Dikshita Kambri

118A2044

BE EXTC A2

EXPERIMENT 4

BACK PROPAGATION NETWORKS

EXPERIMENT 4

BACK PROPAGATION NETWORKS

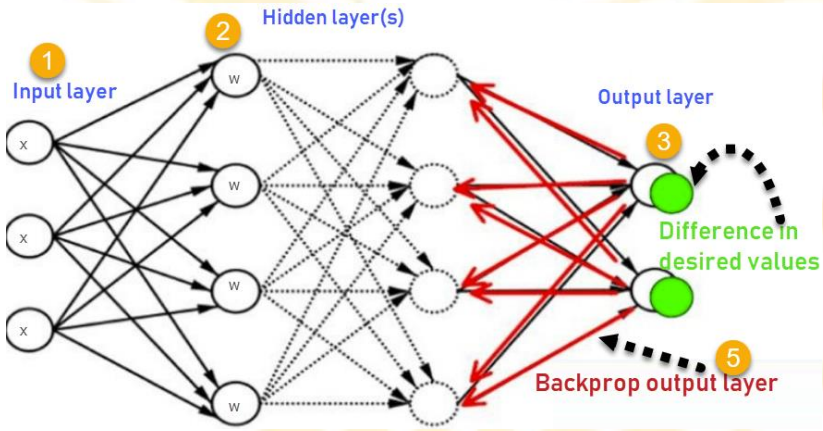
AIM: Implement back propagation neural network

APPARATUS: python

THEORY:

Back-propagation is the essence of neural net training. It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

Backpropagation is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps to calculate the gradient of a loss function with respects to all the weights in the network.



The purpose of training is to build a model that performs the **XOR** (exclusive OR) functionality with two inputs and three hidden units. Second important point is decide activation function at each node

We also need a **hypothesis function** that determines what the input to the activation function is

It is now the time to feed-forward the information from one layer to the next. This goes through two steps that happen at every node/unit in the network:

1- Getting the weighted sum of inputs of a particular unit using the h(x) function.

2- Plugging the value we get from step 1 into the activation function we have and using the activation value we get (i.e. the output of the activation function) as the input feature for the connected nodes in the next layer.

Note that units X0, X1, X2 and Z0 do not have any units connected to them and providing inputs. Therefore, the steps mentioned above do not occur in those nodes. However, for the rest of the nodes/units, this is how it all happens throughout the neural net for the first input sample in the training set:
The actual_y value comes from the training set, while the predicted_y value is what we get model yielded.

Back-propagation is all about feeding this loss backwards in such a way that we can fine-tune the weights based on which. The optimization function (Gradient Descent) will help us find the weights that will — hopefully — yield a smaller loss in the next iteration.

OUTPUT:

OOP

```
[ ]  class Rectangle:
        def __init__(self, length, breadth, unit_cost=0):
            self.length = length
            self.breadth = breadth
            self.unit_cost = unit_cost

        def get_perimeter(self):
            return 2 * (self.length + self.breadth)

        def get_area(self):
            return self.length * self.breadth

        def calculate_cost(self):
            area = self.get_area()
            return area * self.unit_cost
    # breadth = 120 cm, length = 160 cm, 1 cm^2 = Rs 2000
    r = Rectangle(160, 120, 2000)
    print("Perimeter of Rectangle: %s cm" % (r.get_perimeter())) # %s is a string formatting syntax
    print("Area of Rectangle: %s cm^2" % (r.get_area()))
    print("Cost of rectangular field: Rs. %s " %(r.calculate_cost()))
```

```
Perimeter of Rectangle: 560 cm
Area of Rectangle: 19200 cm^2
Cost of rectangular field: Rs. 38400000
Dikshita Kambri
```

```
[ ]  class Cuboid:
       def __init__(self, length, breadth, height):
         self.length = length
         self.breadth = breadth
         self.height = height

       def volume_cuboid(self):
         return self.length * self.breadth * self.height

     C = Cuboid(10, 10, 10)
     print("Volume of cuboid: %s cm^3 "% (C.volume_cuboid()))
```

```
Volume of cuboid: 1000 cm^3
Dikshita Kambri
```

Back propagation network:

Loss <= 0.12

```
import numpy as np

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


class Neural_Network():
  def __init__(self):
    #parameters
    self.inputSize = 2
    self.outputSize = 1
    self.hiddenSize = 2
    #weights
    #np.random.seed(0)
    self.W1 = np.random.randn(self.inputSize, self.hiddenSize)
    self.W2 = np.random.randn(self.hiddenSize, self.outputSize)


  def forward(self, X):

    self.zin = np.dot(X, self.W1)
    self.z = self.sigmoid(self.zin)
    self.yin = np.dot(self.z, self.W2)
```

```
    y = self.sigmoid(self.yin)
    return y

  def sigmoid(self, s):
    return 1/(1+np.exp(-s))

  def sigmoidPrime(self, s):
    return s * (1 - s)

  def backward(self, X, t, y):
    self.y_error = t - y
    self.y_delta = self.y_error*self.sigmoidPrime(y)

    self.z_error = self.y_delta.dot(self.W2.T)
    self.z_delta = self.z_error*self.sigmoidPrime(self.z)

    self.W1 += X.T.dot(self.z_delta)
    self.W2 += self.z.T.dot(self.y_delta)

NN = Neural_Network()
for i in range(1000):
  print("iteration:",i)
  print("Input: \n" + str(X))
  print("Actual Output: \n" + str(t))
  y=NN.forward(X)
  print("Predicted Output: \n" + str(y))
  Loss=np.mean(np.square(t - NN.forward(X)))
  print("Loss: \n" + str(Loss))
  if Loss<=0.12:
    break
  print("\n")
  NN.backward(X, t, y)
```

```
iteration: 56
Input:
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [1.]]
Predicted Output:
[[0.59398638]
 [0.76529767]
 [0.79261027]
 [0.84345053]]
Loss:
0.11885581213817319
```

CONCLUSION:

We studied that python is Object oriented programming language which has classes and objects. Classes are the blueprint of objects while objects are instance of classes.

Also, we observed that, back propagation is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch. Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.