NAME	**Shreya Shetty**
UID	**2019140059**
CLASS	**TE IT**
BATCH	**B**
ACADEMIC YEAR	**2021-22**
SUBJECT	**SC (Soft Computing)**
COURSE CODE	**IT312**
EXPERIMENT NO.	**5**

Aim:

Write a program to design a Multilayer Perceptron Learning algorithm (EBPTA)

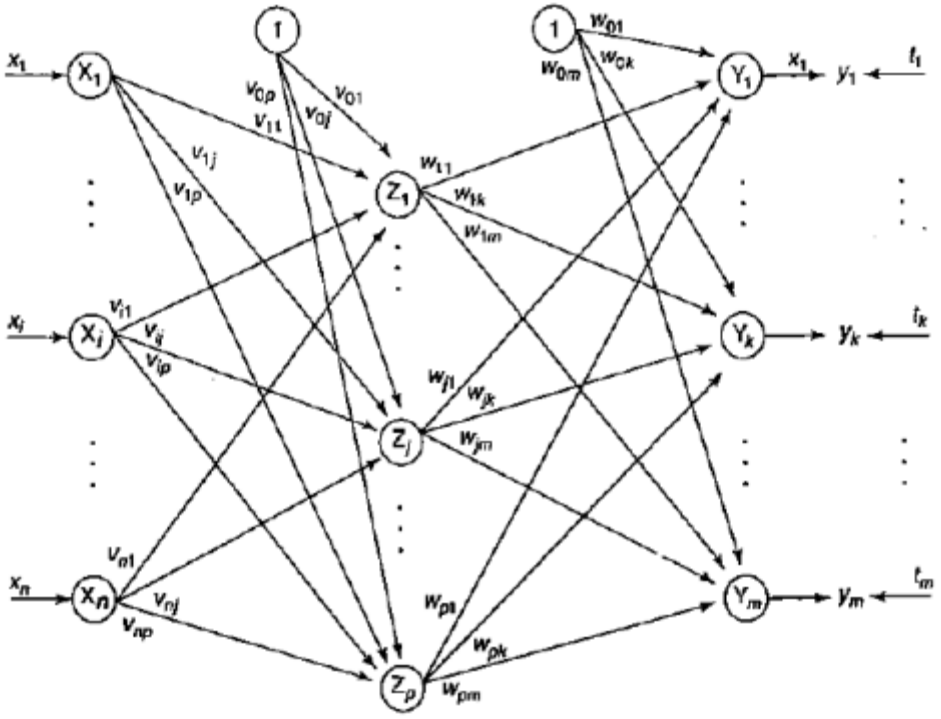
Theory:

Back-Propagation

- The back propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986).
- This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions.
- The networks associated with back-propagation learning algorithm are also called backpropagation networks. (BPNs).
- For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly.
- The basic concept for this weight update algorithm is simply the gradient descent method.
- This is a methods were error is propagated back to the hidden unit.
- Back propagation network is a training algorithm.
- The training of the BPN is done in three stages - the feed-forward of the input training pattern, the calculation and back-propagation of the error, and updation of weights.
- The testing of the BPN involves the computation of feed-forward phase only.
- There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient.
- Even though the training is very slow, once the network is trained it can produce its outputs very rapidly

Architecture

The below figure shows the architectures of back-propagation network.



- A back-propagation neural network is a multilayer, feed-forward neural network consisting of an input layer, a hidden layer and an output layer.
- The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1.
- The bias terms also acts as weights. During the back propagation phase of learning, signals are sent in the reverse direction.
- The inputs sent to the BPN and the output obtained from the net could be either binary (0, 1) or bipolar (-1, +1).
- The activation function could be any function which increases monotonically and is also differentiable.

