SAP ID:

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

COURSE CODE: DJ19DSC501 **DATE:**

COURSE NAME: Machine Learning - II **CLASS: AY 2023-24**

LAB EXPERIMENT NO.3

60009210105

Amitesh Sawarkar

D 12

AIM:

Implement backpropagation algorithm from scratch.

THEORY:

In the field of Artificial Neural Networks, the Backpropagation algorithm is a supervised learning approach for multilayer feed-forward networks.

The information processing of one or more brain cells, termed neurons, is the inspiration for feed-forward neural networks. A neuron receives input signals through its dendrites, which then transmit the signal to the cell body. The axon transmits the signal to synapses, which are the connections between a cell's axon and the dendrites of other cells.

The backpropagation approach works on the premise of modelling a function by changing the internal weightings of input signals to produce an expected output signal. The system is taught using a supervised learning method, in which the difference between the system's output and a known expected output is supplied to it and utilised to change its internal state.

The backpropagation algorithm, in technical terms, is a method for training the weights in a multilayer feed-forward neural network. As a result, it necessitates the definition of a network structure consisting of one or more levels, each of which is fully connected to the next layer. One input layer, one hidden layer, and one output layer make up a conventional network topology. Backpropagation can be used for both classification and regression problems.

How does backpropagation algorithm work?

As we know in artificial neural networks, training occurs in various steps, from:

- Initialization.
- Forward propagation.
- Error Function.
- Backpropagation.

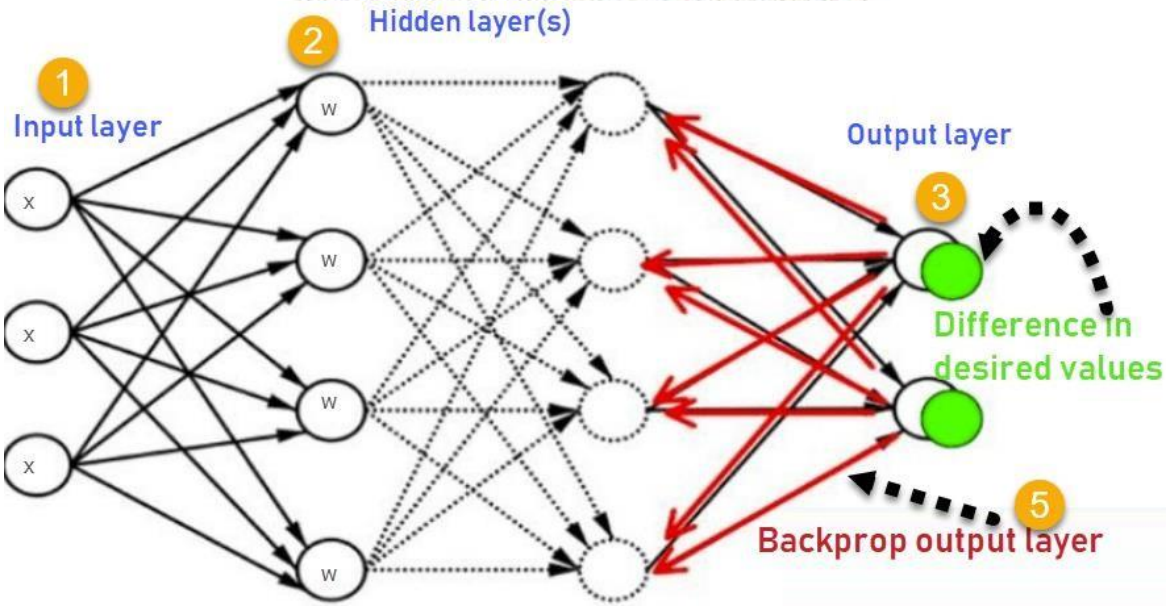
- Weight Update.

- Iteration.

It is the fourth step of the process, a backpropagation algorithm that calculates the gradient of a loss function of the weights in the neural network to ensure the error function is minimum. However, the backpropagation algorithm accomplishes this through a set of Back **Propagation Algorithm Steps**, which involves:

- **Selecting Input & Output:** The first step of the backpropagation algorithm is to choose an input for the process and to set the desired output.
- **Setting Random Weights:** Once the input and output are set, random weights are allocated, as it will be needed to manipulate the input and output values. After this, the output of each neuron is calculated through the forward propagation, which goes through:
    - Input Layer
    - Hidden Layer
    - Output Layer
- **Error Calculation:** This is an important step that calculates the total error by determining how far and suitable the actual output is from the required output. This is done by calculating the errors at the output neuron.
- **Error Minimization:** Based on the observations made in the earlier step, here the focus is on minimizing the error rate to ensure accurate output is delivered.
- **Updating Weights & other Parameters:** If the error rate is high, then parameters (weights and biases) are changed and updated to reduce the rate of error using the delta rule or gradient descent. This is accomplished by assuming a suitable learning rate and propagating backward from the output layer to the previous layer. Acting as an example of dynamic programming, this helps avoid redundant calculations of repeated errors, neurons, and layers.
- **Modeling Prediction Readiness:** Finally, once the error is optimized, the output is tested with some testing inputs to get the desired result.
  This process is repeated until the error reduces to a minimum and the desired output is obtained.

