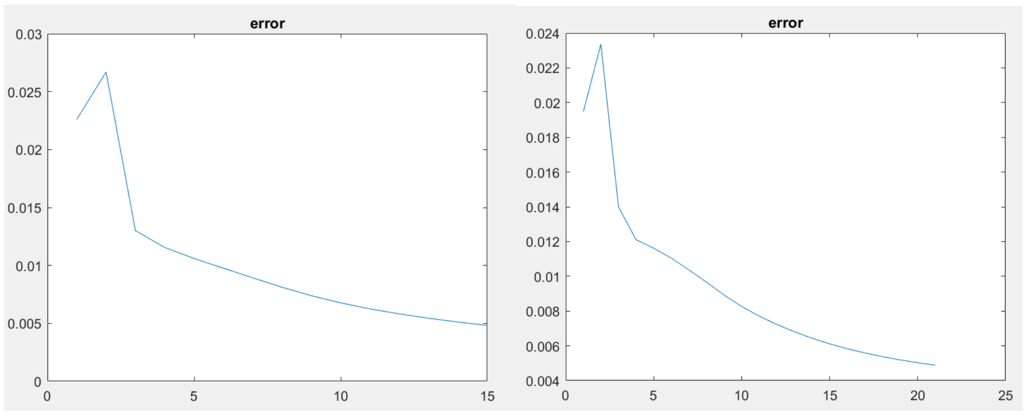


FIGURE 4.7 Cross entropy error function over GenerateNonLinear, back-propagation on the left side, random feedback on the right

It can be observed again, that the error for back-propagation tends to converge slightly quicker than random feedback.

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. Slight modification applied to $Testing\_Mnist$ script (Full code is available in the appendix which can be accessed from here) provided by Dr. Ian Howard, to load 15000 training dataset and 2000 testing dataset. The dataset being multi-class, as a preprocessing step, one-hot coding was applied to the target data to ensure softmax function evaluated correctly.

Through trial and error, it was deduced that two layer with 10 nodes on each layer gave better results than other network configurations. Using the same parameters and configuration, back-propagation and random feedback was used.

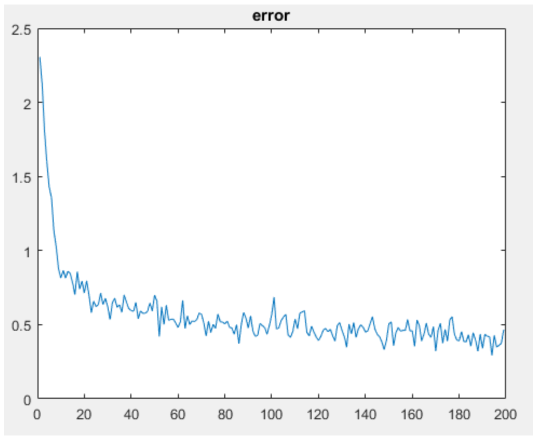


FIGURE 4.8 Cross entropy error function over epochs

Back-propagation classified with 82.7 percent accuracy.

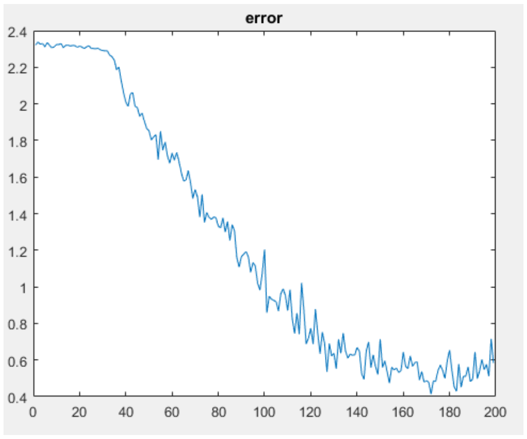


FIGURE 4.9 Cross entropy error function over epochs

Feedback alignment classified with 83.3 percent accuracy.

It can be observed that back-propagation converged quicker than

4.2.2 Random weights from normal distribution

[4] uses uniform distribution for generating the random weights, which has a constant probability. To evaluate the effects of another distribution, choosing the normal distribution.

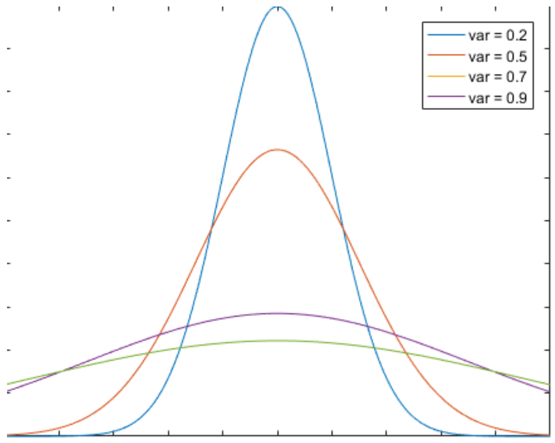


FIGURE 4.10 normal distribution

The case for randomising the weights has been changed for the weights to be randomised from the normal distribution

```
%the distribution of random weights for Feedback Alignment set to uniform if distribution=0
%else, uses the value of the distribution as the variance for uniformly distributed weights
if distribution == 0
    weights randomised over lilcrap uniform distribution between -0.5 and 0.5
else


        generates random weight with
```

```
mean = 0;
variance = distribution;
```

The $Normal\_dis\_plot$ script can be used to evaluate randomised weights using normal distribution with several different variances for several different distributions of datasets. The script then plots different variance over an epoch for a generated dataset. Full code is available in the appendix which can be accessed from here.

$GenerateNonlinear$ gave the following result.

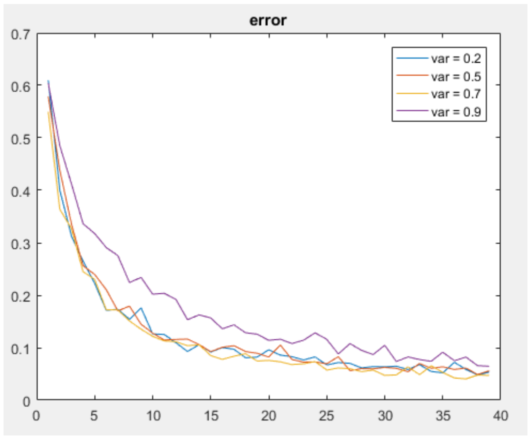


FIGURE 4.11 Comparing different variance

$GenerateNonlinear2$ gave the following result.

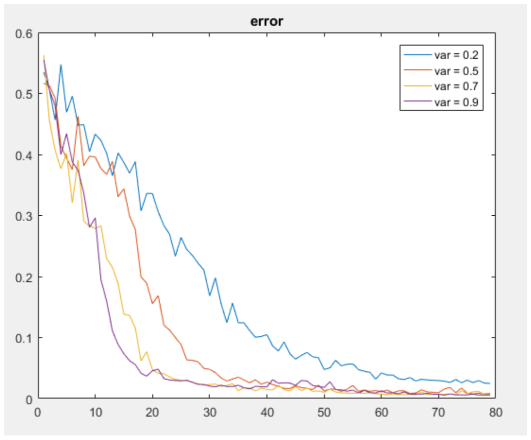


FIGURE 4.12 Comparing different variance

XOR gave the following result.

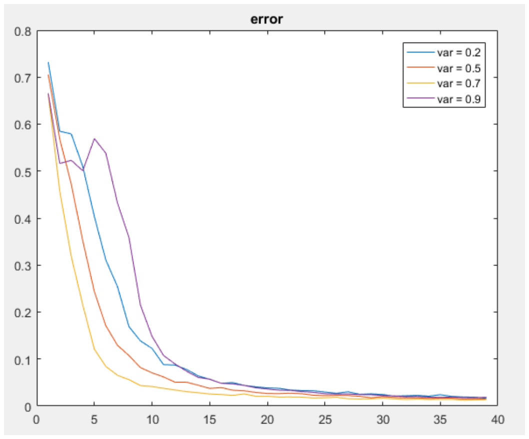


FIGURE 4.13 Comparing difference variance

CHAPTER 5

Summary and Future Work

5.1 Summary

The thesis started with a simple perceptron that was able to solve only linearly seperable functions. Implementation of a hidden layer allowed for a slightly better generalization. Multiple hidden layer in the network allowed for solving complex non-linear problems such as the X-OR problem and other non-linear distributions. Implementing feedback alignment algorithm suggest by [4], the algorithm was able to solve some of the non-linear distribution presented in the thesis (conditions was set exactly the same as back-propagtion). The two algorithms was presented with multi-output problem (Mnist). Both algorithm were able to classify to with roughly 17 percent error rate. It was also noted that random feedback algorithm was a bit slower to converge than the back-propagation algorithm. This was expected, and further verified [4] claims. The effects of the variance on the error when the random weights (of the feedback alignment) are normally distributed is evaluated and plots of several different distributions were compared. var of 0.7 seem to produce the quickest convergence in all cases. Consider 0.7 to be a variance in that is not too far off from the mean nor too close to the mean.

5.2 Future Work

For future work, it is highly suggest to optimise the gradient descent algorithm, as the training was extremly slow. [4] also looks into quadratic functions and deep non-linear functions. Further research can be carried on from there. Use of a larger training dataset, and more realist dataset would provide for a better generalization.

Bibliography

[1] S. Haykin. *Neural Networks A comprehensive Foundation*. Tom Robbins, Upper Saddle River, New Jersey 07458, 2 edition, 1999. chapter 1: introduction, pages 6-11.

[2] S. Haykin. *Neural Networks A comprehensive Foundation*. Tom Robbins, Upper Saddle River, New Jersey 07458, 2 edition, 1999. chapter 3: single-layer perceptrons , pages 128-142.

[3] S. Haykin. *Neural Networks A comprehensive Foundation*. Tom Robbins, Upper Saddle River, New Jersey 07458, 2 edition, 1999. chapter 4: multi-layer perceptrons , pages 156-178.

[4] T. P. Lillicrap. Random synaptic feedback weights support. *Nat. Commun*, 7(13276), Nov 2010. Date accessed: 03/09/2017.

[5] I. T. Nabney. *NETLAB Algorithms for Pattern Recognition*. Springer, London, 1 edition, 2002. chapter 4,5 pages 117-187.

[6] P. Picton. *Neural Networks*. PALGRAVE, Hampshire, 2 edition, 1994. chapter 1,2,3, pages 1-48.

[7] Stork. Is backpropagation biologically plausible? *IEEE Xplore*, Aug 2002. Date accessed: 06/09/2017.