Flowchart for Training Process

The terminologies used in the flowchart and in the training algorithm are as follows:

x = input training vector (x_1,….,x_i,….x_n)

t = target output vector (t_1,….,t_k,….t_m)

α= learning rate parameter

x_i = input unit i. (Since the input layer uses identity activation function, the input and output signals here are same.)

v_{0i} = bias on jth hidden unit

w_{0k} = bias on kth output unit

z_j=hidden unit j. The net input to z_j is

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is

$$z_j = f(z_{inj})$$

y_k = output unit k. The net input to y_k is

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

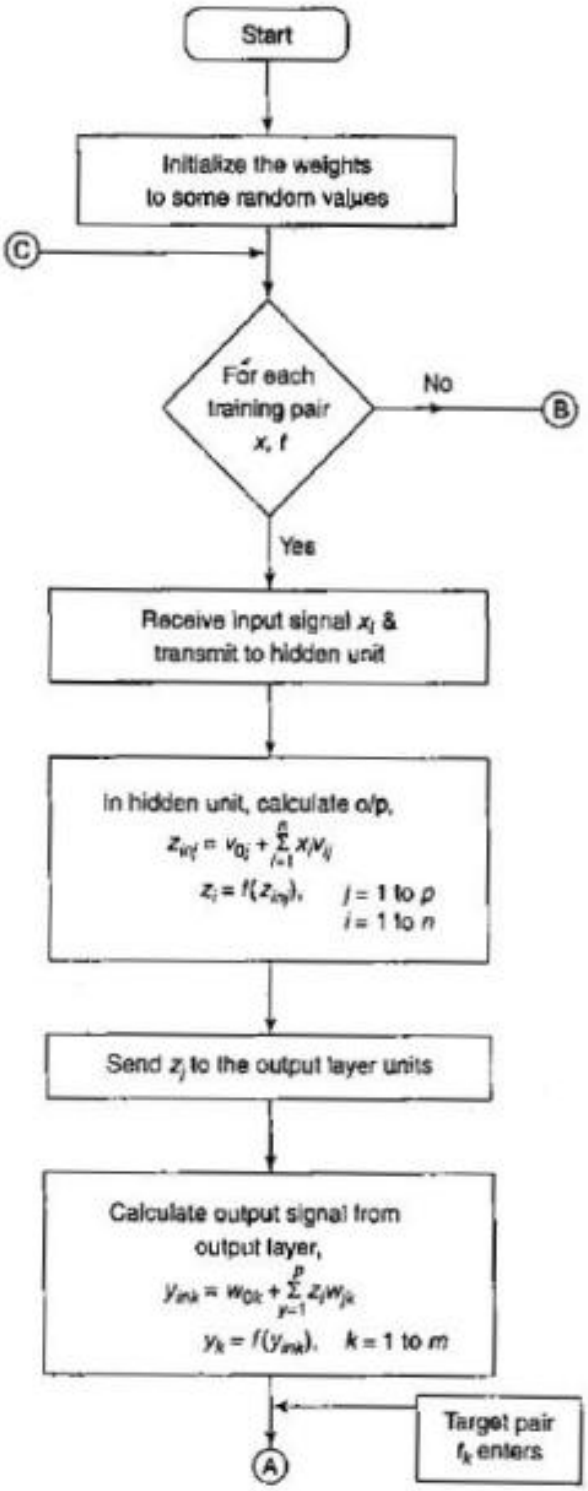
and the output is

$$y_k = f(y_{ink})$$

δ_k = error correction weight adjustment for wjk that is due to an error in unit yk, which is back-propagated to the hidden units that feed into unit yk

δ_j = error correction weight adjustment for vij that is due to the back-propagation of error to the hidden unit is z_j

The commonly used activation functions are binary, sigmoidal and bipolar sigmoidal activation functions. These functions are used in the BPN because of the following characteristics: (i) Continuity (ii) Differentiability iii) Non decreasing monotonic.


Start
Initialize the weights to some random values
C
For each training pair x, t
No
B
Yes
Receive input signal x_i & transmit to hidden unit
In hidden unit, calculate o/p,
$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$
$z_j = f(z_{inj})$, $j = 1$ to p
$i = 1$ to n
Send z_j to the output layer units
Calculate output signal from output layer,
$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$
$y_k = f(y_{ink})$, $k = 1$ to m
Target pair t_k enters
A

