Back-propagation for Non-linear Classification

Viraj Savaliya
Electrical and Computer Engineering
Arizona State University
Tempe, Arizona, USA
Viraj.Savaliya@asu.edu
ASU ID : 1217678787

***Abstract*—The goal of this report is to solve a non-linear classification problem using back-propagation algorithm and use two different activation functions and compare the results of both set of configurations in terms of average accuracy obtained from 10 runs with each time randomly initializing weights of the network. Also, the performance of the network for both the configurations is compared and shown in the report.**

Keywords—back propagation, neural network, activation function, non-linear classification, multiplayer perceptron

I. INTRODUCTION

In the current decade is known as the decade for Machine Learning. In every sector there is tons and tons of data which can be processed and used for getting valuable information or making useful predictions about the foreseeable future values. Machine learning has caught attention in every sector like engineering, business, trading, medicine, energy, agriculture, etc. Having powerful computers available nowadays to process very complex and large amount of computations aids to the demand and resource requirement of Machine Learning. Data Science and Machine learning are driving all the sectors in their development and have become an essential part of the industry. Having the knowledge and understanding the basics of the computations behind the Machine learning models is important before using the tools and to progress in the field. Neural Networks or Artificial Neural Networks are computing systems for machine learning which perform the tasks by analyzing and learning from the training examples.

Neural network consists of multiple layers which are termed as input layer, hidden layer, and output layer. The input layer is basically the input parameters you want your model to train to from the dataset. The output layer is the output values obtained after doing all the necessary computations through the entire network. However, the entire neural network where all the major computations take place are the hidden layers. These layers are not visible or are not in direct use of the dataset inputs and the outputs but are driven by weights associated with it for the inputs given from the input layer. Each layer consists of nodes or neurons which are the smallest or basic computing unit of the entire network. These neurons perform simple task like multiplication or addition and take into consideration weights associated with the inputs and biases and then gives out some output which is then passed on to the next layer. There can be multiple neurons in each layer depending on the requirements of the model. The number of neurons in input layer are equal to the number of input parameters from the dataset. The number of neurons in output layer is equal to the number of outputs desired to be obtained for the input given and the number of neurons in each hidden layer is decided as the model requirements for learning. A simple neural network with one hidden layer can be seen in Fig 1. The weights associated with each connection are updated over number of iterations till the point when the loss or the difference of the expected output and the output obtained from the network becomes minimum. ANNs are used for a number of applications related to function approximation, regression, data processing, sequence recognition, decision making, etc.

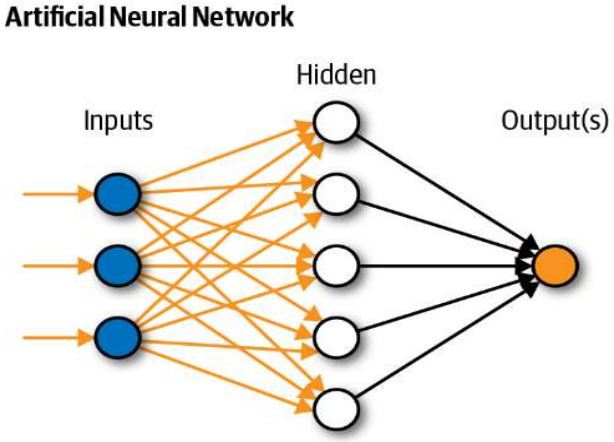

Fig 1. General representation of Artificial Neural Network

II. BACK PROPAGATION ALGORITHM

Back propagation algorithm is used for supervised learning in multilayer ANNs. The principle used in the algorithm is to model a given function or to learn the pattern of the dataset by modifying the weights of all inputs in each layer so that we predict the desire output at the end of the network. This training of the weights is supervised learning as the output layer is known to what is expected and the weights are adjusted accordingly. The weights adjustments are calculated based on the error obtained at the end of each iteration which is a feedback to the system to update its current state. The structure of the network is predefined, and the number of hidden layers and the size of each layer is fixed at the start and it is fully connected. A standard structure is of just one hidden layer with one input and one output layer. Back propagation algorithm can be used for both classification and regression problems. The algorithm can be simply divided into 5 parts namely initialization of network, forward propagation, back propagation of error, updating weights and then prediction. Each of these steps are explained in detail in the following sections. For understanding the algorithm let's take a structure of ANN as shown in Fig 2.