Appendix

GenerateGaussianDataLS

```
function [data targetVec] = GenerateGaussianDataLS(samples)
% Generates a two-class dataset in matrix 'data' with 2D data vectors and
% also indicates each pattern class in the matrix 'targetVec'

% Function first generates 'samples' 2-d data vectors of class 1 by independently
% drawing each dimension from a  Gaussian distribution.
% class 1 patterns have an associated output label vector in matrix targetVec [1 0]'
% The  mean and covariance for class 1 are
% Mean1 = [4; -1;];
% Sigma1 = [0.6 .8; .8 7];

% Function then generates 'samples' 2-d data vectors of class 2 by independently
% drawing each dimension from a  Gaussian distribution.
% class 2 patterns have an associated output label vector in matrix targetVec[0 1]'
% The  mean and covariance for class 2 are
% Mean2 = [-1; 1;];
% Sigma2 = [0.7 .8; .8 9];

% the two sets of class1 and class2 data are then concatenated into the
% output  matrix 'data'
% similarly the label vectors are concatenated onto the output matrix
% 'targetVec'

% Both output matrices therefore have size = (2, 2 * samples)

% see example usage at end of function
% set demo = 1 to run this plot example
%demo = 0; % demo off
demo = 1; % demo on

% Ian Howard, University of Plymouth, December 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% set mean and covariance for class 0
Mean1 = [4; -1;];
Sigma1 = [0.6 .8; .8 7];

% set mean and covariance for class 1
Mean2 = [-4; 1;];
Sigma2 = [0.7 .8; .8 9];


% generate Gaussian distibuted dataset for class 0
data1 =  chol(Sigma1) * randn(2, samples) + repmat(Mean1,1,samples);
% generate target vectors for class 0
label1 = repmat([1; 0;] , 1, samples);
disp(cov(data1'));

% generate Gaussian distibuted datasetn for class 1
```

```
data2 =  chol(Sigma2) * randn(2, samples) + repmat(Mean2,1,samples);
% generate target vectors for class 1
label2 = repmat([0; 1;] , 1, samples);
disp(cov(data2'));

% concatenate class0 and class1 datasets into single dataset
data = [data1 data2];
targetVec = [label1 label2];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(demo)
    % example usage
    % vectorized example to extract all class 1 patterns
    % examine first dimension which is 1 for class 1
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 1);
    c1data = data(:,fidx);

    % vectorized example to extract all class 2 patterns
    % examine first dimension which is 0 for class 2
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 0);
    c2data = data(:,fidx);

    % now plot separated classes  on a figure
    figure
    hold on
    plot(c1data(1,:), c1data(2,:), 'ro');
    plot(c2data(1,:), c2data(2,:), 'b+');
    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class1 patterns in red and class2 patterns in blue');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

GenerateNonLinear

```
\label{genNonLin}
function [data targetVec] = GenerateNonLinear(samples)
% Generates a two-class dataset in matrix 'data' with 2D data vectors and
% also indicates each pattern class in the matrix 'targetVec'

% Function first generates 'samples' 2-d data vectors of class 1 by independently
% drawing each dimension from a  Gaussian distribution.
% class 1 patterns have an associated output label vector in matrix targetVec [1 0]'
% The  mean and covariance for class 1 are
% Mean1 = [4; -1;];
% Sigma1 = [0.6 .8; .8 7];

% Function then generates 'samples' 2-d data vectors of class 2 by independently
% drawing each dimension from a  Gaussian distribution.
% class 2 patterns have an associated output label vector in matrix targetVec[0 1]'
% The  mean and covariance for class 2 are
% Mean2 = [-1; 1;];
% Sigma2 = [0.7 .8; .8 9];

% the two sets of class1 and class2 data are then concatenated into the
% output  matrix 'data'
% similarly the label vectors are concatenated onto the output matrix
% 'targetVec'

% Both output matrices therefore have size = (2, 2 * samples)

% see example usage at end of function
% set demo = 1 to run this plot example
%demo = 1; % demo off
demo = 1; % demo on

% Ian Howard, University of Plymouth, December 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Mean1 = [-15; -18;];
Sigma1 = [0.1 0; -4 1];


% set mean and covariance for class 1
Mean2 = [-5; 0;];
Sigma2 = [0.7 .8; .8 9];

Mean3 = [3; 0;];
Sigma3 = [0.9 -.8; 0.7 7];


% generate Gaussian distibuted dataset for class 0
data1 =  (chol(Sigma1) * randn(2, samples) + repmat(Mean1,1,samples))+14;
% generate target vectors for class 0
label1 = repmat([1; 0] , 1, samples);

disp(cov(data1'));

% generate Gaussian distibuted datasetn for class 1
data2 =  chol(Sigma2) * randn(2, samples) + repmat(Mean2,1,samples);
% generate target vectors for class 1
label2 = repmat([0; 1] , 1, samples);

disp(cov(data2'));

% label 1
data3 =  chol(Sigma3) * randn(2, samples) + repmat(Mean3,1,samples);
% generate target vectors for class 1
label3 = repmat([0; 1] , 1, samples);
disp(cov(data3'));

%put it together
```

```
data2 = [data2 data3];
label2 = [label2 label3];

% concatenate class0 and class1 datasets into single dataset
data = [data1 data2];
targetVec = [label1 label2];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(demo)
    % example usage
    % vectorized example to extract all class 1 patterns
    % examine first dimension which is 1 for class 1
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 1);
    c1data = data(:,fidx);

    % vectorized example to extract all class 2 patterns
    % examine first dimension which is 0 for class 2
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 0);
    c2data = data(:,fidx);


%     fidx = find(targetVec2(1,:) == 3);
%     c3data = data(:,fidx);


    % now plot separated classes  on a figure
    figure
    hold on
   plot(c1data(1,:), c1data(2,:), 'ro');
    plot(c2data(1,:), c2data(2,:), 'b+');

    %plot(c3data(1,:), c3data(2,:), 'c*');

    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class1 patterns in red and class2 patterns in blue');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

GenerateNonLinear2

```
\label{genNonLin2}
function [data targetVec] = GenerateNonLinear2(samples)
% Generates a two-class dataset in matrix 'data' with 2D data vectors and
% also indicates each pattern class in the matrix 'targetVec'

% Function first generates 'samples' 2-d data vectors of class 1 by independently
% drawing each dimension from a  Gaussian distribution.
% class 1 patterns have an associated output label vector in matrix targetVec [1 0]'
% The  mean and covariance for class 1 are
% Mean1 = [4; -1;];
% Sigma1 = [0.6 .8; .8 7];

% Function then generates 'samples' 2-d data vectors of class 2 by independently
% drawing each dimension from a  Gaussian distribution.
% class 2 patterns have an associated output label vector in matrix targetVec[0 1]'
% The  mean and covariance for class 2 are
% Mean2 = [-1; 1;];
% Sigma2 = [0.7 .8; .8 9];

% the two sets of class1 and class2 data are then concatenated into the
% output  matrix 'data'
% similarly the label vectors are concatenated onto the output matrix
% 'targetVec'

% Both output matrices therefore have size = (2, 2 * samples)

% see example usage at end of function
% set demo = 1 to run this plot example
%demo = 1; % demo off
demo = 1; % demo on

% Ian Howard, University of Plymouth, December 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Mean1 = [-13; -6;];
Sigma1 = [0.1 0; -4 1];


% set mean and covariance for class 1
Mean2 = [-5; 20;];
Mean3 = [5; 25;];
Mean4 = [-2; -8;];
Mean5 = [7; 2;];

Sigma2 = [0.7 0; .8 9];





%og
% Mean2 = [-5; 0;];
% Sigma2 = [0.7 .8; .8 9];
%
% Mean3 = [3; 0;];
% Sigma3 = [0.9 -.8; 0.7 7];

% generate Gaussian distibuted dataset for class 0
data1 =  (chol(Sigma1) * randn(2, samples) + repmat(Mean1,1,samples))+14;
% generate target vectors for class 0
label1 = repmat([1; 0] , 1, samples);

disp(cov(data1'));

% generate Gaussian distibuted datasetn for class 1
data2 =  chol(Sigma2) * randn(2, samples) + repmat(Mean2,1,samples);
% generate target vectors for class 1
label2 = repmat([0; 1] , 1, samples);
```

```
disp(cov(data2'));

data3 =  chol(Sigma2) * randn(2, samples) + repmat(Mean3,1,samples);
% generate target vectors for class 1
label3 = repmat([0; 1] , 1, samples);
disp(cov(data3'));

data4 =  chol(Sigma2) * randn(2, samples) + repmat(Mean4,1,samples);
% generate target vectors for class 1
label4 = repmat([0; 1] , 1, samples);
disp(cov(data3'));

data5 =  chol(Sigma2) * randn(2, samples) + repmat(Mean5,1,samples);
% generate target vectors for class 1
label5 = repmat([0; 1] , 1, samples);
disp(cov(data3'));


%put it together
data2 = [data2 data3 data4 data5];
label2 = [label2 label3 label4 label5];

% concatenate class0 and class1 datasets into single dataset
data = [data1 data2];
targetVec = [label1 label2];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(demo)
    % example usage
    % vectorized example to extract all class 1 patterns
    % examine first dimension which is 1 for class 1
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 1);
    c1data = data(:,fidx);

    % vectorized example to extract all class 2 patterns
    % examine first dimension which is 0 for class 2
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 0);
    c2data = data(:,fidx);


%     fidx = find(targetVec2(1,:) == 3);
%     c3data = data(:,fidx);


    % now plot separated classes  on a figure
    figure
    hold on
   plot(c1data(1,:), c1data(2,:), 'ro');
    plot(c2data(1,:), c2data(2,:), 'b+');

    %plot(c3data(1,:), c3data(2,:), 'c*');

    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class1 patterns in red and class2 patterns in blue');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

$GenerateNonLinear3$

```
\label{genNonLin3}
function [data targetVec] = GenerateNonLinear3(samples)
% Generates a two-class dataset in matrix 'data' with 2D data vectors and
% also indicates each pattern class in the matrix 'targetVec'

% Function first generates 'samples' 2-d data vectors of class 1 by independently
% drawing each dimension from a  Gaussian distribution.
% class 1 patterns have an associated output label vector in matrix targetVec [1 0]'
% The  mean and covariance for class 1 are
% Mean1 = [4; -1;];
% Sigma1 = [0.6 .8; .8 7];

% Function then generates 'samples' 2-d data vectors of class 2 by independently
% drawing each dimension from a  Gaussian distribution.
% class 2 patterns have an associated output label vector in matrix targetVec[0 1]'
% The  mean and covariance for class 2 are
% Mean2 = [-1; 1;];
% Sigma2 = [0.7 .8; .8 9];

% the two sets of class1 and class2 data are then concatenated into the
% output  matrix 'data'
% similarly the label vectors are concatenated onto the output matrix
% 'targetVec'

% Both output matrices therefore have size = (2, 2 * samples)

% see example usage at end of function
% set demo = 1 to run this plot example
%demo = 1; % demo off
demo = 1; % demo on

% Ian Howard, University of Plymouth, December 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%samples = 1000;
Mean1 = [4; -1;];
Sigma1 = [0.6 .8; .8 7];

% set mean and covariance for class 1
Mean2 = [-4; 1;];
Sigma2 = [0.7 .8; .8 9];

Mean3 = [-18; 1;];
Sigma3 = [0.9 -.9; -.8 9];


% generate Gaussian distibuted dataset for class 0
data1 =  chol(Sigma1) * randn(2, samples) + repmat(Mean1,1,samples);
% generate target vectors for class 0
label1 = repmat([1; 0] , 1, samples);

disp(cov(data1'));

% generate Gaussian distibuted datasetn for class 1
data2 =  chol(Sigma2) * randn(2, samples) + repmat(Mean2,1,samples);
% generate target vectors for class 1
label2 = repmat([0; 1] , 1, samples);

disp(cov(data2'));

% label 1
data3 =  chol(Sigma3) * randn(2, samples) + repmat(Mean3,1,samples);
% generate target vectors for class 1
label3 = repmat([1; 0] , 1, samples);
disp(cov(data3'));

%put it together
data1 = [data1 data3];
```

```
label1 = [label1 label3];

% concatenate class0 and class1 datasets into single dataset
data = [data1 data2];
targetVec = [label1 label2];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(demo)
    % example usage
    % vectorized example to extract all class 1 patterns
    % examine first dimension which is 1 for class 1
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 1);
    c1data = data(:,fidx);

    % vectorized example to extract all class 2 patterns
    % examine first dimension which is 0 for class 2
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 0);
    c2data = data(:,fidx);


%     fidx = find(targetVec2(1,:) == 3);
%     c3data = data(:,fidx);


    % now plot separated classes  on a figure
    figure
    hold on
   plot(c1data(1,:), c1data(2,:), 'ro');
    plot(c2data(1,:), c2data(2,:), 'b+');

    %plot(c3data(1,:), c3data(2,:), 'c*');