Consider the following Back propagation neural network example diagram to understand:



1. Inputs X, arrive through the preconnected path
2. Input is modelled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

   $Error_B$= Actual Output – Desired Output
5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased

   Process is repeated until the desired output is achieved.

Need of Backpropagation in Neural Network

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

SAP ID:

Tasks to be performed:

a) **Take the iris dataset from kaggle**
b) **Initialize a neural network with random weights.**
c) **Backpropagation Neural Network:**
   i. **Do feedforward propagation**
   ii. **Calculate error**
   iii. **Backpropagation phase**
d) **Perform multiple iterations of step c.**
e) **Update weights accordingly.**
f) **Plot the mean squared error for each iteration**
g) **Similarly plot accuracy for iterations and note the results.**
h) **Predict for test data and evaluate the performance of your backpropagation neural network.**

```
def InitializeWeights(nodes):
    """Initialize weights with random values in [-1, 1] (including bias)"""
    layers, weights = len(nodes), []

    for i in range(1, layers):
        w = [[np.random.uniform(-1, 1) for k in range(nodes[i-1] + 1)]
              for j in range(nodes[i])]
        weights.append(np.matrix(w))

    return weights
```

```
def Sigmoid(x):
    return 1 / (1 + np.exp(-x))

def SigmoidDerivative(x):
    return np.multiply(x, 1-x)
```

```
def Predict(item, weights):
    layers = len(weights)
    item = np.append(1, item)

    activations = ForwardPropagation(item, weights, layers)

    outputFinal = activations[-1].A1
    index = FindMaxActivation(outputFinal)

    y = [0 for i in range(len(outputFinal))]
    y[index] = 1

    return y
```

```
def FindMaxActivation(output):
    """Find max activation in output"""
    m, index = output[0], 0
    for i in range(1, len(output)):
        if(output[i] > m):
            m, index = output[i], i

    return index
```

```
def Accuracy(X, Y, weights):
    """Run set through network, find overall accuracy"""
    correct = 0

    for i in range(len(X)):
        x, y = X[i], list(Y[i])
        guess = Predict(x, weights)

        if(y == guess):
            correct += 1

    return correct / len(X)
```

```
def Train(X, Y, lr, weights):
    layers = len(weights)
    for i in range(len(X)):
        x, y = X[i], Y[i]
        x = np.matrix(np.append(1, x))

        activations = ForwardPropagation(x, weights, layers)
        weights = BackPropagation(y, activations, weights, layers)

    return weights
```

```
def NeuralNetwork(X_train, Y_train, X_val=None, Y_val=None, epochs=10, nodes=[], lr=0.15):
    hidden_layers = len(nodes) - 1
    weights = InitializeWeights(nodes)

    for epoch in range(1, epochs+1):
        weights = Train(X_train, Y_train, lr, weights)

        if(epoch % 20 == 0):
            print("Epoch {}".format(epoch))
            print("Training Accuracy:{}".format(Accuracy(X_train, Y_train, weights)))
            if X_val.any():
                print("Validation Accuracy:{}".format(Accuracy(X_val, Y_val, weights)))

    return weights
```

```
def ForwardPropagation(x, weights, layers):
    activations, layer_input = [x], x
    for j in range(layers):
        activation = Sigmoid(np.dot(layer_input, weights[j].T))
        activations.append(activation)
        layer_input = np.append(1, activation)

    return activations
```

```
def BackPropagation(y, activations, weights, layers):
    outputFinal = activations[-1]
    error = np.matrix(y - outputFinal)

    for j in range(layers, 0, -1):
        currActivation = activations[j]

        if(j > 1):
            prevActivation = np.append(1, activations[j-1])
        else:
            prevActivation = activations[0]

        delta = np.multiply(error, SigmoidDerivative(currActivation))
        weights[j-1] += lr * np.multiply(delta.T, prevActivation)

        w = np.delete(weights[j-1], [0], axis=1)
        error = np.dot(delta, w)

    return weights
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

iris = pd.read_csv("/content/iris.csv")
iris = iris.sample(frac=1).reset_index(drop=True)

X = iris[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']]
X = np.array(X)

one_hot_encoder = OneHotEncoder(sparse=False)
Y = iris.species
Y = one_hot_encoder.fit_transform(np.array(Y).reshape(-1, 1))

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15)
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size=0.1)

epochs = 100
nodes = [len(X[0]), 5, 10, len(Y[0])]
lr = 0.15
```

```
weights = NeuralNetwork(X_train, Y_train, X_val, Y_val, epochs=epochs, nodes=nodes, lr=lr)

def MeanSquaredError(X, Y, weights):
    error = 0

    for i in range(len(X)):
        x, y = X[i], Y[i]
        x = np.matrix(np.append(1, x))

        activations = ForwardPropagation(x, weights, len(weights))
        outputFinal = activations[-1].A1

        error += np.sum((y - outputFinal) ** 2)

    return error / (2 * len(X))
```

```
mse_values = []
accuracy_values = []

for epoch in range(1, epochs+1):
    weights = Train(X_train, Y_train, lr, weights)

    mse = MeanSquaredError(X_train, Y_train, weights)
    mse_values.append(mse)

    if epoch % 20 == 0:
        print("Epoch {}".format(epoch))
        print("Training Accuracy: {}".format(Accuracy(X_train, Y_train, weights)))
        if X_val is not None:
            print("Validation Accuracy: {}".format(Accuracy(X_val, Y_val, weights)))

    accuracy = Accuracy(X_train, Y_train, weights)
    accuracy_values.append(accuracy)
```