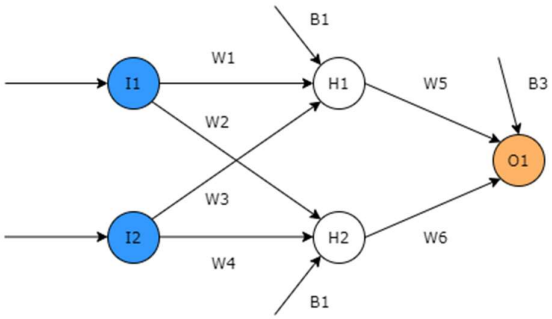

Fig 2. Structure of ANN for explanation

A. Network Initialization

The network consists of two neurons in the input layer and hidden layer each and one neuron in the output layer. Neurons in the hidden and output layer have bias associated to it of value equal to 1. Each neuron in a layer is connected to every neuron of the next layer and hence forms a fully connected network. Now the weights in the network are small randomly initialized numbers at the start of the algorithm. Now, network is initialized, and we can begin the first iteration by taking in the first set of inputs.

B. Forward Propagation

In forward propagation, every neuron is activated, and some value is transferred based on the computation to the next layer. First the value obtained at the hidden layer neurons after passing the inputs with the associated weights can be represented as follows,

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1 \quad \text{- (1)}$$

The above equation is for the hidden neuron H_1 and similarly the computed values at every other hidden neuron can be obtained. The equation can be represented in simple terms as,

$$H_i = B_i + \sum_{n=1}^{n\_inputs} W_i * I_i \quad \text{- (2)}$$

Now, once the value at the neuron is computed, we can apply the activation or thresholding function to normalize the value and make sure the output value remains within a specific range and does not go out of bound. The activation function or also called a transfer function is used to map an output in a specific range in the hidden layer and the output layer. The activation function can be either linear or non-linear and is selected based on the application and nature of the data the model needs to train on. For now, let us take a sigmoid activation function which has its equation as follows,

$$\emptyset\,(z) = \frac{1}{1+e^{-z}} \quad \text{- (3)}$$

Sigmoid transfer function is most commonly used as it predicts a probability as an output in the range of 0 to 1. The other most common types of transfer functions used are hyperbolic tangent function and ReLU (Rectified Linear Unit) function. The value obtained in eqn. (1) is then passed through the sigmoid transfer function given in eqn. (3) and the output of the transfer function will be the output value of a hidden neuron. Similarly, the output of every neuron is calculated and then given on to the next layer which is the output layer in our case. The output neuron also follows the same procedure and gives out an output which will be the final output of the network. This output is then compared against the actual expected output and an error value is also given out by the output layer. The error is calculated using the following formula,

$$E_{total} = 1/2 * \sum(target - output)^2 \quad \text{- (4)}$$

In the first iteration as there is no learning or updating of weights involved the output value are just considered as some random output. In the later steps when the weights are updated the network starts learning and then the forward propagation will give out output more close to the desired ones as the network keeps learning the pattern of the data provided to it.

C. Backward Propagation of Error

The forward propagation is followed by a backward propagation of the error value which is basically like a feedback given to the network to learn the pattern or to update the weights in the network. First, as we know the total error obtained in the forward pass then the effect of the error due to each weight is calculated and accordingly the weights are updated in the next step so that the error keeps on reducing in each iteration.