    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class1 patterns in red and class2 patterns in blue');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

loadMnist

```
\label{loadMnist}
% read testing dataset
% testing data filenames
imgFile = 't10k-images-idx3-ubyte';
labelFile = 't10k-labels-idx1-ubyte';
% start from beginning
offset=0;
% only want 1000 patterns
readDigits=2000;
% Read digits and labels from MNIST database by Sid H
[imgs TestLabels] = readMNIST(imgFile, labelFile, readDigits, offset);
TestData=[];
for idx =1:length(imgs)
 tmp = imgs(:,:,idx);
 TestData(idx,:) = tmp(:);
end

disp('Loaded testing dataset');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% read training dataset
% training data filenames
imgFile = 'train-images-idx3-ubyte';
labelFile = 'train-labels-idx1-ubyte';
% start from beginning
offset=0;
% only want 1000 patterns
readDigits=15000;
% Read digits and labels from MNIST database by Sid H
[imgs TrainLabels] = readMNIST(imgFile, labelFile, readDigits, offset);
TrainData=[];
for idx =1:length(imgs)
 tmp = imgs(:,:,idx);
 TrainData(idx,:) = tmp(:);
end
disp('Loaded training dataset');


% plot all iamges in training dataset
%PlotAllImages(TrainData,'Dataset', 10);
```

matlab_4hid_out

```
\label{matlab}
% Solve a Pattern Recognition Problem with a Neural Network
% Script generated by Neural Pattern Recognition app
% Created 19-Aug-2017 17:20:22
%
% This script assumes these variables are defined:
%
%   trainingData - input data.
%   trainingT - target data.

x = trainingData;
t = trainingTarget;

%t(2,:) = [];

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
trainFcn = 'trainscg';  % Scaled conjugate gradient backpropagation.

% Create a Pattern Recognition Network
hiddenLayerSize = 4;
net = patternnet(hiddenLayerSize);

% Choose Input and Output Pre/Post-Processing Functions
% For a list of all processing functions type: help nnprocess
net.input.processFcns = {'removeconstantrows','mapminmax'};
net.output.processFcns = {'removeconstantrows','mapminmax'};

% Setup Division of Data for Training, Validation, Testing
% For a list of all data division functions type: help nndivide
net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample
net.divideParam.trainRatio = 100/100;

%net.divideParam.valRatio = 5/100;
%net.divideParam.testRatio = 5/100;

% Choose a Performance Function
% For a list of all performance functions type: help nnperformance
net.performFcn = 'crossentropy';  % Cross-Entropy

% Choose Plot Functions
% For a list of all plot functions type: help nnplot
net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
    'plotconfusion', 'plotroc'};

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)
tind = vec2ind(t);
yind = vec2ind(y);
percentErrors = sum(tind ~= yind)/numel(tind);

% Recalculate Training, Validation and Test Performance
trainTargets = t .* tr.trainMask{1};
valTargets = t .* tr.valMask{1};
testTargets = t .* tr.testMask{1};
trainPerformance = perform(net,trainTargets,y)
valPerformance = perform(net,valTargets,y)
testPerformance = perform(net,testTargets,y)
```

```
% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
%figure, plotconfusion(t,y)
%figure, plotroc(t,y)

% Deployment
% Change the (false) values to (true) to enable the following code blocks.
% See the help for each generation function for more information.
if (false)
    % Generate MATLAB function for neural network for application
    % deployment in MATLAB scripts or with MATLAB Compiler and Builder
    % tools, or simply to examine the calculations your trained neural
    % network performs.
    genFunction(net,'myNeuralNetworkFunction');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a matrix-only MATLAB function for neural network code
    % generation with MATLAB Coder tools.
    genFunction(net,'myNeuralNetworkFunction','MatrixOnly','yes');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a Simulink diagram for simulation or deployment with.
    % Simulink Coder tools.
    gensim(net);
end
```

multiple_layer_network_test

```
\label{multiTest}
function [error, result] = multiple_layer_network_test(weights, testingData, testingTarget,no_of_output, no_of_layers
     , plot)

ndata = size(testingData, 1);
bias_appended_input = [ones(ndata, 1), testingData];

all_weights = weights;

incorrect_pred = 0;

if no_of_output == 1
    output_function = 'single';
else
    output_function = 'multi';
end

no_of_weights = no_of_layers + 1;

result = [];

if (plot == 1)

    for n=1:ndata
        error = incorrect_pred;

        for a = 1:no_of_layers

            %forward propagation
            if a == 1

                forward_propagation_hidden = bias_appended_input(n,:)*all_weights.(strcat('weight',num2str(a)));
                % sigmoid function
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                %add bias
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            else
                forward_propagation_hidden = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))*
                     all_weights.(strcat('weight',num2str(a)));
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            end
            %weight for first layer update
        end


        switch output_function

            case 'single'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));
                y = 1./(1 + exp(-forward));


                if y < 0.5
                    output = 0;
                else
                    output = 1;
                end

            case 'multi'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));

                temp = exp(forward);

                y = temp./(sum(temp,2)*ones(1,no_of_output));

                [t_output ind_t_output] = max(y);

                output = ind_t_output;

        end
        result = [result ; output];

    end

else
    for n=1:ndata
        for a = 1:no_of_layers

            %forward propagation
            if a == 1

                forward_propagation_hidden = bias_appended_input(n,:)*all_weights.(strcat('weight',num2str(a)));
                % sigmoid function
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                %add bias
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            else
                forward_propagation_hidden = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))*
                     all_weights.(strcat('weight',num2str(a)));
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            end
            %weight for first layer update
        end

        switch output_function

            case 'single'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));
                y = 1./(1 + exp(-forward));

                target = testingTarget(n);

                if y < 0.5
                    output = 0;
                else
                    output = 1;
                end
            case 'multi'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));

                temp = exp(forward);

                y = temp./(sum(temp,2)*ones(1,no_of_output));

                [t_output ind_t_output] = max(y);

                output = ind_t_output;

                [t indTarget] = max(testingTarget(n,:));

                target = indTarget;
        end

        if target ~= output

            incorrect_pred = incorrect_pred + 1;
        end
        result = [result ; output];
    end


    error = incorrect_pred;
end
```

