**Problem Statement:**

Apply back propagation algorithm on sample {1,0,1} with the class label {1,0}. (Where network topology: 3-2-2-2, all biases and weights are initialized at 0)

**Procedure:**

Step 0: Initialize weights and biases to random number between 0 and 1 for all layers.

Step 1: Forward Pass (Propagation)

For each input neuron calculate the weighted sum for the hidden layer neuron :

Where Wi is the corresponding weight and

For output will be:

Step 2: Calculate the Error (Loss)

Compute the error between the predicted output​ and the actual target label​ using the Mean Squared Error (MSE):

Step 4: Update Weights and Biases

Update the weights and biases using the Gradient Descent rule:

Update weights for each layer:

Update biases for each layer:

Step 5: Repeat the Process

Repeat the process for several iterations (epochs), updating weights and biases until the error converges to a minimal value, or until a predefined number of epochs is reached.

**Source code:**

import numpy as np

class NeuralNetwork():

def \_\_init\_\_(self, input\_layer\_size, hidden\_layer\_size, output\_layer\_size, learning\_rate):

self.input\_size = input\_layer\_size

self.hidden\_size = hidden\_layer\_size

self.output\_size = output\_layer\_size

#self.w1 = np.round(np.random.rand(self.input\_size, self.hidden\_size) - 0.5, 3)

#self.w2 = np.round(np.random.rand(self.hidden\_size, self.output\_size) - 0.5, 3)

#self.b1 = np.round(np.random.rand(1, self.hidden\_size) - 0.5,3)

#self.b2 = np.round(np.random.rand(1, self.output\_size) - 0.5,3)

self.w1 = np.array([[0.2,-0.3],[0.4, 0.1],[-0.5,0.2]])

self.w2 = np.array([-0.3,-0.2])

self.b1 = np.array([-0.4,0.2])

self.b2 = np.array([0.1])

self.learning\_rate = learning\_rate

self.error\_list = []

self.limit = 0.5

self.true\_positives = 0

self.false\_positives = 0

self.true\_negatives = 0

self.false\_negatives = 0

print("Initial Input layer to Hidden layer weights: ", self.w1)

print("Initial Hidden layer to output layer weights: ", self.w2)

print("Initial Biases of Hidden layer: ", self.b1)

print("Initial Bias of Output layer: ",self.b2)

def \_\_sigmoid(self, x):

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

def \_\_sigmoid\_derivative(self, x):

return x \* (1 - x)

def \_\_forward(self, X):

self.z1 = np.dot(X, self.w1) + self.b1

self.a1 = self.\_\_sigmoid(self.z1)

print("Output of hidden layer: ",self.a1)

self.z2 = np.dot(self.a1, self.w2) + self.b2

a2 = self.\_\_sigmoid(self.z2)

print("Output of output layer: ",a2)

return a2

def \_\_backpropagation(self, X, target, output):

self.o\_error = (target - output)

self.o\_delta = self.o\_error \* self.\_\_sigmoid\_derivative(output)

print('delta Output: ',self.o\_delta)

self.z2\_error = self.o\_delta\*self.w2

print('Error of output layer : ',self.z2\_error)

self.z2\_delta = self.z2\_error \* self.\_\_sigmoid\_derivative(self.a1)

print('Delta error of output layer: ',self.z2\_delta)

self.w1 += self.learning\_rate \* np.outer(X, self.z2\_delta)

# Use np.dot to get the correct dimensions for updating w2

self.w2 += self.learning\_rate \* np.dot(self.a1.reshape(self.hidden\_size,self.output\_size), self.o\_delta)

self.b1 += self.learning\_rate \* self.z2\_delta

self.b2 += self.learning\_rate \* self.o\_delta

print('New weight of input layer to hidden layer: ',self.w1)

print('New weight of hidden layer to output layer: ',self.w2)

print('New baises of hidden layer: ',self.b1)

print('New bias of output layer: ',self.b2)

def train(self, X, y, epochs):

for epoch in range(epochs):

print('Epoch: ',epoch)

o = self.\_\_forward(X)

self.\_\_backpropagation(X, y, o)

def predict(self, x\_predicted):

return self.\_\_forward(x\_predicted).item()

if \_\_name\_\_ == "\_\_main\_\_":

n = NeuralNetwork(3,2,1,0.9)

input = np.array([1,0,1])

n.train(input,1,100)

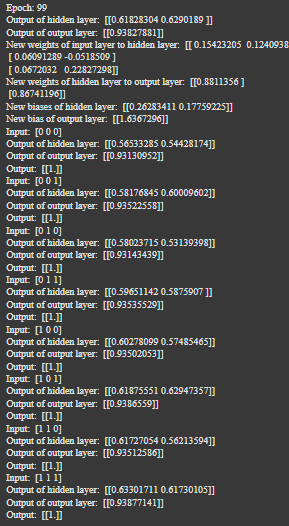
test = np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]])

for test in test:

print('Input: ',test)

print('Output: ',np.round(n.predict(test)))

**Output:**



**Discussion:**

This code implements a simple neural network with one hidden layer, using the sigmoid activation function and backpropagation for training. It includes methods for forward propagation (calculating the output of the network) and backpropagation (adjusting weights and biases based on error). The network is trained for a fixed number of epochs and can predict binary outputs (0 or 1) for given input data.