Now, let's take the weight w_5 and find out how much the weight affects the total error value obtained. This can be represented as $\frac{\partial E_{total}}{\partial w_5}$. To calculate the effect of w_5, we will need the partial derivative of E_{total} with respect to w_5, which is given by,

$$\frac{\partial E_{tot}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5} \quad \text{- (5)}$$

Now the value of each term after finding the partial derivative is given by,

$$\frac{\partial E_{total}}{\partial out_{o1}} = out_{o1} - target$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1} * (1 - out_{o1})$$

$$\frac{\partial net_{o1}}{\partial w_5} = out_{h1} \quad \text{- (6)}$$

After multiplying all the above partial derivatives, we get the value of eqn. (5). which is the effect of error due to w_5. Similarly, the same procedure is followed for all the weights and we get the value of how much each weight affects or needs to be changed to reduce the error. In the above partial derivative, the value of the second term i.e. $\frac{\partial out_{o1}}{\partial net_{o1}}$ is basically the derivative of the transfer function used. As we are using sigmoid function the derivative looks like $z * (1 - z)$. After calculating the effect of error due to all weights in the network, these values can be used in the next step of the algorithm.

D. Learning Weights

This step is where the network is trained or where the weights in the network are updated in order to minimize the error and to learn the pattern of the dataset provided to the

network. The weights in the network are updated using the following equation,

$$w_i = w_i + \alpha * \frac{\partial E_{total}}{\partial w_i} \quad \text{- (7)}$$

In the above eqn. (7), α is the learning rate which is a small number predefined at the start of the algorithm in order to converge the error at a minima. The learning rate can be changed over the number of iterations which is basically adaptive learning done when the predefined learning rate takes a very long time to converge. The change of learning rate can be done after specific number of iterations. After using the eqn. (7) and updating all the weights as per the partial derivative value of effect of total error due to a weight and the learning rate, all the weights in the network can be updated and the same process of forward propagation using the new weights can be repeated followed by backward propagation of error for a fixed number of epochs for every input so that the network is trained to the input with minimum error obtained in the output value. The same process of iterating for a fixed number of epochs is carried on for all the input values in the dataset in order to find the pattern and learn the entire dataset. After the entire procedure is completed over the dataset, the model is trained and we get a set of values for all the weights in the network which can be used to predict any output for a set of inputs from the testing data.

E. Prediction of Output

After the training of the model is done on the training data, the model with the fixed set of values obtained for all the weights in the network can be used to test it for predicting output values for test dataset. The prediction step just uses the network with the fixed weights to forward propagate the input values from the test data and predict some output value. The output values are then compared each time with the expected value and at the end the accuracy of the model can be determined based on the number of correct predictions made over the testing dataset. The accuracy is a measure of performance which can be used to determine how good the model is to predict data for the application it was trained on.

III. IMPLEMENTATION

For the given assignment, an Artificial Neural Network model is created for binary classification of data. The dataset has points inside the circle of radius 1 labeled as class 1 and points outside the circle and within x and range of [-2, 2] as class 0. The dataset for training and testing is created separately using the dataset generation code provided along with the report. In the dataset points are randomly filled inside and outside the circle and labelled accordingly. The training and testing data is also stored separately in different csv files. There are 100 points each inside and outside the circle in training dataset and 50 points each inside and outside the circle in testing dataset.

Now, first the dataset is shuffled, and uniform random noise is added in the range of [-0.2, 0.2] to every x and y value of points in dataset for both training and testing. Then the neural network is initialized with one input layer, one hidden layer and one output layer. The input layer has two neurons, one each for the x and y co-ordinates of the data point. The hidden layer has 80 neurons, and the output layer has just one neuron which predicts the class in which the input data point lies. The weights of the neural network are randomly initialized at the start of the training process. Now to start the training the training function is called and first separation is done of the input and output values from the dataset given and then starts the training using the back propagation as per the and learning rate initialized for the network. The procedure of the algorithm remains the same as explained in the previous section. First, forward propagation of the input data is done using the initial weights of the network and then based on the error received at the end of the forward pass, it is given as the input to the backward pass which finds the effect of error due to every weight using partial derivative and then updates the weights inside the network and at the end again forward propagation using the new weights of the input is performed and this is carried out till the termination condition is met, which is when two consecutive loss values are identical till 4 decimals in the model provided, and is performed for all the input data points. At the end of the iterations, a new set of the weights for the network layer is received which is basically the value of weights which produces the minimum error in the output value prediction by the end of the training process.