multiply_layer_network_train

```
\label{multiTrain}
function [optimised_weights, plot_error] = multiple_layer_network_train(epochs,input, targets, no_of_output,
     net_architecture, output_function,error_function, algorithm, learning,momentum, learning_rate, error_val,
     mb_value, partial_con,distribution)
% epochs: max number of iterations
% input: the input dataset
% targets: corresponding target values for the inputs
% no_of_output: total number of nodes in the output layer
% net_architecture: the hidden architecture is presented with this value in an array
% output_function: depending on the number of output, either sigmoid or softmax is applied
% error_function: either sigmoid or softmax, works out the relevant error
% partial_con: partial connectivity, 1=active, 0= inactive, if active,
% removes some of hidden layer synaptic weights
% distribution: 1 for uniform distribution, else uses the value as variance for normal distribution

%%%This function allows to create any number of layers. extensively made
%%%use of structural array...

no_of_layers = size(net_architecture, 2);
no_of_weights = no_of_layers + 1;

ndata = size(input, 1);
ntargets = size(targets, 2);

if no_of_output == 1
    out_funct = 'single';
else
    out_funct = 'multi';
end


%randomise weights and creates number weights depending on the number of layers
%three different conditions for initializing the weights, as the weights
%between the hidden layers would be depended on the number of layers, the
%elseif allows to correctly created weights of any number of layers
for n= 1:no_of_weights

    if n == 1
        weight = (rand(size(input, 2), net_architecture(n))/sqrt(size(input, 2) + 1))';
        bias =  rand(1, net_architecture(n))/sqrt(size(input, 2) + 1);

        weights = [bias' weight]';

        temp = strcat('weight',num2str(n));
        all_weights.(temp) = weights;

        %structural array to store these weights in batch mode, numbered
        [a b] = size(weights);
        batch = zeros(a,b);
        temp1 = strcat('grad',num2str(n));
        batch_delta_grad.(temp1) = batch;

    elseif n < (no_of_weights)
        weight = (rand(net_architecture(n-1), net_architecture(n))/sqrt(net_architecture(n-1) + 1))';


        bias =  rand(1, net_architecture(n))/sqrt(net_architecture(n-1) + 1);

        weights = [bias' weight]';

        %randomly zero half of the connection of a node in the hidden layer
        %to the next, other than the bias weights
        if partial_con == 1
            [partial_a partial_b] = size(weights);

            weights_reduce = ceil((partial_a-1)/2);
            %loops through the matrix, to generate a non repeating vector
            for u=1:partial_b

                for v=1:weights_reduce
                    partial_array=[];
                    while length(partial_array) < weights_reduce
                        z=randperm(partial_a-1,1);
                        if z==1
                            z=z+1;
                        end
                        partial_array=[partial_array z];
                        partial_array=unique(partial_array);
                    end

                    weights(partial_array(v),u)=0;
                end
            end

        end


        temp = strcat('weight',num2str(n));
        all_weights.(temp) = weights;

        [a b] = size(weights);
        batch = zeros(a,b);
        temp1 = strcat('grad',num2str(n));
        batch_delta_grad.(temp1) = batch;

    else
        end_weight = (rand(net_architecture(n-1), ntargets)/sqrt(net_architecture(n-1) + 1))';
        bias = rand(1, ntargets)/sqrt(net_architecture(n-1) + 1);

        weights = [bias' end_weight]';

        temp = strcat('weight',num2str(n));
        all_weights.(temp) = weights;

        [a b] = size(weights);
        batch = zeros(a,b);
        temp1 = strcat('grad',num2str(n));
        batch_delta_grad.(temp1) = batch;

    end


end

%the distribution of random weights for Feedback Alignment set to uniform if distribution=0
%else, uses the value of the distribution as the variance for uniformly distributed weights
if distribution == 0
    for r=1:no_of_weights

        [a,b] = size(all_weights.(strcat('weight',num2str(r))));

        %lilcrap uniform distribution over -0.5 and 0.5
        random_weight = -.5+(.5+.5)*rand(a,b);

        temp = strcat('weight',num2str(r));
        all_random_weights.(temp) = random_weight;

    end
else
    for r=1:no_of_weights

        [a,b] = size(all_weights.(strcat('weight',num2str(r))));

        %lilcrap uniform distribution over -0.5 and 0.5
        mean = 0;
        variance = distribution;
        random_weight = mean + variance * randn(a,b);

        temp = strcat('weight',num2str(r));
        all_random_weights.(temp) = random_weight;

    end

end

bias_appended_input = [ones(ndata, 1), input];

total_error = 0;
all_error = [];

iter = 1;
while iter < epochs

    random = randi([1 ndata],1,mb_value);
    %mini batch
    for m = 1:mb_value;

        n=random(m);

        %similar to how the weights were initialised, only with forward
        %propagation
        for a = 1:no_of_layers

            %forward propagation
            %makes use of structural array again.
            if a == 1

                forward_propagation_hidden = bias_appended_input(n,:)*all_weights.(strcat('weight',num2str(a)));
                %activation function
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                %add bias
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            else
                forward_propagation_hidden = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))*
                     all_weights.(strcat('weight',num2str(a)));
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            end

        end

        switch output_function

            case 'sigmoid'

                %retrieves the relevant value depending on the number of
                %layers
                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));
                y = 1./(1 + exp(-forward));

            case 'softmax'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));

                temp = exp(forward);

                y = temp./(sum(temp,2)*ones(1,no_of_output));

        end

        switch out_funct
            case 'single'

                switch algorithm

                    case 'back-prop'
                        % back propagation

                        %last layer of weights
                        delta_target = -(targets(n,:) - y);
                        sig_der_out = y*(1-y);

                        sigma_1 = delta_target*(sig_der_out);

                        delta_weight_last= sigma_1*all_forward_propagation_bias.((strcat('f_prop',num2str(
                             no_of_layers))));

                        temp = strcat('delta_w',num2str(no_of_weights));
                        all_delta_weights.(temp) = delta_weight_last';

                        %depending on the number of layer, the gradient with respect to the weight is calculated
                        for a = no_of_layers:-1:1

                            %the
                            counter_weight = no_of_weights-1;

                            %this is the weight before the second last layer of
                            %the network...
                            if a == no_of_layers
                                weight_no_bias =  all_weights.(strcat('weight',num2str(a+1)));
                                weight_no_bias(1,:) = [];

                                sigma_2 = sigma_1 *weight_no_bias;

                                fp =all_forward_propagation.((strcat('f_prop',num2str(a))));
                                sig_der_nxt = fp.*(1-fp);

                                delta_weight = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))'*(
                                     sigma_2'.*sig_der_nxt);

                                temp = strcat('delta_w',num2str(a));
                                all_delta_weights.(temp) = delta_weight;

                                %%%
                            else
                                depth = no_of_weights - a;

                                for b=depth:-1:1
                                    if b == depth
                                        f = all_forward_propagation.((strcat('f_prop',num2str(no_of_layers))));
                                        deri_layer = f.*(1-f);
                                        error_layer = sigma_2.*deri_layer';

                                        temp = strcat('error_',num2str(b));
                                        all_hidden_layer_error.(temp) = error_layer;

                                    elseif b == 1
                                        wei = all_weights.(strcat('weight',num2str(a+1)));
                                        wei(1,:) = [];

                                        el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));
                            error_carry = wei*el;

                            g = all_forward_propagation.((strcat('f_prop',num2str(b+1))));
        %

                            if a == 1
                                %use input
                                g = all_forward_propagation.((strcat('f_prop',num2str(1))));
                                deri_layer = g.*(1-g);
                                deri_layer=deri_layer';

                                delta_weight = bias_appended_input(n,:)'*(error_carry.*deri_layer)';


                                temp = strcat('delta_w',num2str(a));
                                all_delta_weights.(temp) = delta_weight;

                            else

                                in = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))));

                                g = all_forward_propagation.((strcat('f_prop',num2str(no_of_weights-depth
                                     ))));
                                deri_layer = g.*(1-g);
                                deri_layer=deri_layer';

                                delta_weight = in'*(error_carry.*deri_layer)';

                                temp = strcat('delta_w',num2str(a));
                                all_delta_weights.(temp) = delta_weight;
                            end

                        else
                            wei = all_weights.(strcat('weight',num2str(counter_weight)));

                            wei(1,:) = [];

                            el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

                            error_carry = wei*el;

                            g = all_forward_propagation.((strcat('f_prop',num2str(counter_weight-1))));
                            deri_layer = g.*(1-g);
                            deri_layer=deri_layer';

                            error_layer = error_carry.*deri_layer;


                            temp = strcat('error_',num2str(b));
                            all_hidden_layer_error.(temp) = error_layer;

                            counter_weight = counter_weight - 1;
                        end
                    end
                end
            end

            switch learning
                case 'batch'
                    for z= 1:no_of_weights

                        this = batch_delta_grad.(strcat('grad',num2str(z))) + all_delta_weights.(strcat('
                             delta_w',num2str(z)));

                        temp = strcat('grad',num2str(z));
                        batch_delta_grad.(temp) = this;

                    end
                case 'online'

                    for x=1:no_of_weights

                        weight = all_weights.(strcat('weight',num2str(x)));

                        change = all_delta_weights.(strcat('delta_w',num2str(x)));

                        if partial_con == 1

                                [row_zero col_zero] = find(weight==0);

                                for r = 1:size(row_zero,1)
                                        change(row_zero(r),col_zero(r)) = 0;

                                end

                        end


                        updated_weight = weight*momentum - (learning_rate*change);

                        temp = strcat('weight',num2str(x));
                        all_weights.(temp) = updated_weight;
                    end

            end



        case 'feedback_alignment'

            %last layer of weights

            delta_target = -(targets(n,:) - y);
            sig_der_out = y*(1-y);

            sigma_1 = delta_target*(sig_der_out);

            delta_weight_last= sigma_1*all_forward_propagation_bias.((strcat('f_prop',num2str(
                 no_of_layers))));

            temp = strcat('delta_w',num2str(no_of_weights));
            all_delta_weights.(temp) = delta_weight_last';



            for a = no_of_layers:-1:1
                counter_weight = no_of_weights-1;

                if a == no_of_layers
                    %uses random weights instead of the weights in the network
                    ran_weight =  all_random_weights.(strcat('weight',num2str(a+1)));
                    ran_weight(1,:) = [];

                    sigma_2 = delta_target *ran_weight;

                    fp =all_forward_propagation.((strcat('f_prop',num2str(a))));
                    sig_der_nxt = fp.*(1-fp);


                    delta_weight = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))'*(
                         sigma_2'.*sig_der_nxt);

                    temp = strcat('delta_w',num2str(a));
                    all_delta_weights.(temp) = delta_weight;

                else
                    depth = no_of_weights - a;
```

```
for b=depth:-1:1
    if b == depth
        f = all_forward_propagation.((strcat('f_prop',num2str(no_of_layers))));
        deri_layer = f.*(1-f);
        error_layer = sigma_2.*deri_layer';

        temp = strcat('error_',num2str(b));
        all_hidden_layer_error.(temp) = error_layer;

    elseif b == 1
        wei = all_random_weights.(strcat('weight',num2str(a+1)));
        wei(1,:) = [];

        el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

        error_carry = wei*el;

        g = all_forward_propagation.((strcat('f_prop',num2str(b+1))));
%

        if a == 1

            g = all_forward_propagation.((strcat('f_prop',num2str(1))));
            deri_layer = g.*(1-g);
            deri_layer=deri_layer';

            delta_weight = bias_appended_input(n,:)'*(error_carry.*deri_layer)';


            temp = strcat('delta_w',num2str(a));
            all_delta_weights.(temp) = delta_weight;

        else

            in = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))));

            g = all_forward_propagation.((strcat('f_prop',num2str(no_of_weights-depth
                 ))));
            deri_layer = g.*(1-g);
            deri_layer=deri_layer';

            delta_weight = in'*(error_carry.*deri_layer)';

            temp = strcat('delta_w',num2str(a));
            all_delta_weights.(temp) = delta_weight;
        end

    else


        wei = all_weights.(strcat('weight',num2str(counter_weight)));

        wei(1,:) = [];

        el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

        error_carry = wei*el;

        g = all_forward_propagation.((strcat('f_prop',num2str(counter_weight-1))));
        deri_layer = g.*(1-g);
        deri_layer=deri_layer';

        error_layer = error_carry.*deri_layer;

        temp = strcat('error_',num2str(b));
        all_hidden_layer_error.(temp) = error_layer;
```

```
                            end
                        end
                    end
                end

                switch learning
                    case 'batch'
                        total_gradient_first = delta_weight_first + total_gradient_first;
                        total_gradient_second = total_gradient_second + delta_weight_second';
                    case 'online'

                        for x=1:no_of_weights

                            weight = all_weights.(strcat('weight',num2str(x)));
                            change = all_delta_weights.(strcat('delta_w',num2str(x)));

                            %partial connection
                            if partial_con == 1

                                    [row_zero col_zero] = find(weight==0);

                                    for r = 1:size(row_zero,1)
                                            change(row_zero(r),col_zero(r)) = 0;

                                    end

                            end

                            updated_weight = weight - (learning_rate*change);

                            temp = strcat('weight',num2str(x));
                            all_weights.(temp) = updated_weight;
                        end

                end


        end
    case 'multi'

        switch algorithm

            case 'back-prop'
                % back propagation

                %last layer of weights

                delta_target = -(targets(n,:) - y);
                sig_der_out = y.*(1-y);

                sigma_1 = delta_target.*(sig_der_out);

                delta_weight_last= all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))'*
                     sigma_1;

                temp = strcat('delta_w',num2str(no_of_weights));
                all_delta_weights.(temp) = delta_weight_last;



                for a = no_of_layers:-1:1

                    counter_weight = no_of_weights-1;

                    if a == no_of_layers
                        weight_no_bias =  all_weights.(strcat('weight',num2str(a+1)));
                        weight_no_bias(1,:) = [];

                        sigma_2 = weight_no_bias* sigma_1';
            fp =all_forward_propagation.((strcat('f_prop',num2str(a))));
            sig_der_nxt = fp.*(1-fp);

            delta_weight = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))'*(
                 sigma_2'.*sig_der_nxt);

            temp = strcat('delta_w',num2str(a));
            all_delta_weights.(temp) = delta_weight;

        else
            depth = no_of_weights - a;

            for b=depth:-1:1
                if b == depth
                    f = all_forward_propagation.((strcat('f_prop',num2str(no_of_layers))));
                    deri_layer = f.*(1-f);
                    error_layer = sigma_2.*deri_layer';

                    temp = strcat('error_',num2str(b));
                    all_hidden_layer_error.(temp) = error_layer;

                elseif b == 1
                    wei = all_weights.(strcat('weight',num2str(a+1)));
                    wei(1,:) = [];

                    el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

                    error_carry = wei*el;

                    g = all_forward_propagation.((strcat('f_prop',num2str(b+1))));
%                     deri_layer = g.*(1-g);
%                     deri_layer=deri_layer';

                    if a == 1
                        g = all_forward_propagation.((strcat('f_prop',num2str(1))));
                        deri_layer = g.*(1-g);
                        deri_layer=deri_layer';

                        delta_weight = bias_appended_input(n,:)'*(error_carry.*deri_layer)';


                        temp = strcat('delta_w',num2str(a));
                        all_delta_weights.(temp) = delta_weight;

                    else

                        in = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))));

                        g = all_forward_propagation.((strcat('f_prop',num2str(no_of_weights-depth
                             ))));
                        deri_layer = g.*(1-g);
                        deri_layer=deri_layer';

                        delta_weight = in'*(error_carry.*deri_layer)';

                        temp = strcat('delta_w',num2str(a));
                        all_delta_weights.(temp) = delta_weight;
                    end

                else
                    wei = all_weights.(strcat('weight',num2str(counter_weight)));

                    wei(1,:) = [];

                    el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

                    error_carry = wei*el;
                    g = all_forward_propagation.((strcat('f_prop',num2str(counter_weight-1))));
                    deri_layer = g.*(1-g);
                    deri_layer=deri_layer';

                    error_layer = error_carry.*deri_layer;

                    temp = strcat('error_',num2str(b));
                    all_hidden_layer_error.(temp) = error_layer;

                    counter_weight = counter_weight - 1;
                end

            end

        end

    end

    switch learning
        case 'batch'
            for z= 1:no_of_weights

                this = batch_delta_grad.(strcat('grad',num2str(z))) + all_delta_weights.(strcat('
                     delta_w',num2str(z)));

                temp = strcat('grad',num2str(z));
                batch_delta_grad.(temp) = this;

            end
        case 'online'

            for x=1:no_of_weights

                weight = all_weights.(strcat('weight',num2str(x)));
                change = all_delta_weights.(strcat('delta_w',num2str(x)));

                %partial connection
                if partial_con == 1

                        [row_zero col_zero] = find(weight==0);

                        for r = 1:size(row_zero,1)
                                change(row_zero(r),col_zero(r)) = 0;

                        end

                end

                updated_weight = weight - (learning_rate*change);

                temp = strcat('weight',num2str(x));
                all_weights.(temp) = updated_weight;
            end

    end


case 'feedback_alignment'

    %last layer of weights

    delta_target = -(targets(n,:) - y);
    sig_der_out = y.*(1-y);

    sigma_1 = delta_target.*(sig_der_out);

    delta_weight_last= all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))'*
         sigma_1;
```

```
temp = strcat('delta_w',num2str(no_of_weights));
all_delta_weights.(temp) = delta_weight_last;



for a = no_of_layers:-1:1

    counter_weight = no_of_weights-1;

    if a == no_of_layers

        ran_weight =  all_random_weights.(strcat('weight',num2str(a+1)));
        ran_weight(1,:) = [];

        sigma_2 = ran_weight*delta_target';

        fp =all_forward_propagation.((strcat('f_prop',num2str(a))));
        sig_der_nxt = fp.*(1-fp);

        delta_weight = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))'*(
             sigma_2'.*sig_der_nxt);

        temp = strcat('delta_w',num2str(a));
        all_delta_weights.(temp) = delta_weight;

    else
        depth = no_of_weights - a;

        for b=depth:-1:1
            if b == depth
                f = all_forward_propagation.((strcat('f_prop',num2str(no_of_layers))));
                deri_layer = f.*(1-f);
                error_layer = sigma_2.*deri_layer';

                temp = strcat('error_',num2str(b));
                all_hidden_layer_error.(temp) = error_layer;

            elseif b == 1
                wei = all_random_weights.(strcat('weight',num2str(a+1)));
                wei(1,:) = [];

                el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

                error_carry = wei*el;

                g = all_forward_propagation.((strcat('f_prop',num2str(b+1))));
%
                if a == 1

                    g = all_forward_propagation.((strcat('f_prop',num2str(1))));
                    deri_layer = g.*(1-g);
                    deri_layer=deri_layer';

                    delta_weight = bias_appended_input(n,:)'*(error_carry.*deri_layer)';

                    temp = strcat('delta_w',num2str(a));
                    all_delta_weights.(temp) = delta_weight;

                else

                    in = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))));

                    g = all_forward_propagation.((strcat('f_prop',num2str(no_of_weights-depth
                         ))));
                    deri_layer = g.*(1-g);
                    deri_layer=deri_layer';

                    delta_weight = in'*(error_carry.*deri_layer)';
```

```
                                    temp = strcat('delta_w',num2str(a));
                                    all_delta_weights.(temp) = delta_weight;
                                end

                            else

                                wei = all_weights.(strcat('weight',num2str(counter_weight)));

                                wei(1,:) = [];

                                el = all_hidden_layer_error.(strcat('error_',num2str(b+1)));

                                error_carry = wei*el;

                                g = all_forward_propagation.((strcat('f_prop',num2str(counter_weight-1))));
                                deri_layer = g.*(1-g);
                                deri_layer=deri_layer';

                                error_layer = error_carry.*deri_layer;

                                temp = strcat('error_',num2str(b));
                                all_hidden_layer_error.(temp) = error_layer;

                                counter_weight = counter_weight - 1;
                            end

                        end

                    end

                end

                switch learning
                    case 'batch'
                        total_gradient_first = delta_weight_first + total_gradient_first;
                        total_gradient_second = total_gradient_second + delta_weight_second';
                    case 'online'

                        for x=1:no_of_weights

                            weight = all_weights.(strcat('weight',num2str(x)));
                            change = all_delta_weights.(strcat('delta_w',num2str(x)));

                            %partial connection
                            if partial_con == 1

                                    [row_zero col_zero] = find(weight==0);

                                    for r = 1:size(row_zero,1)
                                            change(row_zero(r),col_zero(r)) = 0;

                                    end

                            end

                            updated_weight = weight - (learning_rate*change);

                            temp = strcat('weight',num2str(x));
                            all_weights.(temp) = updated_weight;
                        end

                end



        end
end
```

```
    switch error_function
        case 'cross entropy s'
            %calculating error error
            %cross entropy error function
            err = -(targets(n).*log(y) + (1-targets(n)).*log(1-y));
            total_error = err + total_error;

        case 'cross entropy m'
            err = -sum(targets(n,:).*log(y));

            total_error = err + total_error;
    end
end

%updating weights with batch training
if strcmp(learning,'batch') == 1

    for w= 1:no_of_weights

        this = batch_delta_grad.(strcat('grad',num2str(w)))/ndata;

        temp = strcat('weight',num2str(w));
        update = all_weights.(temp) - learning_rate*this;

         %partial connection
                                if partial_con == 1

                                        [row_zero col_zero] = find(weight==0);

                                        for r = 1:size(row_zero,1)
                                                this(row_zero(r),col_zero(r)) = 0;

                                        end

                                end

        temp = strcat('weight',num2str(w));
        all_weights.(temp) = update;

    end

    %zero the batch total
    for x=1:no_of_weights

        zero = batch_delta_grad.(strcat('grad',num2str(x)));
        change = 0* zero;
        temp = strcat('grad',num2str(x));
        all_weights.(temp) = change;
    end
end

%mean batch error
berror = 0 ;
for n= 1:ndata

    for a = 1:no_of_layers

        %forward propagation
        if a == 1
            forward_propagation_hidden = bias_appended_input(n,:)*all_weights.(strcat('weight',num2str(a)));
            % sigmoid function
            forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
            %add bias
            forward_propagation_bias = [ones(1,1), forward_propagation_activation];

            temp = strcat('f_prop',num2str(a));
            all_forward_propagation_bias.(temp) = forward_propagation_bias;
            all_forward_propagation.(temp) = forward_propagation_activation;
            else
                forward_propagation_hidden = all_forward_propagation_bias.((strcat('f_prop',num2str(a-1))))*
                     all_weights.(strcat('weight',num2str(a)));
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                temp = strcat('f_prop',num2str(a));
                all_forward_propagation_bias.(temp) = forward_propagation_bias;
                all_forward_propagation.(temp) = forward_propagation_activation;

            end
            %weight for first layer update
        end
        switch output_function

            case 'sigmoid'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));
                y = 1./(1 + exp(-forward));

                err = -sum(sum(targets(n,1).*log(y) + (1-targets(n,1)).*log(1-y)));

                berror = berror + err;


            case 'softmax'

                forward = all_forward_propagation_bias.((strcat('f_prop',num2str(no_of_layers))))*all_weights.(strcat
                     ('weight',num2str(no_of_weights)));
                temp = exp(forward);
                y = temp./(sum(temp,2)*ones(1,no_of_output));

                err = -sum(targets(n,:).*log(y));
                berror = err + berror;

        end


    end

    switch learning
        case 'batch'
            %ndata is used because for batch error goes over the whole data
                eval_error = (berror / ndata);
        case 'online'
            %mb_value is used because the error is total over mb_value
                eval_error = total_error/mb_value;

    end

    fprintf('error: %d \n', eval_error);

    total_error = 0;
    iter = iter + 1;

    all_error = [all_error;eval_error];

    if eval_error < error_val
        break
    end

end

optimised_weights = all_weights;
plot_error = all_error;
```

