Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Final Year B. Tech (EE)

Trimester: I **Subject:** **Artificial Intelligence and Machine Learning**

Name: Shreerang Mhatre **Class: Ty**

Roll No: 52 **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

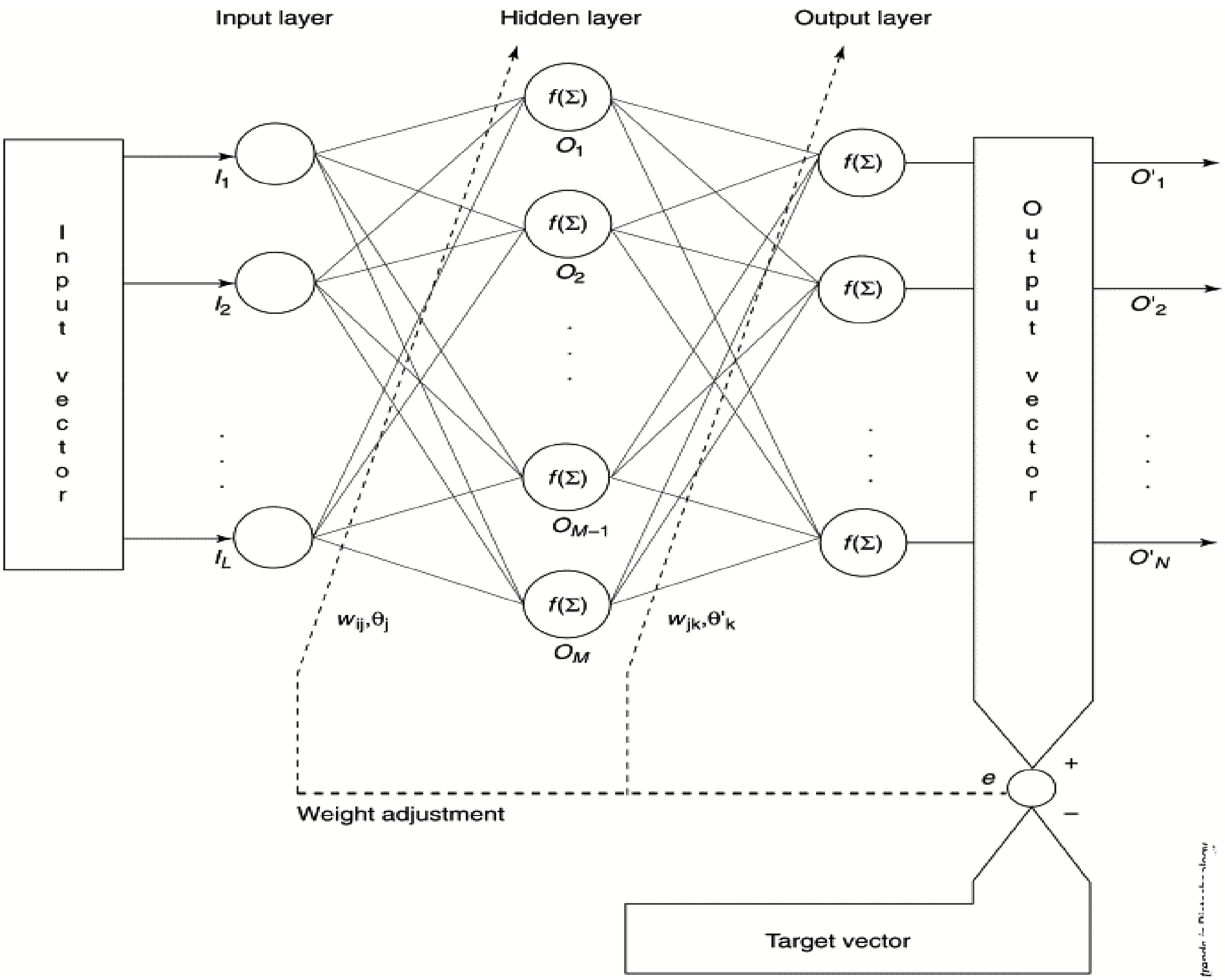
Theory

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 ($X_{i,}$ i = 1……. n) receives input signal $x_{i\text{ and}}$ broadcasts this signal to all units in the hidden layer above (the hidden units.