The output of set of the weights for the network received from the training process is then used for the testing or prediction of data for testing dataset. During the testing procedure the weights of the network is fixed and the output value for each of the inputs is determined by the network. The output value is then compared with the actual expected value and then the performance of the model is determined using a confusion matrix and accuracy of the number of correct predictions made during the testing phase. Also, a decision boundary of the binary classification of data is obtained from the testing phase. Now this entire process of training and testing is repeated for three different activation functions used for hidden layer neurons which are logistic, hyperbolic and ReLU. The same set of initialization weights at the start for the network and same training and testing data points with noise is used to compare the performance of all the hypotheses. The final accuracy for all the hypotheses is obtained after 10 runs with each time have different random initialization of random weights at the start of training.

IV. RESULTS

For the assignment, MLP network using back propagation is implemented using the python codes provided in the appendix. First the data set for training and testing is generated using *dataset_gen.py.* Then before starting the data is shuffled and noise is added to the data. The plot of both the datasets can be seen in Fig 3.

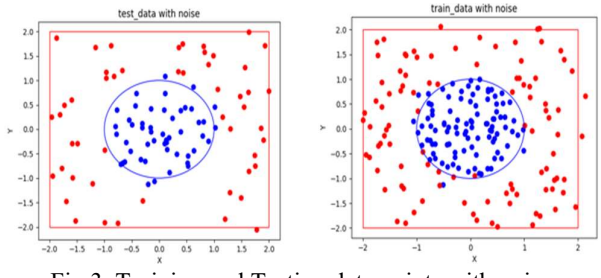

Fig 3. Training and Testing data points with noise

The shuffling and adding noise of dataset is random but can be re-implemented the same using the seed set to 1 at the start of the main function. For training the MLP network is the number of hidden neurons is selected as 80 and the learning rate used is set as 0.01 for the network. Also, at the start of the main function you can select either of hyperbolic, logistic or ReLU as your activation function for the hidden layer neurons. The activation function for output layer is always sigmoid/logistic in the model generated. The weights at the start of every run are randomly initialized but to re-implemented the same weights use and to obtain he exact same values as shown in the current report another seed value is used for randomly initializing the weights of the network. For the assignment, each hypothesis is ran for 10 times, each time with different randomly initialized weights. The seeds used for 10 runs of different randomly initialized weights are 1, 11, 21, and so on till 91. This also ensures the same set of randomly initialized weights is used for 1st run of all three hypotheses.

$$E_{total} = L(y, p') = -y * \log(p') - (1 - y) * \log(1 - p') \quad \text{-(8)}$$

Where, $y =$ label

$p' =$ estimated probability of belonging to positive class

The loss functions used for the current implementation is the error rate provided by equation (4). You can also use the cross entropy as the loss function instead and accordingly use its derivative while backward propagation. The equation of cross entropy loss function can be seen in equation (8). After running the model using Hyperbolic and Logistic activation functions for 10 times the final average confusion matrix for both obtained is the same and is shown in Fig 4.

	Predicted 0	Predicted 1
Actual 0	50	0
Actual 1	5	45

Fig 4. Confusion Matrix for Hyperbolic and Logistic Hypothesis

The confusion matrix data and the terminating epoch and loss value for each run and for all hypothesis is provided in a excel sheet attached with the submission and the performance for all 3 hypotheses is compared as shown below in Fig 5.

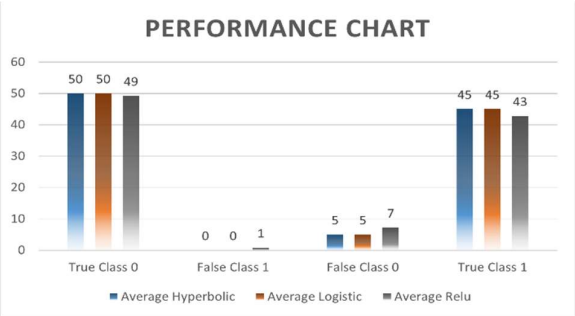

Fig 5. Performance comparison chart of all three hypothesis

The average accuracy obtained while testing is 95% for both Hyperbolic and Logistic Hypothesis and for ReLU it is 92%. The decision boundary obtained at the end of testing for both Hyperbolic and Logistic Hypothesis is as shown in Fig 6.