readMNIST

```
\label{readMnist}
% readMNIST by Siddharth Hegde
%
% Description:
% Read digits and labels from raw MNIST data files
% File format as specified on http://yann.lecun.com/exdb/mnist/
% Note: The 4 pixel padding around the digits will be remove
%       Pixel values will be normalised to the [0...1] range
%
% Usage:
% [imgs labels] = readMNIST(imgFile, labelFile, readDigits, offset)
%
% Parameters:
% imgFile = name of the image file
% labelFile = name of the label file
% readDigits = number of digits to be read
% offset = skips the first offset number of digits before reading starts
%
% Returns:
% imgs = 20 x 20 x readDigits sized matrix of digits
% labels = readDigits x 1 matrix containing labels for each digit
%
function [imgs labels] = readMNIST(imgFile, labelFile, readDigits, offset)

    % Read digits
    fid = fopen(imgFile, 'r', 'b');
    header = fread(fid, 1, 'int32');
    if header ~= 2051
        error('Invalid image file header');
    end
    count = fread(fid, 1, 'int32');
    if count < readDigits+offset
        error('Trying to read too many digits');
    end

    h = fread(fid, 1, 'int32');
    w = fread(fid, 1, 'int32');

    if offset > 0
        fseek(fid, w*h*offset, 'cof');
    end

    imgs = zeros([h w readDigits]);

    for i=1:readDigits
        for y=1:h
            imgs(y,:,i) = fread(fid, w, 'uint8');
        end
    end

    fclose(fid);

    % Read digit labels
    fid = fopen(labelFile, 'r', 'b');
    header = fread(fid, 1, 'int32');
    if header ~= 2049
        error('Invalid label file header');
    end
    count = fread(fid, 1, 'int32');
    if count < readDigits+offset
        error('Trying to read too many digits');
    end

    if offset > 0
        fseek(fid, offset, 'cof');
    end
    labels = fread(fid, readDigits, 'uint8');
    fclose(fid);

    % Calc avg digit and count
    imgs = trimDigits(imgs, 4);
    imgs = normalizePixValue(imgs);
    %[avg num stddev] = getDigitStats(imgs, labels);

end

function digits = trimDigits(digitsIn, border)
    dSize = size(digitsIn);
    digits = zeros([dSize(1)-(border*2) dSize(2)-(border*2) dSize(3)]);
    for i=1:dSize(3)
        digits(:,:,i) = digitsIn(border+1:dSize(1)-border, border+1:dSize(2)-border, i);
    end
end

function digits = normalizePixValue(digits)
    digits = double(digits);
    for i=1:size(digits, 3)
        digits(:,:,i) = digits(:,:,i)./255.0;
    end
end
```

Script1

```
\label{Script1}
close all;clear;clc;

trainingSamples = 10000;
testingSamples = 2000;

[trainingData, trainingTarget] = GenerateNonLinear(trainingSamples);
[testingData, testingTarget] = GenerateNonLinear(testingSamples);

%uses the training data and training target to run matlab neural network

%with 1 hidden layer consisting of 4 hidden nodes
% matlab_4hid_1out;

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

%three_layer_network_train(epochs, input, targets, no_of_output, no_hidden_neuron, output_function,error_function,
     algorithm, learning,momentum, learning_rate, error_val, mini_batch, mb_value, distribution)

% %batch
% [first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'back-prop', 'batch', 1,0.2, 0.005,0, 200,0);

%online
% [first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'back-prop', 'online', 1,0.2, 0.005,0, 200,0);

%mini-batch online
[first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'back-prop', 'online', 1,0.2, 0.005,1, 200,0);

figure

plot(error);
title('error');
[ incorrect_prediction, result ] = three_layer_network_test(first_layer, second_layer, testingData', testingTarget',
     1,0);

%Parameter for the plots
x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

%creating random points within the boundary of the input training and
%testing data, for visualising the decision boundary
rand_points = [(x_max+10)*1.5*(rand(1,trainingSamples)-0.5); ((y_max)+10)*1.5*(rand(1,trainingSamples)-0.5)];
[ ith, classification_of_rand_points ] = three_layer_network_test(first_layer, second_layer,rand_points',
     testingTarget',1,1);

a1 = [];a2=[];c1=[];c2=[];c3=[];c4=[];c5=[];c6=[];

%Seperating the different classes for plots

%Random points
for n=1:size(rand_points,2)
if (classification_of_rand_points(n) == 1)
    a1 = [a1 rand_points(:,n)];
else
    a2 = [a2 rand_points(:,n)];
end
end

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
```

```
    c5 = [c5 trainingData(:,n)];
else
    c6 = [c6 trainingData(:,n)];
end
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end

%predicted result
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

%the plot
figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on
title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rand_points(1,:), rand_points(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on
title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

%scaling plots
linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

Script2

```
\label{Script2}
close all;clear;clc;

trainingSamples = 10000;
testingSamples = 2000;

% [trainingData, trainingTarget] = GenerateNonLinear2(trainingSamples);
% [testingData, testingTarget] = GenerateNonLinear2(testingSamples);

[trainingData, trainingTarget] = GenerateNonLinear3(trainingSamples);
[testingData, testingTarget] = GenerateNonLinear3(testingSamples);

%uses the training data and training target to run matlab neural network

%with 1 hidden layer consisting of 4 hidden nodes
% matlab_4hid_1out;

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

%three_layer_network_train(epochs, input, targets, no_of_output, no_hidden_neuron, output_function,error_function,
     algorithm, learning,momentum, learning_rate, error_val, mini_batch, mb_value)

%online
[first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'back-prop', 'online', 1,0.2, 0.005,0, 200);

%mini-batch online
%[first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'back-prop', 'online', 1,0.2, 0.005,1, 200);

figure

plot(error);
title('error');
[ incorrect_prediction, result ] = three_layer_network_test(first_layer, second_layer, testingData', testingTarget',
     1,0);

%Parameter for the plots
x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

%creating random points within the boundary of the input training and
%testing data, for visualising the decision boundary
rand_points = [(x_max+10)*1.5*(rand(1,trainingSamples)-0.5); ((y_max)+10)*1.5*(rand(1,trainingSamples)-0.5)];
[ ith, classification_of_rand_points ] = three_layer_network_test(first_layer, second_layer,rand_points',
     testingTarget',1,1);

a1 = [];a2=[];c1=[];c2=[];c3=[];c4=[];c5=[];c6=[];

%Seperating the different classes for plots

%Random points
for n=1:size(rand_points,2)
if (classification_of_rand_points(n) == 1)
    a1 = [a1 rand_points(:,n)];
else
    a2 = [a2 rand_points(:,n)];
end
end

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
    c5 = [c5 trainingData(:,n)];
```

```
else
    c6 = [c6 trainingData(:,n)];
end
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end

%predicted result
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

%the plot
figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on
title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rand_points(1,:), rand_points(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on
title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