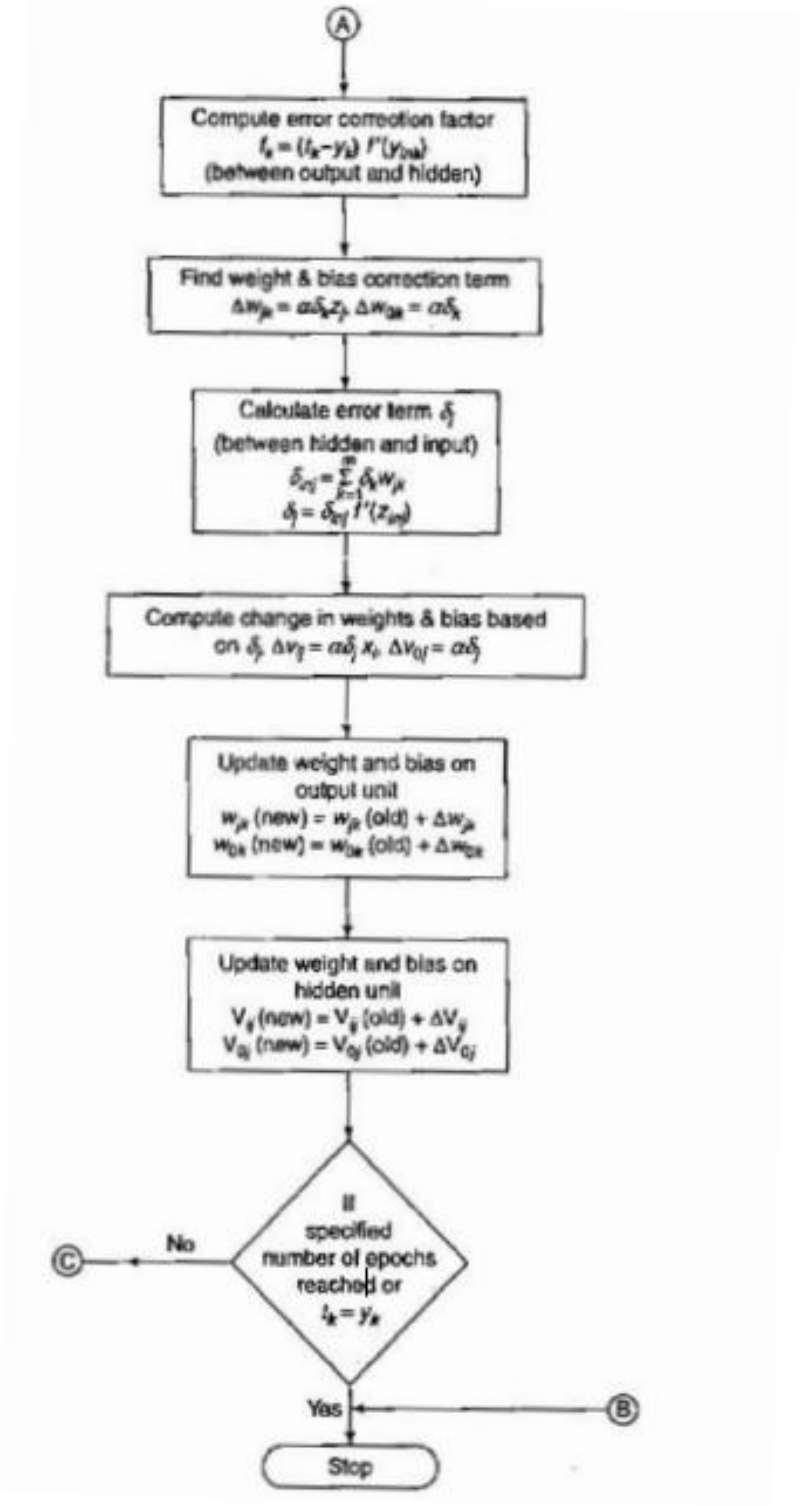


Figure 2.7: Flowchart for back - propagation network training

Training Algorilhm

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2-9 when stopping condition is false.

Step 2: Perform Steps 3-8 for each training pair.

Feedforward Phase 1

Step 3: Each input unit receives input signal xi and sends it to the hidden unit (i = l to n).

Step 4: Each hidden unit zj (j = 1 to p) sums its weighted input signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation functions over $zinj$ (binary or bipolar sigmoidal activation function):

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units.