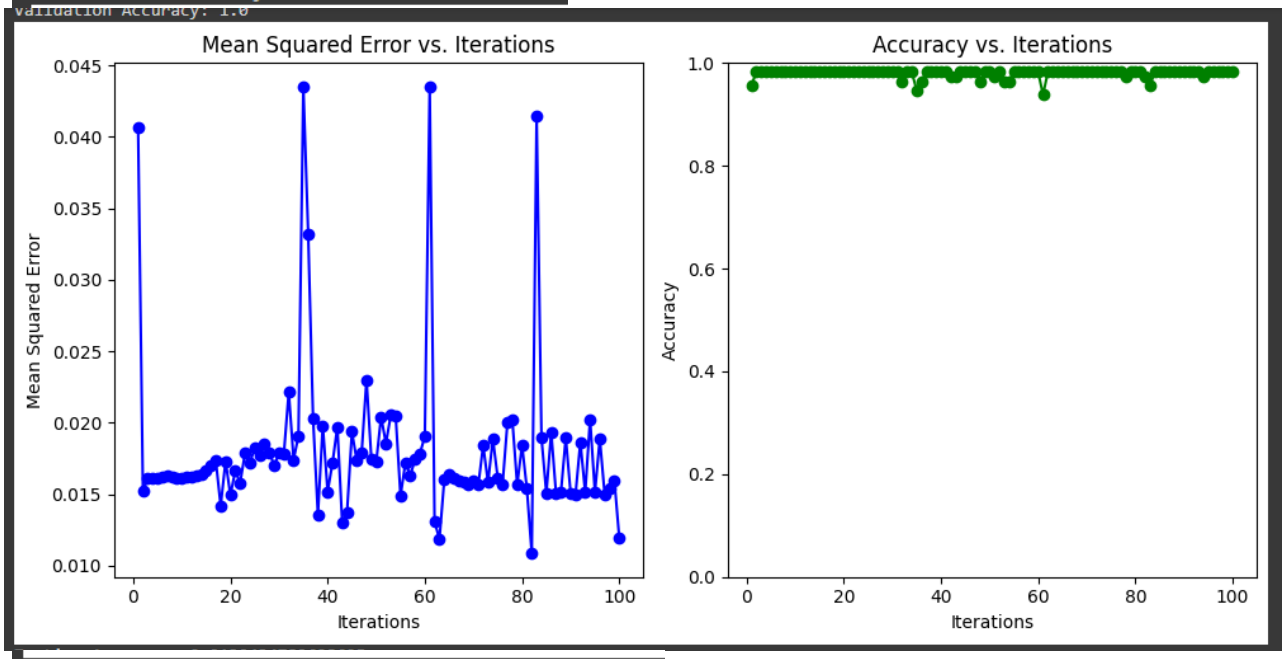
```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, len(mse_values) + 1), mse_values, marker='o', linestyle='-', color='b')
plt.title('Mean Squared Error vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error')
```

```
lt.subplot(1, 2, 2)
lt.plot(range(1, len(accuracy_values) + 1), accuracy_values, marker='o', linestyle='-', color='g'
lt.title('Accuracy vs. Iterations')
lt.xlabel('Iterations')
lt.ylabel('Accuracy')
lt.ylim(0, 1.0)
```

```
plt.tight_layout()
plt.show()

testing_accuracy = Accuracy(X_test, Y_test, weights)
print("Testing Accuracy: {}".format(testing_accuracy))
```

```
Epoch 20
Training Accuracy:0.6578947368421053
Validation Accuracy:0.7692307692307693
Epoch 40
Training Accuracy:0.9649122807017544
Validation Accuracy:1.0
Epoch 60
Training Accuracy:0.9473684210526315
Validation Accuracy:0.9230769230769231
Epoch 80
Training Accuracy:0.9649122807017544
Validation Accuracy:0.9230769230769231
Epoch 100
Training Accuracy:0.9736842105263158
Validation Accuracy:1.0
Epoch 20
Training Accuracy: 0.9824561403508771
Validation Accuracy: 1.0
Epoch 40
Training Accuracy: 0.9824561403508771
Validation Accuracy: 1.0
Epoch 60
Training Accuracy: 0.9824561403508771
Validation Accuracy: 1.0
Epoch 80
Training Accuracy: 0.9824561403508771
Validation Accuracy: 1.0
Epoch 100
Training Accuracy: 0.9824561403508771
Validation Accuracy: 1.0
```




```
Testing Accuracy: 0.9130434782608695
```