Step 4: Each hidden unit ($Z_{j,\ j}$ = 1……. p) sums its weighted input signals.

$Z_{inj} = v_{oj} + \sum {}^{xi} v_{ij\,;}$ i=1…..n

applies its activation function to compute its output signal.

$$Z_j = f(Z_{inj})$$

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

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

$$Y_{ink} = w_{ok} + \sum {}^{zj} w_{jk\,;} \quad i=1\ldots\ldots n$$

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.

$$\delta_k = (t_k - y_k)\, f'(y\_in_k)$$

calculates its bias correction term (used to update w_{jk})

$$\Delta\ \mathbf{w}_{jk} = \alpha\delta_k z_j$$

calculates its bias correction term

$$\Delta \mathbf{w}_{ok=}\ \alpha\delta_k$$

and sends δ_k to units in the layer below.

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

$$\Delta_{\_inj} = \sum_{k=1}^{m} \delta_k w_{jk},$$

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

$$\Delta_j = \delta_{\_inj}'(z\_inj),$$

Calculates its weight correction term

$$\Delta \mathbf{v}_{jk} = \alpha\delta_k x_j$$

and calculates its bias correction term

$$\Delta \mathbf{v}_{oj=} \alpha\delta_j$$

Update weights and biases

Step 8: Each output unit updates its biases and weights

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

each hidden unit updates its biases and weights

$$v_{jk}(new) = v_{jk(old)} + \Delta \mathbf{v}_{jk}$$

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?

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 (ipykernel)

Run Code

In [3]:

```
import numpy as np

#Data Preparation
x=np.array(([2,9],[1,5],[3,6]),dtype=float)
y=np.array(([92],[86],[89]),dtype=float)
```

In [4]:

```
# Scale units
x= x/ np.amax(x,axis=0)
#Maximum test score=100
y=y/100
print(x)
print(y)
```

```
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
[[0.92]
 [0.86]
 [0.89]]
```

In [8]:

```
#Neural Network Class
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)
        self.W2= np.random.randn(self.hiddenSize, self.outputSize)

    def feedForward(self, X):
        self.z=np.dot(X, self.W1)
        self.z2=self.sigmoid(self.z)

        self.z3=np.dot(self.z2, self.W2)
        output=self.sigmoid(self.z3)
        return output

    def sigmoid(self, s, deriv=False):
        if deriv:
            return s*(1-s)
        return 1/ (1+ np.exp(-s))
```

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 (ipykernel)

Run Code

```
        output=self.sigmoid(self.z3)
        return output

    def sigmoid(self, s, deriv=False):
        if deriv:
            return s*(1-s)
        return 1/ (1+ np.exp(-s))

    def backward(self, X, y, output):
        self.output_error=y-output
        self.output_delta=self.output_error*self.sigmoid(output,deriv=True)
        self.z2_error=self.output_delta.dot(self.W2.T)
        self.z2_delta=self.z2_error*self.sigmoid(self.z2,deriv=True)

        #Update weights
        self.W2+= self.z2.T.dot(self.output_delta)
        self.W1+=X.T.dot(self.z2_delta)

    def train(self,X,y):
        output=self.feedForward(X)
        self.backward(X,y,output)
```

