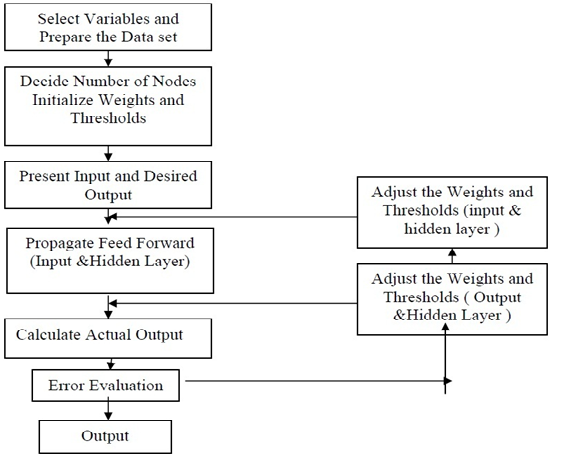
BACK PROPAGATION

**Flow chart**



# Steps :

1. Initialize the network with small random weights
2. Present an input pattern to the input layer of the network
3. Feed the input pattern forward through the network to calculate its activation value
4. Take the difference between desired output and the activation value to calculate the network’s activation error
5. Adjust the weights feeding the output neuron to reduce its activation error for this input pattern
6. Propagate an error value back to each hidden neuron that is proportional to their contribution of the network’s activation error.
7. Adjust the weights feeding each hidden neuron to reduce their contribution of error for this input pattern.
8. Repeat steps 2 to 7 for each input pattern in the input collection.
9. Repeat step 8 until the network is suitably trained.

# Implementation

First, let us create a data structure with our input data which will be analyzed later using a neural network.

**pat = [**

**[[0,0], [0]],**

**[[0,1], [1]],**

**[[1,0], [1]],**

**[[1,1], [0]]**

**]**

The goal of our neural net is to rediscover the relation between 2 inputs and the output, pretending that we know nothing about the such dependence.

A next step is to build a back propogation artificial neural network with 2 inputs, 2 hidden layers and one output.

**n = NN(2, 2, 1)**

Now we have prepared data and NN. Next step is to train the NN using the input data. It should be noted that it is desirable to rescale the input values so they will be between [-1,1].

**n.train(pat)**

And then the data is tested.

**n.test(pat)**

# Functions

calculating a random number where: a <= rand < b

**def rand(a, b):**

**return (b-a)\*random.random() + a**

The sigmoid function, tanh is easier to compute than the standard 1/(1+e^-x)

**def sigmoid(x):**

**return math.tanh(x)**

The activations for nodes

**self.ai = [1.0]\*self.ni**

**self.ah = [1.0]\*self.nh**

**self.ao = [1.0]\*self.no**

Creating the weights

**self.wi = makeMatrix(self.ni, self.nh)**

**self.wo = makeMatrix(self.nh, self.no)**

Setting the weights to random values

**for i in range(self.ni):**

**for j in range(self.nh):**

**self.wi[i][j] = rand(-0.2, 0.2)**

**for j in range(self.nh):**

**for k in range(self.no):**

**self.wo[j][k] = rand(-2.0, 2.0)**

Function that does the backpropagation

**def backPropagate(self, targets, N, M):**

**if len(targets) != self.no:**

**raise ValueError('wrong number of target values')**

calculating the error terms for output

**output\_deltas = [0.0] \* self.no**

**for k in range(self.no):**

**error = targets[k]-self.ao[k]**

**output\_deltas[k] = dsigmoid(self.ao[k]) \* error**

Calculating the error terms for hidden nodes

**hidden\_deltas = [0.0] \* self.nh**

**for j in range(self.nh):**

**error = 0.0**

**for k in range(self.no):**

**error = error + output\_deltas[k]\*self.wo[j][k]**

**hidden\_deltas[j] = dsigmoid(self.ah[j]) \* error**

updating the output weights

**for j in range(self.nh):**

**for k in range(self.no):**

**change = output\_deltas[k]\*self.ah[j]**

**self.wo[j][k] = self.wo[j][k] + N\*change + M\*self.co[j][k]**

**self.co[j][k] = change**

**#print N\*change, M\*self.co[j][k]**

updating the input weights

**for i in range(self.ni):**

**for j in range(self.nh):**

**change = hidden\_deltas[j]\*self.ai[i]**

**self.wi[i][j] = self.wi[i][j] + N\*change + M\*self.ci[i][j]**

**self.ci[i][j] = change**

calculation of the error

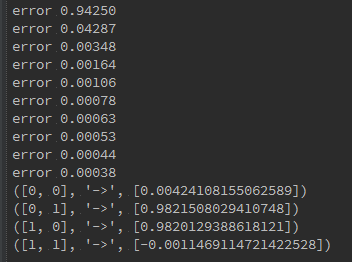
**error = 0.0**

**for k in range(len(targets)):**

**error = error + 0.5\*(targets[k]-self.ao[k])\*\*2**

**return error**

# Output



The output of this scripts gives the training error, and the last lines are the predictions after the training.