Step 5: For each output unit yk (k = 1 to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

Back-propagation of error (Phase II)

Step 6: Each output unit yk(k=1 to m) receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative $f'(yink)$ can be calculated as in activation function section. On the basis of the calculated error correction term, update the change in weights and bias:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

Also, send δk to the hidden layer backwards

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

Step 7: Each hidden unit (zj = 1 to p) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

The term δinj gets multiplied with the derivative of $f(zinj)$ to calculate the error term:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(zinj)$ can be calculated as activation function section depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δj , update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$

Weight and bias updation (Phase III):

Step 8: Each output unit (yk, k = 1 to m) updates the bias and weights:

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

$$w_{0k}(new) = w_{0k}(old) + \Delta w_{0k}$$

Each hidden unit (zj; j = 1 to p) updates its bias and weights:

$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$$

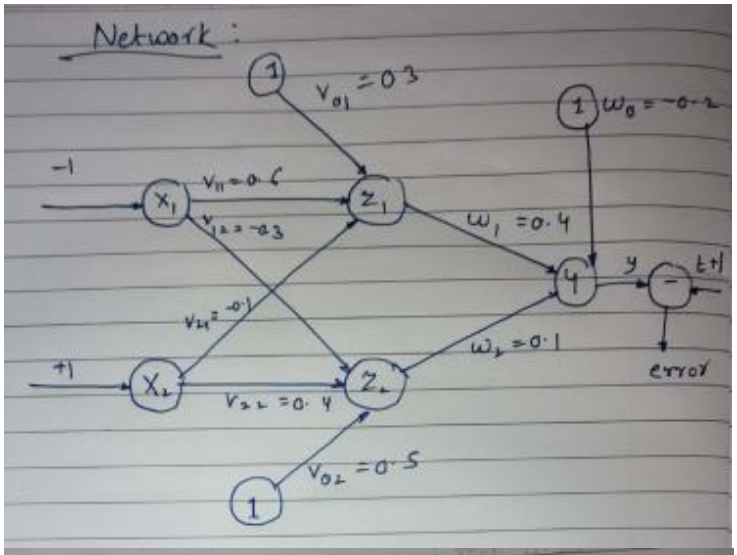
$$v_{0j}(new) = v_{0j}(old) + \Delta v_{0j}$$

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

Procedure:

1. Initialize the weights which are applied to all the neurons.
2. Then, Forward propagation is performed where the inputs from a training set are passed through the neural network and an output is computed.
3. Since we are working with a training set, the correct output is known. An error function is defined, which captures the delta between the correct output and the actual output of the model, given the current model weights.
4. Then perform back propagation to change the weights for the neurons, in order to bring the error function to a minimum.
5. Finally, update the weights are to the optimal values according to the results of the back propagation algorithm.
6. Since, the weights are updated a small delta step at a time, several iterations are required in order for the network to learn. After each iteration, the gradient descent force updates the weights towards less and less global loss function. The amount of iterations needed to converge depends on the learning rate, the network meta-parameters, and the ptimization method used.

Solved Example:



Initial weights given

$$\begin{bmatrix} V_{11} & V_{21} & V_{01} \\ V_{12} & V_{22} & V_{02} \end{bmatrix} = \begin{bmatrix} 0.6 & -0.1 & 0.3 \\ -0.3 & 0.4 & 0.5 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_2 & w_0 \end{bmatrix} = \begin{bmatrix} 0.4 & 0.1 & -0.2 \end{bmatrix}$$