In [9]:

```
#Training the Neural Network
NN= NeuralNetwork()
for i in range(1000):
    if i%100==0:
        print("Loss:" + str(np.mean(np.square(y - NN.feedForward(x)))))
    NN.train(x,y)
```

```
Loss:0.4254937933066482
Loss:0.00019921786226856074
Loss:0.0001948173231228523
Loss:0.00019223173810252743
Loss:0.00018972574675697772
Loss:0.00018729488428071548
Loss:0.00018493570100475493
Loss:0.00018264493331119304
Loss:0.00018041949341614713
Loss:0.00017825645908976824
```

In [10]:

```
#Final Result
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)))
```

File Edit View Insert Cell Kernel Widgets Help   Not Trusted   Python 3 (ipykernel)

Run   Code

In [9]:
```
#Training the Neural Network
NN= NeuralNetwork()
for i in range(1000):
    if i%100==0:
        print("Loss:" + str(np.mean(np.square(y - NN.feedForward(x)))))
    NN.train(x,y)
```

```
Loss:0.4254937933066482
Loss:0.00019921786226856074
Loss:0.0001948173231228523
Loss:0.0001922317381025274З
Loss:0.00018972574675697772
Loss:0.00018729488428071548
Loss:0.00018493570100475493
Loss:0.00018264493331119304
Loss:0.00018041949341614713
Loss:0.00017825645908976824
```

In [10]:
```
#Final Result
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)))
```

```
Input: [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual output: [[0.92]
 [0.86]
 [0.89]]
Loss: 0.00017615306388121677


Predicted Output: [[0.90113699]
 [0.8723686 ]
 [0.89443436]]
```

In [ ]:

Exp-6 - MLP trained with back-propogation

PAGE No.
DATE 11/10/23

* Post Lab Questions -

Q1) Explain the method to initialize weights for a Back propagation network.

-> Back propagation neural network is an important step to ensure the network learns effectively. There are several methods to initialize weights, and the choice can impact training speed and convergence. One common method is to use small random values for the initial weights.

① Random Initialization
② Bias Initialization
③ Xavier / Glorot Initialization
④ He Initialization
⑤ Custom Initialization.

Q2) Explain the choice of learning rate parameter.

-> The learning rate in training neural networks is a pivotal hyperparameter that dictates the step size at which the model adjusts its weights during optimization. Selecting the ideal learning rate necessitates careful consideration and often involves methods like grid search or learning rate scheduling. If the learning rate is too high, the model can diverge or oscillate, while a rate that's too low can result in slow convergence and getting stuck in local minima. Learning rate decay and adaptive optimization algorithms offer strategies to strike a balance between fast initial convergence and fine-tuning. Experimentation and continuous monitoring of training progress, including validation performance, are essential to find the most suitable learning rate for a given neural network architecture, dataset, & problem.

Q3) Explain Generalization.

→ Generalization in machine learning refers to the model's ability to perform well on unseen or new data that it hasn't been trained on. It's a fundamental goal of model training, indicating that the model has learned the underlying patterns and relationships within the training data without merely memorizing it. A well-generalized model can make accurate predictions or classifications for various inputs, not just those it has seen during training.

Q4) How many training data patterns should be used to train a back propagation network?

→ The number of training data patterns required to train a back propagation network depends on various factors, including the complexity of the problem, the architecture of the neural network, and the desired level of generalization. Some general guidlines -

1. Sufficiency.
2. Complexity of the Problem
3. Data Augmentation
4. Cross - validation
5. Regularization.

Thus, there's no fixed number of training data patterns that applies universally. The goal is to have enough data to capture the underlying patterns of the problem and achieve good generalization

Q5) How to determine the number of Hidden Layer Nodes?

-> Determining the number of hidden layer nodes in a neural network is a critical aspect of its architecture design. while there's no one-size-fits-all rule, some general guidelines can help. Firstly, start with a minimal number of nodes and gradually increase as needed. Too few nodes can lead to under fitting, while too many may result in over fitting. Consider the complexity of your problem and dataset size more complex problems often require more nodes. Ultimately the number of hidden layer nodes should be chosen based on the specific problem data, and empirical performance evaluation.