%scaling plots
linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

Script3

```
\label{Script3}
close all;clear;clc;

trainingSamples = 1000;
testingSamples = 200;

[trainingData, trainingTarget] = XOR(trainingSamples);
[testingData, testingTarget] = XOR(testingSamples);
%
% [trainingData, trainingTarget] = GenerateNonLinear2(trainingSamples);
% [testingData, testingTarget] = GenerateNonLinear2(testingSamples);

% [trainingData, trainingTarget] = GenerateNonLinear3(trainingSamples);
% [testingData, testingTarget] = GenerateNonLinear3(testingSamples);

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

net = [20 20];

             %multiple_layer_network_train(epochs,input, targets, no_of_output, net_architecture, output_function,
                  error_function, algorithm, learning,momentum, learning_rate, error_val, mb_value, partial_con,
                  distribution)

[weight, error] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy s
     ', 'back-prop','online',1, 0.2, 0.005,200,0,0.1);

figure

plot(error);
title('error');

                            %multi_layer_network_test(weights, testingData, testingTarget,no_of_output, no_of_layers,
                                  plot)
[incorrect_prediction, result]  = multiple_layer_network_test(weight, testingData', testingTarget', 1,2,0);

x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

rd = [(x_max+100)*3*(rand(1,trainingSamples*150)-0.5); (y_max+100)*3*(rand(1,trainingSamples*150)-0.5)];
[ ith, result2 ] = multiple_layer_network_test(weight, rd', testingTarget', 1, 2,1);

a1 = [];
a2=[];
%test data
for n=1:size(rd,2)
if (result2(n) == 1)
    a1 = [a1 rd(:,n)];
else
    a2 = [a2 rd(:,n)];
end
end

c1 = [];
c2 = [];
c3 = [];
c4 = [];c5 = [];c6=[];

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
    c5 = [c5 trainingData(:,n)];
else
    c6 = [c6 trainingData(:,n)];
end
```

```
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end
%predicted result
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on

title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rd(1,:), rd(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on

title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

Script4

```
\label{Script4}
close all;clear;clc;

trainingSamples = 10000;
testingSamples = 2000;

[trainingData, trainingTarget] = GenerateNonLinear(trainingSamples);
[testingData, testingTarget] = GenerateNonLinear(testingSamples);

%uses the training data and training target to run matlab neural network

%with 1 hidden layer consisting of 4 hidden nodes
% matlab_4hid_1out;

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

%three_layer_network_train(epochs, input, targets, no_of_output, no_hidden_neuron, output_function,error_function,
     algorithm, learning,momentum, learning_rate, error_val, mini_batch, mb_value)

% %batch
% [first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'feedback_alignment', 'batch', 1,0.2, 0.005,0, 200,0);

%online
[first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,0, 200,0);

% %mini-batch online
% [first_layer, second_layer, error] = three_layer_network_train(80,trainingData',trainingTarget' ,1, 4, 'sigmoid', '
     cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,1, 200,0);

figure

plot(error);
title('error');
[ incorrect_prediction, result ] = three_layer_network_test(first_layer, second_layer, testingData', testingTarget',
     1,0);

%Parameter for the plots
x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

%creating random points within the boundary of the input training and
%testing data, for visualising the decision boundary
rand_points = [(x_max+10)*1.5*(rand(1,trainingSamples)-0.5); ((y_max)+10)*1.5*(rand(1,trainingSamples)-0.5)];
[ ith, classification_of_rand_points ] = three_layer_network_test(first_layer, second_layer,rand_points',
     testingTarget',1,1);

a1 = [];a2=[];c1=[];c2=[];c3=[];c4=[];c5=[];c6=[];

%Seperating the different classes for plots

%Random points
for n=1:size(rand_points,2)
if (classification_of_rand_points(n) == 1)
    a1 = [a1 rand_points(:,n)];
else
    a2 = [a2 rand_points(:,n)];
end
end

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
```

```
    c5 = [c5 trainingData(:,n)];
else
    c6 = [c6 trainingData(:,n)];
end
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end

%predicted result
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

%the plot
figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on
title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rand_points(1,:), rand_points(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on
title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

%scaling plots
linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

Script5

```
\label{Script5}
close all;clear;clc;

trainingSamples = 1000;
testingSamples = 200;

% [trainingData, trainingTarget] = XOR(trainingSamples);
% [testingData, testingTarget] = XOR(testingSamples);

[trainingData, trainingTarget] = GenerateNonLinear2(trainingSamples);
[testingData, testingTarget] = GenerateNonLinear2(testingSamples);

% [trainingData, trainingTarget] = GenerateNonLinear3(trainingSamples);
% [testingData, testingTarget] = GenerateNonLinear3(testingSamples);

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

net = [20 20];

             %multiple_layer_network_train(epochs,input, targets, no_of_output, net_architecture, output_function,
                  error_function, algorithm, learning,momentum, learning_rate, error_val, mb_value, partial_con,
                  distribution)

[weight, error] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy s
     ', 'feedback_alignment','online',1, 0.2, 0.005,200,0,0.1);

figure

plot(error);
title('error');

                            %multi_layer_network_test(weights, testingData, testingTarget,no_of_output, no_of_layers,
                                  plot)
[incorrect_prediction, result]  = multiple_layer_network_test(weight, testingData', testingTarget', 1,2,0);

x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

rd = [(x_max+100)*3*(rand(1,trainingSamples*150)-0.5); (y_max+100)*3*(rand(1,trainingSamples*150)-0.5)];
[ ith, result2 ] = multiple_layer_network_test(weight, rd', testingTarget', 1, 2,1);

a1 = [];
a2=[];
%test data
for n=1:size(rd,2)
if (result2(n) == 1)
    a1 = [a1 rd(:,n)];
else
    a2 = [a2 rd(:,n)];
end
end

c1 = [];
c2 = [];
c3 = [];
c4 = [];c5 = [];c6=[];

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
    c5 = [c5 trainingData(:,n)];
else
    c6 = [c6 trainingData(:,n)];
end
```

```
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end
%predicted result
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on

title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rd(1,:), rd(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on

title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

simple_network_test

```
\label{simpleNetworkTest}
function [ error, result] = simple_network_test(weight, testingData,testingTarget,find_boundary)
%find_boundary can be set to either 0 or 1, 1 disables the testingTargets,
%as it is used for generating a boundary of classification
ndata = size(testingData, 1);
bias_appended_input = [ones(ndata, 1), testingData];
weights = weight;

incorrect_pred = 0;

result = [];
for n= 1:ndata

    linear_combiner = bias_appended_input(n,:)*weights;

    a = sum(linear_combiner);

    if a < 0
        output = 0;
    else
        output = 1;
    end

    if find_boundary==0
        target = testingTarget(n,1);

        if target ~= output
            incorrect_pred = incorrect_pred + 1;
        end
    else

    end

    result = [result ; output];
end
if find_boundary==0
    error = incorrect_pred;
else
    error = 0; %redundant
end

end
```

simple_network_train

```
function [optimal_weight] = simple_network_train(input, targets, network, learning_rate)
% network: perceptron or the adaline

%weights initialised by random selection from a zero mean, unit variance
%isotropic Gaussian where the variance is scaled by the fan in of the
%hidden or output units
first_layer_weight = (randn(size(input, 2), 1)/sqrt(size(input, 2) + 1))';
bias_weight =  randn(1, 1)/sqrt(size(input, 2) + 1);

weights = [bias_weight' first_layer_weight]';

ndata = size(input, 1);

%append 1 to the data as the bias input value
bias_appended_input = [ones(ndata, 1), input];

total_error = 0;
error = 10;

% 0 error means convergence
while error > 0


    for n= 1:ndata

        switch network
            case 'perceptron'
                linear_combiner = bias_appended_input(n,:)*weights;

                %hard_limiter
                if linear_combiner > 0
                    output = 1;
                else
                    output = 0;
                end

                target = targets(n,1);

            case 'adaline'

                linear_combiner = bias_appended_input(n,:)*weights;

                %linear
                if linear_combiner > 0
                    output = 1;
                else
                    output = 0;
                end

                target = targets(n,1);

        end

        %if output and target are not a match, weights are updated
        if target ~= output

            switch network

                case 'perceptron'

                    delta_target = (targets(n,1) - output);
                    in = bias_appended_input(n,:)';
                    delta_weight_val = learning_rate*in*delta_target;

                    weights = weights + delta_weight_val;
                    total_error = total_error + delta_target;
                case 'adaline'

                    delta_target = (targets(n,1) - linear_combiner);
                    in = bias_appended_input(n,:)';
                    delta_weight_val = learning_rate*in*delta_target;

                    weights = weights + delta_weight_val;
                    total_error = total_error + delta_target;

            end

        end

    end

    error = total_error/ndata;

    fprintf('Error: %d \n', error);

    total_error = 0;

end

optimal_weight = weights;
```

test_simple_network

```
\label{testSimple}
close all;clc;clear;
%generates a linearly seperable dataset
trainingSamples = 10000;
testingSamples = 1000;
[trainingData, trainingTarget] = GenerateGaussianDataLS(trainingSamples);
[testingData, testingTarget] = GenerateGaussianDataLS(testingSamples);

%this removes the two
trainingTarget(2,:) = [];
testingTarget(2,:) = [];

%%%Perceptron
% %simple_network_train(input, targets, network, learning_rate)
% [optimised_weights] = simple_network_train(trainingData', trainingTarget','perceptron', 0.2);
% %simple_network_test(weight, testingData,testingTarget,find_boundary)
% [incorrect_predictions, result] = simple_network_test(optimised_weights, testingData',testingTarget',0);

%%%Adaline
% %simple_network_train(input, targets, network, learning_rate)
% [optimised_weights] = simple_network_train(trainingData', trainingTarget','adaline', 0.2);
% %simple_network_test(weight, testingData,testingTarget,find_boundary)
% [incorrect_predictions, result] = simple_network_test(optimised_weights, testingData',testingTarget',0);

%Parameter for the plots
x_max = max([max(trainingData(1,:)) max(testingData(1,:))]);
x_min = min([min(trainingData(1,:)) min(testingData(1,:))]);
y_max = max([max(trainingData(2,:)) max(testingData(2,:))]);
y_min = min([min(trainingData(2,:)) min(testingData(2,:))]);

%creating random points within the boundary of the input training and
%testing data, for visualising the decision boundary
rand_points = [(x_max+10)*1.5*(rand(1,trainingSamples)-0.5); ((y_max)+10)*1.5*(rand(1,trainingSamples)-0.5)];
[ ith, classification_of_rand_points ] = simple_network_test(optimised_weights, rand_points', testingTarget', 1);

a1 = [];a2=[];c1=[];c2=[];c3=[];c4=[];c5=[];c6=[];

%Seperating the different classes for plots

%Random points
for n=1:size(rand_points,2)
if (classification_of_rand_points(n) == 1)
    a1 = [a1 rand_points(:,n)];
else
    a2 = [a2 rand_points(:,n)];
end
end

%training data
for n=1:size(trainingData,2)
if (trainingTarget(n) == 1)
    c5 = [c5 trainingData(:,n)];
else
    c6 = [c6 trainingData(:,n)];
end
end

%test data
for n=1:size(testingData,2)
if (testingTarget(n) == 1)
    c1 = [c1 testingData(:,n)];
else
    c2 = [c2 testingData(:,n)];
end
end

%predicted result
```

```
for n=1:size(testingData,2)
if (result(n) == 1)
    c3 = [c3 testingData(:,n)];
else
    c4 = [c4 testingData(:,n)];
end
end

%the plot
figure

h(1) = subplot(2,2,1);
hold on
title('Desired Result');
plot(c1(1,:), c1(2,:), 'ro');
plot(c2(1,:), c2(2,:), 'b+');

h(2) = subplot(2,2,2);
hold on
title('Predicted Result');
plot(c3(1,:), c3(2,:), 'm*');
plot(c4(1,:), c4(2,:), 'gx');

h(3) = subplot(2,2,3);
title('Decision Boundary');
hold on
plot(rand_points(1,:), rand_points(2,:),'o');
plot(a2(1,:), a2(2,:), 'gx');
plot(a1(1,:), a1(2,:), 'm*');

h(4) = subplot(2,2,4);
hold on
title('Training Data');
plot(c5(1,:), c5(2,:), 'ro');
plot(c6(1,:), c6(2,:), 'b+');

%scaling plots
linkaxes(h)
ylim([y_min-2 y_max+2])
xlim([x_min-2 x_max+2])
```

Testing_Mnist

```
\label{testingMnist}
close all;clear;clc;

%loads Mnist training and testing Dataset
loadMnist;

%1 hot representation
size_train = size(TrainLabels,1);
size_test = size(TestLabels,1);

ones_train = ones(size_train);
ones_test = ones(size_test);

ones_train = ones_train(:,1);
ones_test = ones_test(:,1);

change_train_label = TrainLabels + ones_train;
training_label = dummyvar(change_train_label);

change_test_label = TestLabels + ones_test;
testing_label = dummyvar(change_test_label);

net = [10 10];

%multiple_layer_network_train(epochs,input, targets, no_of_output, net_architecture, output_function,error_function,
     algorithm, learning,momentum, learning_rate, error_val, mb_value, partial_con,distribution)

