**Task 12**

**Write a program to implement artificial neural network with back propagation.**

**Tools: Google co-lab, Python.**

**Aim:**

To implement artificial neural network with back propagation.

**Algorithm:**

**Artificial Neural Network**

* A neural network with a single layer is called a perceptron. A multi-layer perceptron is called Artificial Neural Networks. A Neural network can possess any number of layers. Each layer can have one or more neurons or units. Each of the neurons is interconnected with each and every other neuron. Each layer could have different activation functions as well.
* ANN consists of two phases Forward propagation and Backpropagation. The forward propagation involves multiplying weights, adding bias, and applying activation function to the inputs and propagating it forward.
* The backpropagation step is the most important step which usually involves finding optimal parameters for the model by propagating in the backward direction of the Neural network layers. The backpropagation requires optimization function to find the optimal weights for the model.
* ANN can be applied to both Regression and Classification tasks by changing the activation functions of the output layers accordingly. (Sigmoid activation function for binary classification, Softmax activation function for multi-class classification and Linear activation function for Regression).

**Epochs**

* In terms of ANN, an epoch is one cycle through the full training dataset. Usually training an ANN takes more than a few epochs.
* Different models require different no of epochs and time to train depending on the size of the training data.

**Learning Rate**

* The learning rate of the model controls how quickly the model is adapted to the problem. The learning rate value liles between 0.0 to 1.0.
* Smaller learning rates take a lot of time to train the model since changes in weight are small in each epoch.
* Larger learning rates result in rapid changes in weights and require fewer training epochs.

**Programs:**

#importing the required libraries

import numpy as np

# Define the input data 'X' as a NumPy array.

# Each row represents an input example, and each column represents a feature.

# In this case, we have 4 examples with 2 features each (binary inputs).

X = np.array(([0, 0], [0, 1], [1, 0], [1, 1]), dtype=float)

# Define the corresponding target output 'y' as a NumPy array.

# Each row represents the expected output corresponding to the input example.

# In this case, we have 4 expected outputs (binary values).

y = np.array(([0], [1], [1], [0]), dtype=float)

#The X array represents the input features, which are binary values (0 or 1)

X

array([[0., 0.],

[0., 1.],

[1., 0.],

[1., 1.]])

#y array represents the corresponding expected output for each input example

y

array([[0.],

[1.],

[1.],

[0.]])

# Define a class named NeuralNetwork

class NeuralNetwork(object):

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

# Initialize the network's architecture parameters

self.input = 2 # Number of input neurons

self.output = 1 # Number of output neurons

self.hidden = 3 # Number of neurons in the hidden layer

# Initialize the weights for the neural network

self.W1 = np.random.randn(self.input, self.hidden) # Weight matrix from input to hidden layer (random initialization)

self.W2 = np.random.randn(self.hidden, self.output) # Weight matrix from hidden to output layer (random initialization)

def feedForward(self, X):

# Perform forward propagation through the network

self.z = np.dot(X, self.W1) # Calculate the dot product of inputs (X) and weights (W1)

self.z2 = self.sigmoid(self.z) # Apply the sigmoid activation function to the result (z)

self.z3 = np.dot(self.z2, self.W2) # Calculate the dot product of hidden layer outputs (z2) and weights (W2)

output = self.sigmoid(self.z3) # Apply the sigmoid activation function to the result (z3)

return output

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

# Define the sigmoid activation function and its derivative

if deriv:

return s \* (1 - s) # Derivative of the sigmoid function

return 1 / (1 + np.exp(-s)) # Sigmoid activation function

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

# Perform backward propagation through the network to update weights

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

self.output\_delta = self.output\_error \* self.sigmoid(output, deriv=True) # Calculate the delta for the output layer

self.z2\_error = self.output\_delta.dot(self.W2.T) # Calculate the error for the hidden layer (z2)

self.z2\_delta = self.z2\_error \* self.sigmoid(self.z2, deriv=True) # Calculate the delta for the hidden layer

self.W1 += X.T.dot(self.z2\_delta) # Update the weights of the first layer (input -> hidden)

self.W2 += self.z2.T.dot(self.output\_delta) # Update the weights of the second layer (hidden -> output)

def train(self, X, y):

output = self.feedForward(X) # Perform forward propagation to get the output

self.backward(X, y, output) # Perform backward propagation to update the weights

# Create an instance of the NeuralNetwork class

# This instance can be used to create and train a neural network

# Create an instance of the NeuralNetwork class

NN = NeuralNetwork()

# Loop through the training process for a specified number of epochs (500 times in this case)

for i in range(500):

# Print the loss every 100 epochs

if i % 100 == 0:

# Calculate the loss by finding the mean squared error between the predicted output and the actual output

loss = np.mean(np.square(y - NN.feedForward(X)))

print("Loss", i, ":", loss)

# Train the neural network using the training data (X) and target outputs (y)

NN.train(X, y)

Loss 0 : 0.4561869134161245

Loss 100 : 0.21166085544220267

Loss 200 : 0.1660153250396753

Loss 300 : 0.1409831740040693

Loss 400 : 0.12234738087141583

# Print the input data

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

# Print a newline for separation

print("\n")

# Print the actual target output

print("Actual Output: \n", y)

# Print a newline for separation

print("\n")

# Calculate and print the loss, which is the mean squared error between the actual target output and the predicted output

loss = np.mean(np.square(y - NN.feedForward(X)))

print("Loss: \n" + str(loss))

# Print a newline for separation

print("\n")

# Calculate and print the predicted output from the neural network

predicted\_output = NN.feedForward(X)

print("Predicted Output: " + str(predicted\_output))

Input:

[[0. 0.]

[0. 1.]

[1. 0.]

[1. 1.]]

**Actual Output:**

[[0.]

[1.]

[1.]

[0.]]

**Loss:**

0.10818459770162811

**Predicted Output:** [[0.08578565]

[0.69032782]

[0.68675423]

[0.48099838]]

**Result:**

Thus the program of artificial neural network with back propagation was successfully implemented using python.