ASSIJGNMENT

Backpropagation Algorithm

Computational Intelligence Lab

NAME-SIDDHANT JAIN
BATCH-C12
ROLL NUMBER-22

Code:

```
import string
import math
import random

class Neural:
	def __init__(self, pattern):
		#
		# Lets take 2 input nodes, 3 hidden nodes and 1 output node.
		# Hence, Number of nodes in input(ni)=2, hidden(nh)=3, output(no)=1.
		#
		self.ni=3
		self.nh=3
		self.no=1

		#
		# Now we need node weights. We'll make a two dimensional array that maps node from one layer to the next.
		# i-th node of one layer to j-th node of the next.
		#
		self.wih = []
		for i in range(self.ni):
			self.wih.append([0.0]*self.nh)
```

```
                self.who = []
                for j in range(self.nh):
                        self.who.append([0.0]*self.no)

                #
                # Now that weight matrices are created, make the activation matrices.
                #
                self.ai, self.ah, self.ao = [],[],[]
                self.ai=[1.0]*self.ni
                self.ah=[1.0]*self.nh
                self.ao=[1.0]*self.no

                #
                # To ensure node weights are randomly assigned, with some bounds on values,
we pass it through randomizeMatrix()
                #
                randomizeMatrix(self.wih,-0.2,0.2)
                randomizeMatrix(self.who,-2.0,2.0)

                #
                # To incorporate momentum factor, introduce another array for the 'previous
change'.
                #
                self.cih = []
                self.cho = []
                for i in range(self.ni):
                        self.cih.append([0.0]*self.nh)
                for j in range(self.nh):
                        self.cho.append([0.0]*self.no)

        # backpropagate() takes as input, the patterns entered, the target values and the obtained
values.
        # Based on these values, it adjusts the weights so as to balance out the error.
        # Also, now we have M, N for momentum and learning factors respectively.
        def backpropagate(self, inputs, expected, output, N=0.5, M=0.1):
                # We introduce a new matrix called the deltas (error) for the two layers output and
hidden layer respectively.
                output_deltas = [0.0]*self.no
                for k in range(self.no):
                        # Error is equal to (Target value - Output value)
```

```
                error = expected[k] - output[k]
                output_deltas[k]=error*dsigmoid(self.ao[k])

        # Change weights of hidden to output layer accordingly.
        for j in range(self.nh):
                for k in range(self.no):
                        delta_weight = self.ah[j] * output_deltas[k]
                        self.who[j][k]+= M*self.cho[j][k] + N*delta_weight
                        self.cho[j][k]=delta_weight

        # Now for the hidden layer.
        hidden_deltas = [0.0]*self.nh
        for j in range(self.nh):
                # Error as given by formule is equal to the sum of (Weight from each node in hidden layer times output delta of output node)
                # Hence delta for hidden layer = sum (self.who[j][k]*output_deltas[k])
                error=0.0
                for k in range(self.no):
                        error+=self.who[j][k] * output_deltas[k]
                # now, change in node weight is given by dsigmoid() of activation of each hidden node times the error.
                hidden_deltas[j]= error * dsigmoid(self.ah[j])

        for i in range(self.ni):
                for j in range(self.nh):
                        delta_weight = hidden_deltas[j] * self.ai[i]
                        self.wih[i][j]+= M*self.cih[i][j] + N*delta_weight
                        self.cih[i][j]=delta_weight

    # Main testing function. Used after all the training and Backpropagation is completed.
    def test(self, patterns):
        for p in patterns:
                inputs = p[0]
                print ('For input:', p[0], ' Output -->', self.runNetwork(inputs), '\tTarget: ', p[1])


    # So, runNetwork was needed because, for every iteration over a pattern [] array, we need to feed the values.
    def runNetwork(self, feed):
```

```
        if(len(feed)!=self.ni-1):
            print ('Error in number of input values.')

        # First activate the ni-1 input nodes.
        for i in range(self.ni-1):
            self.ai[i]=feed[i]

        #
        # Calculate the activations of each successive layer's nodes.
        #
        for j in range(self.nh):
            sum=0.0
            for i in range(self.ni):
                sum+=self.ai[i]*self.wih[i][j]
            # self.ah[j] will be the sigmoid of sum. # sigmoid(sum)
            self.ah[j]=sigmoid(sum)

        for k in range(self.no):
            sum=0.0
            for j in range(self.nh):
                sum+=self.ah[j]*self.wih[j][k]
            # self.ah[k] will be the sigmoid of sum. # sigmoid(sum)
            self.ao[k]=sigmoid(sum)

        return self.ao


    def trainNetwork(self, pattern):
        for i in range(500):
            # Run the network for every set of input values, get the output values and
Backpropagate them.
            for p in pattern:
                # Run the network for every tuple in p.
                inputs = p[0]
                out = self.runNetwork(inputs)
                expected = p[1]
                self.backpropagate(inputs,expected,out)
        self.test(pattern)

# End of class.
```

```
def randomizeMatrix ( matrix, a, b):
        for i in range ( len (matrix) ):
                for j in range ( len (matrix[0]) ):
                        # For each of the weight matrix elements, assign a random weight
uniformly between the two bounds.
                        matrix[i][j] = random.uniform(a,b)


# Now for our function definition. Sigmoid.
def sigmoid(x):
        return 1 / (1 + math.exp(-x))


# Sigmoid function derivative.
def dsigmoid(y):
        return y * (1 - y)


def main():
        # take the input pattern as a map. Suppose we are working for AND gate.
        pat = [
                [[0,0], [0]],
                [[0,1], [1]],
                [[1,0], [1]],
                [[1,1], [1]]
        ]
        newNeural = Neural(pat)
        newNeural.trainNetwork(pat)


if __name__ == "__main__":
        main()
```

Output:

```
For input: [0, 0]  Output --> [0.17035056290080428]     Target:  [0]
For input: [0, 1]  Output --> [0.942600312146022]       Target:  [1]
For input: [1, 0]  Output --> [0.9394586242003756]      Target:  [1]
```

```
For input: [1, 1]  Output --> [0.9968820152144211]      Target:  [1]

Process returned 0 (0x0)        execution time : 0.207 s
Press any key to continue . . .
```