%feedback alignment
[weight, error] = multiple_layer_network_train(200,TrainData,training_label, 10, net, 'softmax', 'cross entropy m', '
     feedback_alignment','online',1, 0.2, 0.005,300,0,1);

%back prop
%[weight, error] = multiple_layer_network_train(200,TrainData,training_label, 10, net, 'softmax', 'cross entropy m',
     'back-prop','online',1, 0.2, 0.005,300,0,1);

figure

plot(error);
title('error');

%[error, result] = multiple_layer_network_test(weights, testingData, testingTarget,no_of_output, no_of_layers, plot)

[incorrect_prediction, result]  = multiple_layer_network_test(weight, TestData, testing_label, 10,2,0);
```

three_layer_network_test

```
\label{threeLayerTest}
function [ error, result ] = three_layer_network_test(weight1, weight2, testingData, testingTarget,no_of_output, plot
     )
%forward propagation is done in the same way
ndata = size(testingData, 1);
bias_appended_input = [ones(ndata, 1), testingData];
weights_first_layer = weight1;
weights_second_layer = weight2;
incorrect_pred = 0;

if no_of_output == 1
    out = 'single';
else
    out = 'multi';
end

result = [];
%plotting the prediction, plot=1 does not compare with the target used to draw the boundary, plot=0
%compares with the targets

if (plot == 1)
    for n= 1:ndata

        error = incorrect_pred;

        %forward propagation
        forward_propagation_hidden = bias_appended_input(n,:)*weights_first_layer;
        % sigmoid function
        forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
        %add bias
        forward_propagation_bias = [ones(1,1), forward_propagation_activation];

        switch out

            case 'single'

                forward = forward_propagation_bias*weights_second_layer;

                y = 1./(1 + exp(-forward));
                %at the output of the network, a classification is given
                if y < 0.5
                    output = 0;
                else
                    output = 1;
                end

            case 'multi'

                %forward_propagation = bias_appended_input(n,:)*weights;
                forward = forward_propagation_bias*weights_second_layer;


                temp = exp(forward);

                y = temp./(sum(temp,2)*ones(1,no_of_output));

                %the index of the maximum is used as the output
                [t_output ind_t_output] = max(y);

                %same for multi-output layer, where a classification is
                %given
                output = ind_t_output;

        end
        result = [result ; output];

    end
```

```
else
    for n= 1:ndata
        %forward propagation
        forward_propagation_hidden = bias_appended_input(n,:)*weights_first_layer;
        % sigmoid function
        forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
        %add bias
        forward_propagation_bias = [ones(1,1), forward_propagation_activation];

        switch out

            case 'single'

                forward = forward_propagation_bias*weights_second_layer;

                y = 1./(1 + exp(-forward));

                target = testingTarget(n,1);

                if y < 0.5
                    output = 0;
                else
                    output = 1;
                end

            case 'multi'

                %forward_propagation = bias_appended_input(n,:)*weights;
                forward = forward_propagation_bias*weights_second_layer;

                temp = exp(forward);

                %y(y<realmin) = realmin;
                y = temp./(sum(temp,2)*ones(1,no_of_output));

                %%the index is used to check if they belong to the same
                %%class
                [t_output ind_t_output] = max(y);

                output = ind_t_output;

                [t indTarget] = max(testingTarget(n,:));

                target = indTarget;

        end

        if target ~= output
            %total the number of incorrect_prediction
            incorrect_pred = incorrect_pred + 1;

        end

        result = [result ; output];
    end

    error = incorrect_pred;
end
end
```

three_layer_network_train

```
\label{threeLayerTrain}
function [first_layer, second_layer, plot_error] = three_layer_network_train(epochs, input, targets, no_of_output,
     no_hidden_neuron, output_function,error_function, algorithm, learning,momentum, learning_rate, error_val,
     mini_batch, mb_value,distribution)
%Input----1Hidden------Output

%plot_error: acumulates all the error from each epoch
%epochs: number of iterations
%output_function: sigmoid or softmax depending on the number of outputs
%error_function: cross_entropy for single or multiple output
%algorithm: back-prop or random feedback
ndata = size(input, 1);

%sigmoid is used for 2 class problem, while softmax is used for >2 class
%problem
if no_of_output == 1
    out_funct = 'single';
else
    out_funct = 'multi';
end

%Initialising weights for the two layers
first_layer_weight = (rand(size(input, 2), no_hidden_neuron)/sqrt(size(input, 2) + 1))';
first_layer_bias =  rand(1, no_hidden_neuron)/sqrt(size(input, 2) + 1);

second_layer_weight = (rand(no_hidden_neuron, no_of_output)/sqrt(no_hidden_neuron + 1))';
second_layer_bias = rand(1, no_of_output)/sqrt(no_hidden_neuron + 1);

weights_first_layer = [first_layer_bias' first_layer_weight]';
weights_second_layer = [second_layer_bias' second_layer_weight]';

%the distribution of random weights for Feedback Alignment set to uniform if distribution=0
%else, uses the value of the distribution as the variance for uniformly distributed weights
if distribution == 0
    if no_of_output == 1
        random_weight = -.5+(.5+.5)*rand(1, no_hidden_neuron);

    else
        [a,b] = size(second_layer_weight);
        random_weight = -.5+(.5+.5)*rand(b,a);
    end
else
    if no_of_output == 1
        mean = 0;
        variance = distribution;
        random_weight = mean + variance * randn(1,no_hidden_neuron);


    else
        [a,b] = size(second_layer_weight);
        mean = 0;
        variance = distribution;
        random_weight = mean + variance * randn(b,a);
    end


end
%used to total the gradients for batch gradient descent
[a b] = size(weights_first_layer);
total_gradient_first = zeros(a,b);
[a b] = size(weights_second_layer);
total_gradient_second = zeros(a,b);

%bias value added
bias_appended_input = [ones(ndata, 1), input];

%total the errors
```

```
total_error = 0;
%store all
all_error = [];
batch = 1;

%stopping criteria
while batch < epochs

    %choose between mini-batch or batch, 1 for mini-batch, 0 for batch or
    %online
    switch mini_batch
        case 0
            for n=1:ndata;
                %forward propagation
                forward_propagation_hidden = bias_appended_input(n,:)*weights_first_layer;
                % activation function (sigmoid)
                forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
                %add bias
                forward_propagation_bias = [ones(1,1), forward_propagation_activation];

                switch output_function
                    case 'sigmoid'
                        %forward propagation
                        forward = forward_propagation_bias*weights_second_layer;
                        %activation function
                        y = 1./(1 + exp(-forward));

                    case 'softmax'
                        forward = forward_propagation_bias*weights_second_layer;
                        temp = exp(forward);
                        y = temp./(sum(temp,2)*ones(1,no_of_output));

                end
                %this is specified aboved..
                switch out_funct
                    case 'single'
                        switch algorithm
                            case 'back-prop'
                                % back propagation is used to calculate the
                                % error

                                %second layer
                                delta_target = -(targets(n,1) - y);
                                sig_der_2 = y*(1-y);
                                %this value is used to update the weights
                                %in the second layer
                                delta_weight_second= delta_target*(sig_der_2)*forward_propagation_bias;

                                %first layer
                                sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);

                                %a bias valued added previously needs to be
                                %removed here
                                bias_out = weights_second_layer;
                                bias_out(1,:) = [];

                                %calculating the error in steps
                                delta_weight_first_1 = (delta_target*sig_der_2* bias_out);
                                delta_weight_first_2 = delta_weight_first_1 .*sig_der_1';
                                %this value is sed to update the weights in
                                %the first layer
                                delta_weight_first = bias_appended_input(n,:)'*delta_weight_first_2';

                                %mode of learning
                                switch learning
                                    case 'batch'
                                        %acumulating the errors
                                        total_gradient_first = delta_weight_first + total_gradient_first;
                                        total_gradient_second = total_gradient_second + delta_weight_second';
                                    case 'online'
                                        %second layer update
                                        weights_second_layer = weights_second_layer*momentum - (learning_rate*
                                             delta_weight_second');

                                        %first layer update
                                        weights_first_layer = weights_first_layer*momentum - (learning_rate*
                                             delta_weight_first);
                                end

                            case 'feedback_alignment'

                                %second layer
                                %similar to back-prop for this layer
                                delta_target = -(targets(n) - y);
                                sig_der_2 = y*(1-y);
                                delta_weight_second= delta_target*(sig_der_2)*forward_propagation_bias;
                                weights_second_layer = weights_second_layer - (learning_rate*delta_weight_second');

                                %first layer
                                sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);
                                delta_weight_first_1 = (delta_target*(bias_appended_input(n,:))');
                                %Instead of the transpose weight in the
                                %network, a random weight matrix is used
                                %instead
                                delta_weight_first_2 = (random_weight.*sig_der_1);
                                delta_weight_first = delta_weight_first_1 * delta_weight_first_2;

                                weights_first_layer = weights_first_layer - (learning_rate * delta_weight_first);


                        end
                    case 'multi'

                        switch algorithm

                            case 'back-prop'
                                % back propagation performed same way as
                                % for single output
                                %second layer
                                t = targets(n,:);
                                delta_target = y - t;
                                sig_der_2 = y.*(1-y);
                                delta_weight_second= forward_propagation_bias'* (delta_target.*(sig_der_2));

                                %first layer
                                bias_out = weights_second_layer;
                                bias_out(1,:) = [];

                                sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);

                                delta_weight_first_1 = bias_out*(delta_target.*sig_der_2)';

                                delta_weight_first = (bias_appended_input(n,:))'*(sig_der_1.*delta_weight_first_1');

                                switch learning
                                    case 'batch'
                                        total_gradient_first = delta_weight_first + total_gradient_first;
                                        total_gradient_second = total_gradient_second + delta_weight_second;
                                    case 'online'
                                        %second layer update
                                        weights_second_layer = weights_second_layer - (learning_rate*
                                             delta_weight_second);

                                        %first layer update
                                        weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first
                                             );

                                end
```

```
                    case 'feedback_alignment'
                        %updates are done in the same way as above
                        %second layer
                        delta_target = -(targets(n,:) - y);
                        sig_der_2 = y.*(1-y);
                        delta_weight_second= forward_propagation_bias'* (delta_target.*(sig_der_2));

                        weights_second_layer = weights_second_layer - (learning_rate*delta_weight_second);

                        %first layer
                        sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);


                        delta_weight_first_1 = random_weight*(delta_target)';

                        delta_weight_first = (bias_appended_input(n,:))'*(sig_der_1.*delta_weight_first_1');

                        weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first);

                        switch learning
                            case 'batch'
                                %some problems here..
                                total_gradient_first = delta_weight_first + total_gradient_first;

                                total_gradient_second = total_gradient_second + delta_weight_second';
                            case 'online'
                                %second layer update
                                weights_second_layer = weights_second_layer - (learning_rate*
                                     delta_weight_second);

                                %first layer update
                                weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first
                                     );

                        end


                end
            end

            switch error_function
                %cross entropy error function for single and multi outputs
                case 'cross entropy s'

                    err = -sum(sum(targets(n,1).*log(y) + (1-targets(n,1)).*log(1-y)));

                    total_error = err + total_error;

                case 'cross entropy m'
                    err = -sum(targets(n,:).*log(y));
                    total_error = err + total_error;
            end

        end

    case 1
        %For mini-match, mb_value random ordered inputs are chosen
        random = randi([1 ndata],1,mb_value);
        for m=1:mb_value;
            %mini batch gradient descent
            n = random(m);
            %forward propagation
            forward_propagation_hidden = bias_appended_input(n,:)*weights_first_layer;
            % sigmoid function
            forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
            %add bias
            forward_propagation_bias = [ones(1,1), forward_propagation_activation];
```

```
switch output_function

    case 'sigmoid'

        forward = forward_propagation_bias*weights_second_layer;
        y = 1./(1 + exp(-forward));

    case 'softmax'

        forward = forward_propagation_bias*weights_second_layer;
        temp = exp(forward);
        y = temp./(sum(temp,2)*ones(1,no_of_output));

end

switch out_funct
    case 'single'

        switch algorithm

            case 'back-prop'
                % back propagation, same as above
                %second layer
                delta_target = -(targets(n,1) - y);

                sig_der_2 = y*(1-y);
                delta_weight_second= delta_target*(sig_der_2)*forward_propagation_bias;