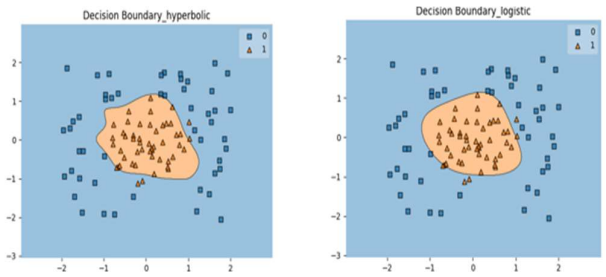

Fig 6. Decision Boundary obtained for Hyperbolic and Logistic Hypothesis

The decision boundary for ReLU is attached as an image in the submission. The graph showing how the loss function is converging over the number of iterations can be in Fig 7. The loss function for ReLu is also provided as an image attached along with submission. All the figures generated are using the seed value as 91 for the weights.

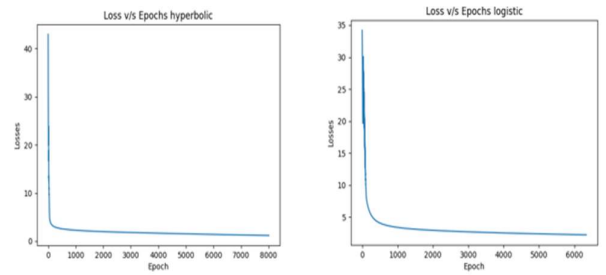

Fig 7. Loss value convergence for Hyperbolic and Logistic Hypothesis

V. Conclusion

In the assignment, a MLP network using backpropagation algorithm is implemented for binary classification using 3 different activation functions for the hidden layer. From the results obtained we can clearly see that the performance of the MLP model using Hyperbolic and Logistic Hypothesis is very similar and the average confusion matrix after randomly initializing weights for 10 runs is identical. The performance of the model using ReLU is not the same and the accuracy is a little less compared to the other two which can be explained because of the sharp nature of the function, which can also be seen from its decision boundary obtained. Also, the number of iterations taking for the loss function to converge is similar for Hyperbolic and Logistic Hypothesis and for ReLU it converges very quickly comparatively.

References

[1] EEE511: ANC, Module 3: Multilayer networks lecture notes, by Dr. Jennie Si, Fall 2020

[2] Jason Brownlee; How to code a Neural Network with Backpropagation in Python (from scratch); URL: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

[3] Matt Mazur; A Step by Step Backpropogation Example; URL: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

APPENDIX A – *dataset_gen.py*

```
import random
import math
import matplotlib.pyplot as plt
import numpy as np
import csv

'''
Class for (x,y) point
'''
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return str((self.x, self.y))

'''
Class for Circle
'''
class Circle:
    def __init__(self, origin, radius):
        self.origin = origin
        self.radius = radius

origin = Point(0, 0)
radius = 1
circle = Circle(origin, radius)

circle_array, rectangle_array = [], []
plt.figure()

'''
Fill random points inside the circle;
Set range as 50 for testing data and 100 for training data
Plot the data points in a figure
'''
for i in range(0, 50):
    p = random.random() * 2 * math.pi
    r = circle.radius * math.sqrt(random.random())
    x = math.cos(p) * r
    y = math.sin(p) * r
    circle_array.append([x,y,1])
```

```
    plt.scatter(x, y, color = 'blue')


'''
Fill random points outside the circle but inside the square boundary;
Set range as 50 for testing data and 100 for training data
Plot the data points in a figure
'''
while len(rectangle_array) != 50:
    x_point = random.uniform(-2,2)
    y_point = random.uniform(-2,2)
    if np.sqrt(x_point**2 + y_point**2) > 1:
        rectangle_array.append([x_point, y_point, 0])
        plt.scatter(x_point, y_point, color = 'red')

'''
Add the data points to their respective csv files
Training = 100_data_points.csv
Testing = 50_data_points.csv
'''
with open('50_data_points.csv', 'w', newline = '') as f:
    writer = csv.writer(f)
    for row in circle_array:
        writer.writerow(row)
    for row in rectangle_array:
        writer.writerow(row)

'''
Plot the Circle and square class boundaries of the dataset
'''
plt.xlabel('X')
plt.ylabel('Y')
circle_1 = plt.Circle((0,0), 1, color='b', fill=False)
rectangle_1 = plt.Rectangle((-2,-
2),4,4,linewidth=1,edgecolor='r',facecolor='none')
ax = plt.gca()
ax.add_artist(circle_1)
ax.add_patch(rectangle_1)
plt.show()
```

