

**Final Year B. Tech (EE)**

**Trimester: I**

**Subject:**

**Artificial Intelligence and Machine Learning**

**Name: Shreerang Mhatre**

**Roll No: 52**

**Class: Ty**

**Batch: A3**

**Experiment No: 06**

**Name of the Experiment**:Implement and test MLP trained with back – propagation algorithm



**Marks** **Teacher’s Signature with date**

**Performed on: 11/10/2023**

**Submitted on: 11/10/2023**



**Aim:** To create a multilayer neural network and train with back propagation algorithm using Python.

**Prerequisite:** Knowledge of MLP, gradient descent method, Least Mean Square Error

**Objective:**

To create a multi-layer neural network and train with back propagation algorithm using Python Programming.

**Components and Equipment required:**

SkLearn Python module, Python software, NumPy and Panda Libraries

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | |  | **Expt. 6- 1** | |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  | **WWW.MITWPU.COM** | | |



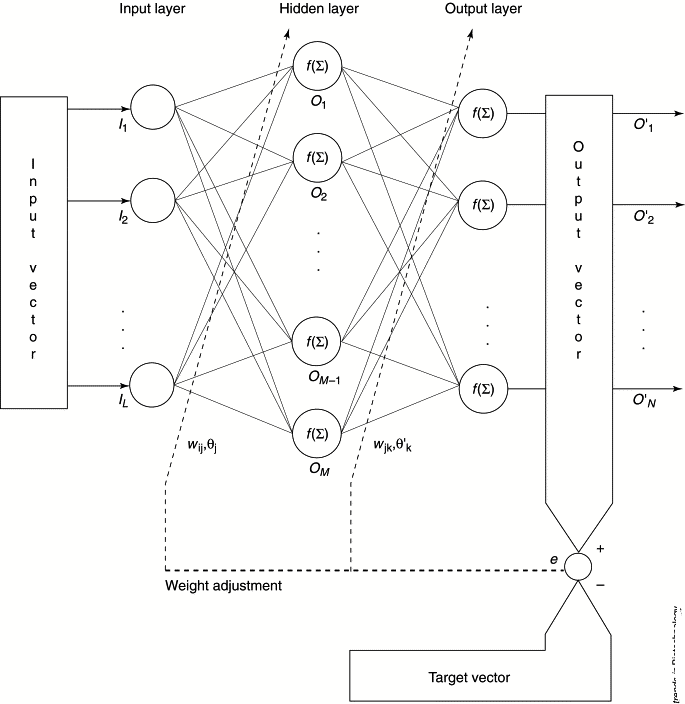
**Theory**

Multilayer feed forward networks are an important class of neural networks. Typically, the network consists of sensory units (source nodes) that constitute the input layer, one or more hidden layers of computation nodes and an output layer of computation nodes. The input signal propagates through the network in the forward direction on a layer-by-layer basis. These neural networks are commonly known as Multilayer Perceptron’s (MLPs)

Multilayer perceptron’s have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with the highly popular algorithm known as the error **back-propagation algorithm**. This algorithm is based on the error-correction learning rule.

Basically error back-propagation learning consists of two passes through the different layers of the network: a forward pass and a backward pass. In the forward pass an activity pattern (input vector) is applied to the sensory nodes of the network and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward pass all synaptic weights of the network are fixed. During the backward pass, all synaptic weights are adjusted in accordance with an error correction rule. Specifically, the actual response of the network is subtracted from the network desired (target) response to produce an error signal. This error signal is then propagated backward through against the direction of the synaptic weights and hence the name “**error back propagation**”..

**Building our model**

****

Procedure

**Step 0:**  Initialize weights. (Set to small random values)

**Step 1:** While stopping condition is false, do steps 2-9.

**Step 2:** For each training pair, do steps 3-8.

***Feed forward*:**

**Step 3:** Each input unit (Xi, i = 1……. n) receives input signal xi and broadcasts this signal to all units in the hidden layer above (the hidden units.

**Step 4:** Each hidden unit (Zj, j = 1……. p) sums its weighted input signals.

Zinj = voj  + ∑ xi vij ; i=1…..n

applies its activation function to compute its output signal.

Zj = f(Zinj)

and sends this signal to all units in the layer above (output units).