                %first layer
                sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);

                bias_out = weights_second_layer;
                bias_out(1,:) = [];
                delta_weight_first_1 = (delta_target*sig_der_2* bias_out );
                delta_weight_first_2 = delta_weight_first_1 .*sig_der_1';

                %bias_appended_input(n,:)'
                delta_weight_first = bias_appended_input(n,:)'*delta_weight_first_2';

                switch learning
                    case 'batch'
                        total_gradient_first = delta_weight_first + total_gradient_first;
                        total_gradient_second = total_gradient_second + delta_weight_second';
                    case 'online'

                        %second layer update
                        weights_second_layer = weights_second_layer*momentum - (learning_rate*
                             delta_weight_second');

                        %first layer update
                        weights_first_layer = weights_first_layer*momentum - (learning_rate*
                             delta_weight_first);
                end

            case 'feedback_alignment'

                %second layer
                delta_target = -(targets(n) - y);
                sig_der_2 = y*(1-y);
                delta_weight_second= delta_target*(sig_der_2)*forward_propagation_bias;
                weights_second_layer = weights_second_layer - (learning_rate*delta_weight_second');

                %first layer
                sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);
                delta_weight_first_1 = (delta_target*(bias_appended_input(n,:))');
                delta_weight_first_2 = (random_weight.*sig_der_1);
                delta_weight_first = delta_weight_first_1 * delta_weight_first_2;

                weights_first_layer = weights_first_layer - (learning_rate * delta_weight_first);
    end
case 'multi'

    switch algorithm

        case 'back-prop'
            % back propagation
            %second layer
            t = targets(n,:);
            delta_target = y - t;
            sig_der_2 = y.*(1-y);
            delta_weight_second= forward_propagation_bias'* (delta_target.*(sig_der_2));

            %first layer
            bias_out = weights_second_layer;
            bias_out(1,:) = [];

            sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);

            delta_weight_first_1 = bias_out*(delta_target.*sig_der_2)';

            delta_weight_first = (bias_appended_input(n,:))'*(sig_der_1.*delta_weight_first_1');

            switch learning
                case 'batch'
                    total_gradient_first = delta_weight_first + total_gradient_first;
                    total_gradient_second = total_gradient_second + delta_weight_second;
                case 'online'
                    %second layer update
                    weights_second_layer = weights_second_layer - (learning_rate*
                         delta_weight_second);

                    %first layer update
                    weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first
                         );

            end

        case 'feedback_alignment'

            %second layer
            delta_target = -(targets(n,:) - y);
            sig_der_2 = y.*(1-y);
            delta_weight_second= forward_propagation_bias'* (delta_target.*(sig_der_2));

            weights_second_layer = weights_second_layer - (learning_rate*delta_weight_second);

            %first layer
            sig_der_1 = forward_propagation_activation.*(1-forward_propagation_activation);

            % random
            delta_weight_first_1 = random_weight*(delta_target)';

            delta_weight_first = (bias_appended_input(n,:))'*(sig_der_1.*delta_weight_first_1');

            weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first);

            switch learning
                case 'batch'
                    %some problems here..
                    total_gradient_first = delta_weight_first + total_gradient_first;

                    total_gradient_second = total_gradient_second + delta_weight_second';
                case 'online'
                    %second layer update
                    weights_second_layer = weights_second_layer - (learning_rate*
                         delta_weight_second);
```

```
                                    %first layer update
                                    weights_first_layer = weights_first_layer - (learning_rate*delta_weight_first
                                         );
                            end

                    end
            end
            switch error_function

                %cross entropy error function

                case 'cross entropy s'

                    err = -sum(sum(targets(n,1).*log(y) + (1-targets(n,1)).*log(1-y)));

                    total_error = err + total_error;

                case 'cross entropy m'
                    err = -sum(targets(n,:).*log(y));
                    total_error = err + total_error;
            end

        end

end

if strcmp(learning,'batch') == 1
    avg_delta_weight_first = total_gradient_first/ndata;
    avg_delta_weight_second = total_gradient_second/ndata;
    %second layer update
    weights_second_layer = weights_second_layer*momentum - (learning_rate*avg_delta_weight_second);

    %first layer update
    weights_first_layer = weights_first_layer*momentum - (learning_rate*avg_delta_weight_first);
end

%batch error
%if batch was selected, mean error is calculated using this updated weight
%on the whole dataset
berror = 0;
for nn=1:ndata

    forward_propagation_hidden = bias_appended_input(nn,:)*weights_first_layer;
    forward_propagation_activation = 1./(1 + exp(-forward_propagation_hidden));
    forward_propagation_bias = [ones(1,1), forward_propagation_activation];

    forward = forward_propagation_bias*weights_second_layer;
    y = 1./(1 + exp(-forward));

    switch error_function

        case 'cross entropy s'
            %cross entropy error function
            err1 = -sum(sum(targets(n,1).*log(y) + (1-targets(n,1)).*log(1-y)));

            berror = err1 + berror;

        case 'cross entropy m'
            err1 = -sum(targets(n,:).*log(y));
            berror = err1 + berror;
    end

end

%calculating the error
switch learning
    case 'batch'
        eval_error = berror/ndata;
        case 'online'

            if mini_batch == 1
                eval_error = total_error/mb_value;
            else
                eval_error = (total_error/ndata);
            end
    end

    fprintf('error: %d \n', eval_error);

    total_error = 0;
    batch = batch + 1;

    all_error = [all_error;eval_error];

    %stopping criteria
    if eval_error < error_val
        break
    end
end

first_layer = weights_first_layer;
second_layer = weights_second_layer;
plot_error = all_error;
```

XOR

```
\label{XOR}
function [data targetVec] = GenerateNonLinear2(samples)
% Generates a two-class dataset in matrix 'data' with 2D data vectors and
% also indicates each pattern class in the matrix 'targetVec'

% Function first generates 'samples' 2-d data vectors of class 1 by independently
% drawing each dimension from a  Gaussian distribution.
% class 1 patterns have an associated output label vector in matrix targetVec [1 0]'
% The  mean and covariance for class 1 are
% Mean1 = [4; -1;];
% Sigma1 = [0.6 .8; .8 7];

% Function then generates 'samples' 2-d data vectors of class 2 by independently
% drawing each dimension from a  Gaussian distribution.
% class 2 patterns have an associated output label vector in matrix targetVec[0 1]'
% The  mean and covariance for class 2 are
% Mean2 = [-1; 1;];
% Sigma2 = [0.7 .8; .8 9];

% the two sets of class1 and class2 data are then concatenated into the
% output  matrix 'data'
% similarly the label vectors are concatenated onto the output matrix
% 'targetVec'

% Both output matrices therefore have size = (2, 2 * samples)

% see example usage at end of function
% set demo = 1 to run this plot example
%demo = 1; % demo off
demo = 1; % demo on

% Ian Howard, University of Plymouth, December 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Mean1 = [-13; -6;];
Sigma1 = [0.1 0; -4 1];


% set mean and covariance for class 1
Mean2 = [-5; 20;];  %top left   1
Mean3 = [5; 25;];   %top right   2
Mean4 = [-2; -8;];  % bottom left  2
Mean5 = [7; 2;];    %bottom right   1

Sigma2 = [0.7 0; .8 9];





%og
% Mean2 = [-5; 0;];
% Sigma2 = [0.7 .8; .8 9];
%
% Mean3 = [3; 0;];
% Sigma3 = [0.9 -.8; 0.7 7];

% generate Gaussian distibuted dataset for class 0
data1 =  (chol(Sigma1) * randn(2, samples) + repmat(Mean1,1,samples))+14;
% generate target vectors for class 0
label1 = repmat([1; 0] , 1, samples);

disp(cov(data1'));

% generate Gaussian distibuted datasetn for class 1
data2 =  chol(Sigma2) * randn(2, samples) + repmat(Mean2,1,samples);
% generate target vectors for class 1
label2 = repmat([0; 1] , 1, samples);
```

```
disp(cov(data2'));

data3 =  chol(Sigma2) * randn(2, samples) + repmat(Mean3,1,samples);
% generate target vectors for class 1
label3 = repmat([1; 0] , 1, samples);
disp(cov(data3'));

data4 =  chol(Sigma2) * randn(2, samples) + repmat(Mean4,1,samples);
% generate target vectors for class 1
label4 = repmat([1; 0] , 1, samples);
disp(cov(data3'));

data5 =  chol(Sigma2) * randn(2, samples) + repmat(Mean5,1,samples);
% generate target vectors for class 1
label5 = repmat([0; 1] , 1, samples);
disp(cov(data3'));


%put it together

% concatenate class0 and class1 datasets into single dataset
% data = [data1 data2];
% targetVec = [label1 label2];
%

data = [data2 data3 data4 data5];
targetVec = [label2 label3 label4 label5];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(demo)
    % example usage
    % vectorized example to extract all class 1 patterns
    % examine first dimension which is 1 for class 1
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 1);
    c1data = data(:,fidx);

    % vectorized example to extract all class 2 patterns
    % examine first dimension which is 0 for class 2
    % you could also use a loop if you wish
    fidx = find(targetVec(1,:) == 0);
    c2data = data(:,fidx);


%     fidx = find(targetVec2(1,:) == 3);
%     c3data = data(:,fidx);


    % now plot separated classes  on a figure
    figure
    hold on
   plot(c1data(1,:), c1data(2,:), 'ro');
    plot(c2data(1,:), c2data(2,:), 'b+');

    %plot(c3data(1,:), c3data(2,:), 'c*');

    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class1 patterns in red and class2 patterns in blue');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Normal_dis_plot

```
close all;clear;clc;

trainingSamples = 1000;
testingSamples = 200;

% [trainingData, trainingTarget] = GenerateNonLinear(trainingSamples);
% [testingData, testingTarget] = GenerateNonLinear(testingSamples);

%[trainingData, trainingTarget] = GenerateNonLinear2(trainingSamples);
%[testingData, testingTarget] = GenerateNonLinear2(testingSamples);

[trainingData, trainingTarget] = GenerateNonLinear3(trainingSamples);
[testingData, testingTarget] = GenerateNonLinear3(testingSamples);

% [trainingData, trainingTarget] = XOR(trainingSamples);
% [testingData, testingTarget] = XOR(testingSamples);

trainingTarget(2,:) = [];
testingTarget(2,:) = [];

%three_layer_network_train(epochs, input, targets, no_of_output, no_hidden_neuron, output_function,error_function,
     algorithm, learning,momentum, learning_rate, error_val, mini_batch, mb_value, distribution)

%three_layer
% [first_layer, second_layer, error1] = three_layer_network_train(40,trainingData',trainingTarget' ,1, 4, 'sigmoid',
     'cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,1, 200,0.2);
% [first_layer, second_layer, error2] = three_layer_network_train(40,trainingData',trainingTarget' ,1, 4, 'sigmoid',
     'cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,1, 200,0.5);
% [first_layer, second_layer, error3] = three_layer_network_train(40,trainingData',trainingTarget' ,1, 4, 'sigmoid',
     'cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,1, 200,0.7);
% [first_layer, second_layer, error4] = three_layer_network_train(40,trainingData',trainingTarget' ,1, 4, 'sigmoid',
     'cross entropy s', 'feedback_alignment', 'online', 1,0.2, 0.005,1, 200,0.9);

%multiple_layer
net = [20 20];

             %multiple_layer_network_train(epochs,input, targets, no_of_output, net_architecture, output_function,
                  error_function, algorithm, learning,momentum, learning_rate, error_val, mb_value, partial_con,
                  distribution)

[weight, error1] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy
     s', 'feedback_alignment','online',1, 0.2, 0.005,200,0,0.2);
[weight, error2] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy
     s', 'feedback_alignment','online',1, 0.2, 0.005,200,0,0.5);
[weight, error3] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy
     s', 'feedback_alignment','online',1, 0.2, 0.005,200,0,0.7);
[weight, error4] = multiple_layer_network_train(80,trainingData',trainingTarget' , 1, net, 'sigmoid', 'cross entropy
     s', 'feedback_alignment','online',1, 0.2, 0.005,200,0,0.9);



figure
x=1:1:65;
plot(x,error1',x,error2',x,error3',x,error4');
title('error');
legend('var = 0.2','var = 0.5','var = 0.7','var = 0.9');
```