APPENDIX B – Viraj_BackProp.*py*

```
import numpy as np
import copy
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from mlxtend.plotting import plot_decision_regions
from random import seed
from random import uniform

'''
Activation functions to used for hidden layer
'''
sigmoid = lambda x: 1/(1 + np.exp(-x))
sigmoid_prime = lambda x: sigmoid(x)*(1.0-sigmoid(x))
tanh = lambda x: np.tanh(x)
tanh_prime = lambda x: 1-np.tanh(x)**2
relu = np.vectorize(lambda x: x if x>0  else 0)
relu_prime = np.vectorize(lambda x: 1 if x>0 else 0)

classify = lambda y: 0 if y <= 0.5 else 1


'''
Accuracy Prediction function
'''
def accuracy(Y, Y_hat):
    correct = 0
    for y, y_hat in zip(Y, Y_hat):
        if y == y_hat:
            correct += 1
    return correct / len(Y)

'''
Function to add noise to the dataset
'''
def add_noise(data_points, msg):
    plt.figure()
    for i in data_points:
        i[0], i[1] = i[0] + uniform(-0.2, 0.2), i[1] + uniform(-0.2, 0.2)
        if i[2] == 1:
            plt.scatter(i[0], i[1], color = 'blue')
        else:
            plt.scatter(i[0], i[1], color = 'red')

    plt.title('{}_data with noise'.format(msg))
```

```
    plt.ylabel('Y')
    plt.xlabel('X')
    circle_1 = plt.Circle((0,0), 1, color='b', fill=False)
    rectangle_1 = plt.Rectangle((-2,-
2),4,4,linewidth=1,edgecolor='r',facecolor='none')
    ax = plt.gca()
    ax.add_artist(circle_1)
    ax.add_patch(rectangle_1)
    plt.savefig("{}_noisy_data.png".format(msg))
    plt.cla()

    return data_points


'''
Class for MLP network
'''
class NN:
    def __init__(self, activation_fnc, epsilon=0.01, hidden_n=80):
        self.activation_fnc = activation_fnc
        self.hidden_n = hidden_n
        self.epsilon = epsilon

        if activation_fnc == "logistic":
            self.act = sigmoid
            self.act_p = sigmoid_prime
        elif activation_fnc == "hyperbolic":
            self.act = tanh
            self.act_p = tanh_prime
        elif activation_fnc == "relu":
            self.act = relu
            self.act_p = relu_prime

        #Set the random seed for initialising weights (1,11,...,91)
        np.random.seed(1)

        self.weights = {
            'W1': np.random.randn(hidden_n, 2),
            'b1': np.zeros(hidden_n),
            'W2': np.random.randn(hidden_n),
            'b2': 0,
        }

    '''
    Forward propoagtion function for MLP model
```

```
    '''
    def forward_propagation(self, X):
        # this implement the vectorized equations defined above.
        Z1 = np.dot(X, self.weights['W1'].T)  + self.weights['b1']
        H = self.act(Z1)
        Z2 = np.dot(H, self.weights['W2'].T) + self.weights['b2']
        Y = sigmoid(Z2)
        return Y, Z2, H, Z1

    '''
    Backward propoagtion function for MLP model
    '''
    def back_propagation(self, X, Y_T):
        N_points = X.shape[0]

        # forward propagation
        Y, Z2, H, Z1 = self.forward_propagation(X)
        L = (1/2) * np.sum((Y_T - Y)**2)

        # back propagation
        dLdY = -(Y_T - Y)
        dLdZ2 = np.multiply(dLdY, sigmoid_prime(Z2))
        dLdW2 = np.dot(H.T, dLdZ2)