Learning Rate (α) = 0.25

Input pattern $[x_1, x_2] = [-1, 1]$

Target Output (t) : +1

Use bipolar sigmoidal activation function

Activation function used in binary sigmoidal activation function is given by -

$$f(x) = \frac{2}{1+e^{-x}} - 1 = \frac{1-e^{-x}}{1+e^{-x}}$$

Now, Input sample $[x_1, x_2] = [-1, 1]$ ↑
target $t = 1$

→ Calculate net input : for layer Z_1

$$Z_{in1} = V_{01} + x_1 V_{11} + x_2 V_{21}$$

$$= 0.3 + (-1) \times 0.6 + 1 \times (-0.1)$$

$$\therefore Z_{in1} = -0.4$$

for layer Z_2,

$$Z_{in2} = V_{02} + x_1 V_{12} + x_2 V_{22}$$

$$= 0.5 + (-1) \times (0.3) + 1 \times (0.4)$$

$$\therefore Z_{in2} = 1.2$$

Applying activation to calculate output –

$$z_1 = f(z_{in1}) = \frac{1 - e^{-z_{in1}}}{1 + e^{-z_{in1}}}$$

$$= \frac{1 - e^{+0.4}}{1 + e^{+0.4}}$$

$$\therefore z_1 = -0.1974$$

$$z_2 = f(z_{in2}) = \frac{1 - e^{-z_{in2}}}{1 + e^{-z_{in2}}}$$

$$= \frac{1 - e^{-1.2}}{1 + e^{-1.2}}$$

$$\therefore z_2 = 0.537$$

→ Calculate net input entering output layer for y layer –

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$

$$= -0.2 + (-0.1974) \times 0.4 + 0.537 \times 0.1$$

$$\therefore y_{in} = -0.22526$$

Applying activation to calculate output –

$$y = f(y_{in}) = \frac{1 - e^{-y_{in}}}{1 + e^{-y_{in}}}$$

$$= \frac{1 - e^{+0.22526}}{1 + e^{+0.22526}}$$

$$\therefore y = -0.1122$$

→ Compute error portion δ_k :

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = 0.5\left(1 + f(y_{in})\right)\left[1 - f(y_{in})\right]$$

$$= 0.5\,[1 - 0.1122]\,[1 + 0.1122]$$

$$\therefore f'(y_{in}) = 0.4937$$

$$\Rightarrow \delta_1 = (1 + 0.1122)(0.4937) = 0.5491$$

→ finding changes in weights between hidden & output layer :

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.5491 \times (-0.1974)$$

$$\therefore \Delta w_1 = -0.0271$$

$$\Delta w_2 = \alpha \delta_2 z_2 = 0.25 \times 0.5491 \times 0.537$$

$$\therefore \Delta w_2 = 0.0737$$

$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.5491 = 0.1373$$

→ Computing error portion δ_j between input & hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{in_j} f'(z_{in_j})$$

$$\delta_{in_j} = \sum_{k=1}^{m} \delta_k w_{jk}$$

~~$\delta_{in_j} = \delta w_j$~~

$$\delta_{in_j} = \delta_1 w_{j1}$$ ... since, only one output neuron

$\Rightarrow \delta_{in1} = \delta_1 \omega_{11} = 0.5491 \times 0.4 = 0.21964$

$\Rightarrow \delta_{in2} = \delta_1 \omega_{21} = 0.5491 \times 0.1 = 0.05491$

Error, $\delta_1 = \delta_{in1} \times f'(z_{in1})$

$= 0.21964 \times [0.5 \times (1 + 0.1974)(1 - 0.$

$\therefore \delta_1 = 0.1056$

Error, $\delta_2 = \delta_{in2} \cdot f'(z_{in2})$

$= (0.005491) \times [0.5 \times (1 - 0.537)(1 + 0.537)]$

$\therefore \delta_2 = 0.0195$

Now, we find changes in weights between input & hidden layer;

$\Delta V_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.1056 \times (-1)$

$\therefore \Delta V_{11} = -0.0264$

$\Delta V_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.1056 \times 1$

$\therefore \Delta V_{21} = 0.0264$

$\Delta V_{01} = \alpha \delta_1 = 0.25 \times 0.1056 = 0.0264$

$\Delta V_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.0195 \times (-1) = -0.0049$

$\Delta V_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.0195 \times 1 = 0.0049$

$\Delta V_{02} = \alpha \delta_2 = 0.25 \times 0.0195 = 0.0049$

Now, computing final weights of network:

$V_{11}(new) = V_{11}(old) + \Delta V_{11}$
$= 0.6 - 0.0264$
$\therefore V_{11}(new) = 0.5736$

$V_{12}(new) = V_{12}(old) + \Delta V_{12} = -0.3 - 0.0049$
$\therefore V_{12}(new) = -0.5736 - 0.3049$

$V_{21}(new) = V_{21}(old) + \Delta V_{21} = -0.1 + 0.0264$
$\therefore V_{21}(new) = -0.0736$

$V_{22}(new) = V_{22}(old) + \Delta V_{22}$
$= 0.4 + 0.0049$
$\therefore V_{22}(new) = 0.4049$