**Step 5:** Each output unit (Yk =1……m) sums its weighted input signals.

Yink = wok  + ∑ zj wjk ; i=1…….n

and applies its activation function to compute its output signal.

***Back propagation of error*:**

**Step 6:** Each output unit receives a target pattern corresponding to the input training

pattern, computes its error information term.

δk = (tk-yk) f’(y\_ink)

calculates its bias correction term (used to update wjk )

∆ wjk  = αδkzj

calculates its bias correction term

∆wok= αδk

and sends δk to units in the layer below.



**Step 7:** Each hidden unit sums its delta inputs (from above in the layer),

Δ\_inj = ∑k=1m δk wjk,

Multiplies by the derivative of its activation function to calculate its error information term.

Δj= δ\_inj’(z\_inj),

Calculates its weight correction term

∆ vjk = αδkxj

and calculates its bias correction term

∆voj= αδj

***Update weights and biases***

**Step 8:** Each output unit updates its biases and weights

wjk(new)= wjk(old) + ∆ wjk

each hidden unit updates its biases and weights

vjk(new)= vjk(old) + ∆ vjk

Test stopping condition.

**Python Programming**

import numpy as np

# X = (hours sleeping, hours studying), y = test score of the student

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([92], [86], [89]), dtype=float)

# scale units

X = X/np.amax (X, axis=0) #maximum of X array

y = y/100 # maximum test score is 100

class NeuralNetwork(object):

def \_\_init\_\_(self):

#parameters

self.inputSize = 2

self.outputSize = 1

self.hiddenSize = 3

#weights

self. W1 = np.random.randn(self.inputSize, self.hiddenSize) # (2x3) weight matrix from input to hidden layer

self. W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer

def feedForward(self, X):

#forward propogation through the network

self.z = np.dot(X, self.W1) #dot product of X (input) and first set of weights (3x2)

self. z2 = self.sigmoid(self.z) #activation function

self. z3 = np.dot (self. z2, self. W2) #dot product of hidden layer (z2) and second set of weights (3x1)

output = self.sigmoid(self.z3)

return output

def sigmoid (self, s, deriv=False):

if (deriv == True):

return s \* (1 - s)

return 1/ (1 + np.exp(-s))

def backward (self, X, y, output):

#backward propogate through the network

self.output\_error = y - output # error in output

self.output\_delta = self.output\_error \* self.sigmoid(output, deriv=True)

self. z2\_error = self.output\_delta.dot (self. W2. T) #z2 error: how much our hidden layer weights contribute to output error

self. z2\_delta = self. z2\_error \* self. Sigmoid (self. z2, deriv=True) #applying derivative of sigmoid to z2 error

self. W1 += X.T.dot (self. z2\_delta) # adjusting first set (input -> hidden) weights

self. W2 += self.z2.T.dot (self. output delta) # adjusting second set (hidden -> output) weights

def train (self, X, y):

output = self.feedForward(X)

self.backward(X, y, output)

NN = Neural Network ()

for i in range (1000): #trains the NN 1000 times

if (i % 100 == 0):

print ("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))

NN.train(X, y)

print ("Input: " + str(X))

print ("Actual Output: " + str(y))

print ("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))

print("\n")

print ("Predicted Output: " + str(NN.feedForward(X)))

**Output**

Loss: 0.00024141756958904204

Loss: 0.00021544094373364948

Loss: 0.00019600501703614026

Loss: 0.000179502381372854

Loss: 0.00016538139974727012

Loss: 0.0001532073361205993

Loss: 0.00014263506354982082

Loss: 0.000133389143354652

Loss: 0.00012524850080110458

Input: [[0.66666667 1.] [0.33333333 0.55555556]

[1. 0.66666667]] Actual Output: [[0.92] [0.86] [0.89]] Loss: 0.00011803465359404784

Predicted Output: [[0.90612361] [0.87271003] [0.89007064]]

**Conclusion:**

**Post Lab Questions:**

1. Explain the method to initialize weights for a Backpropagation network.
2. Explain the choice of learning rate parameter.
3. Explain Generalization.
4. How many training data patterns should be used to train a backpropagation network?
5. How to determine the number of Hidden Layer Nodes?

**Expt. 6- 9**



**WWW.MITWPU.COM**