        dLdb2 = np.dot(dLdZ2.T, np.ones(N_points))
        dLdH = np.dot(dLdZ2.reshape(N_points, 1), self.weights['W2'].reshape(1, s
elf.hidden_n))
        dLdZ1 = np.multiply(dLdH, self.act_p(Z1))
        dLdW1 = np.dot(dLdZ1.T, X)
        dLdb1 = np.dot(dLdZ1.T, np.ones(N_points))

        self.gradients = {
            'W1': dLdW1,
            'b1': dLdb1,
            'W2': dLdW2,
            'b2': dLdb2,
        }
        return L

    '''
    Training function for MLP model
    '''
    def training(self, train_data):
        X = train_data[:, :2]
        Y = train_data[:, 2:].flatten()
```

```
        losses = []

        print("="*80)
        print("Training MLP with thresholding function: {}\n".format(self.activat
ion_fnc))

        L = self.back_propagation(X, Y)
        losses.append(L)
        iteration = 1

        while 1:
        #Terminate condition set as when two consecutive loss values are equal ti
ll 4 decimal

            for weight_name in self.weights:
                self.weights[weight_name] -
= self.epsilon * self.gradients[weight_name]

            L = self.back_propagation(X, Y)
            losses.append(L)
            iteration += 1

            if round(losses[-1], 4) == round(losses[-2], 4):
                print("Epoch end:  {}\nFinal loss={L:.4f}".format(iteration, L=L)
)
                break

            if iteration % 100 == 0:
                print("epoch:  {}\tlosses={L:.4f}".format(iteration, L=L))

        plt.plot(losses)
        plt.title('Loss v/s Epochs {}'.format(self.activation_fnc))
        plt.ylabel('Losses')
        plt.xlabel('Epoch')
        plt.savefig("losses_{}.png".format(self.activation_fnc))
        plt.cla()

    '''
    Prediction/Testing function for MLP model
    '''
    def predict(self, test_data):
        X = test_data[:, :2]
        Y = test_data[:, 2:].flatten()

        Y_, _, _, _ = self.forward_propagation(X)
```

```
        Y_hat = np.array(list(map(classify, Y_)))

        return Y_hat


if __name__ == "__main__":

    #Select the Activation function for hidden layer
    activation_fnc = "hyperbolic"  #    hyperbolic logistic relu

    seed(1)
    np.random.seed(1)

    #Read training and testing data
    train_data = np.genfromtxt("100_data_points.csv", delimiter=',')
    test_data = np.genfromtxt("50_data_points.csv", delimiter=',')

    #Shuffle training and testing data
    np.random.shuffle(train_data)
    np.random.shuffle(test_data)

    #Add noise to training and testing data
    train_data = add_noise(train_data, "train")
    test_data = add_noise(test_data, "test")

    #Separate Input and Ouput values
    X = test_data[:, :2]
    Y = test_data[:, -1]

    nn = NN(activation_fnc)
    nn.training(train_data)
    Y_hat = nn.predict(test_data)

    print("Accuracy = {}".format(accuracy(Y, Y_hat)))

    confusion_mat = confusion_matrix(Y, Y_hat)
    print("confusion_matrix:\n", confusion_mat)

    plot_decision_regions(X, Y.astype(np.int64), clf=nn)
    plt.title('Decision Boundary_{}'.format(activation_fnc))

    plt.savefig("decision_boundary_{}.png".format(activation_fnc))
```