$w_1(new) = w_1(old) + \Delta w_1 = 0.4 - 0.0271$
$\therefore w_1(new) = 0.3729$

$w_2(new) = w_2(old) + \Delta w_2 = 0.1 + 0.0737$
$\therefore w_2(new) = 0.1737$

$V_{01}(new) = V_{01}(old) + \Delta V_{01} = 0.3 + 0.0264$
$\therefore V_{01}(new) = 0.3264$

$V_{02}(new) = V_{02}(old) + \Delta V_{02}$
$= 0.5 + 0.0049$
$\therefore V_{02}(new) = 0.5049$

$w_0(new) = w_0(old) + \Delta w_0$
$= -0.2 + 0.1373$
$\therefore w_0(new) = -0.0627$

Thus, above are the final weights computed

Code:

```
# Importing required libraries
import numpy as np
# Sigmoid Function
def binary_sigmoid(x):
    return (1/(1+np.exp(-x)))
# Derivative used for back propagation
def der_binary_sigmoid(x):
    return x*(1-x)
# Forward Propagation Function
def forward_prop(x,v,w,bias_hidden,bias_output):
    zin=np.dot(x,v)+bias_hidden
    z=binary_sigmoid(zin)
    yin=np.dot(z,w.T)+bias_output
    y=binary_sigmoid(yin)
    return zin,z,yin,y
# Backward propagation function
def      backward_prop(t,y,z,x):
    delta_o=(t-y)*der_binary_sigmoid(y)
    delta_w=alpha*delta_o*z
    delta_b_o=alpha*delta_o
    delta_h=delta_o*w*der_binary_sigmoid(z)
    delta_v=alpha*np.dot(delta_h.T,x)
    delta_b_h=alpha*delta_h
    return delta_w,delta_v,delta_b_o,delta_b_h
# Function to Update weights
def update_weights(w,v,delta_w,delta_v):
    wnew=w+delta_w
    vnew=v+delta_v
    return wnew,vnew
# Function to update bias
def update_bias(bias_hidden,bias_output,delta_b_o,delta_b_h):
    b_h_new=bias_hidden+delta_b_h
    b_o_new=bias_output+delta_b_o
    return b_h_new,b_o_new

t=1 #target
alpha=0.3 #learning rate
```

```
# INPUT
x=np.array([[1,-1]])
v=np.array([[0.6,-0.3],[-0.1,0.4]])
w=np.array([0.4,0.1])

bias_hidden=np.array([0.3,0.5])
bias_output=-0.2

print("To implement Multilayer Perceptron Learning algorithm(EBPTA)\n\n")
print("Initially :\nx = {}\nv = {}\nw = {}\nbias_hidden={}\nbias_output =
{}\n".format(x,v,w,bias_hidden,bias_output))
for _ in range(500):
    #forward propagation
   zin,z,yin,y=forward_prop(x,v,w,bias_hidden,bias_output)
   #backward propagation
   delta_w,delta_v,delta_b_o,delta_b_h=backward_prop(t,y,z,x)
   #update weights
   w,v=update_weights(w,v,delta_w,delta_v)
   #update biases
   bias_hidden,bias_output=update_bias(bias_hidden,bias_output,delta_b_o,delta_b_h)

print("After 500 epochs :\ny = {}\nv = {}\nbias_hidden={}\nbias_output =
{}".format(y,v,bias_hidden,bias_output))
```

Output:

```
To implement Multilayer Perceptron Learning algorithm(EBPTA)

Initially :
x = [[ 1 -1]]
v = [[ 0.6 -0.3]
 [-0.1  0.4]]
w = [0.4 0.1]
bias_hidden=[0.3 0.5]
bias_output = -0.2

After 500 epochs :
y = [[0.95440285]]
v = [[ 0.87706662 -0.57706662]
 [ 0.07768854  0.22231146]]
bias_hidden=[[0.57706662 0.67768854]]
bias_output = [[1.3793767]]
```

Conclusion:

In this experiment, I have implemented the Multilayer Perceptron Learning algorithm in python. I learned the concept of Back-Propagation Learning Algorithm and epoch. With more number of epoch, errors are reduced.