backpropogation

In [ ]:
```
import numpy as np

class ANN:
    def __init__(self,input_size,hidden_layers,hidden_neurons,output_size,learning_rate):
        self.weights=[]
        self.bias=[]
        self.hidden_layers=hidden_layers
        self.learning_rate=learning_rate;

        for i in range(hidden_layers+1):
            if i==0:
                self.weights.append(np.random.randn(hidden_neurons,input_size))
                self.bias.append(np.full((hidden_neurons,1),1))
            elif i==hidden_layers:
                self.weights.append(np.random.randn(output_size,hidden_neurons))
                self.bias.append(np.full((output_size,1),1))
            else:
                self.weights.append(np.random.randn(hidden_neurons,hidden_neurons))
                self.bias.append(np.full((hidden_neurons,1),1))

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

    def first_order_sigmoid(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def forward(self,x):
        activations=[]
        activations.append(x)
        for i in range(self.hidden_layers+1):
            x=np.dot(self.weights[i],activations[i])+self.bias[i]
            activations.append(self.sigmoid(x))
        return activations

    def backward(self,activations,di,m):
        delta=(activations[-1]-di.T) * self.first_order_sigmoid(np.dot(self.weights[-1],activations[-2])+self.bias[-1])
        for i in range(self.hidden_layers,-1,-1):
            if i==self.hidden_layers:
                prev=np.array(self.weights[i])
                self.weights[i]=self.weights[i]-(self.learning_rate/m) * np.dot(delta,activations[i].T)
                self.bias[i]=self.bias[i]-(self.learning_rate/m) * np.sum(delta,axis=1,keepdims=True)
            else:
                delta=np.dot(prev.T, delta) * self.first_order_sigmoid(np.dot(self.weights[i], activations[i])+self.bias[i])
                prev=np.array(self.weights[i])
                self.weights[i]=self.weights[i]-(self.learning_rate/m) * np.dot(delta,activations[i].T)
                self.bias[i]=self.bias[i]-(self.learning_rate/m) * np.sum(delta,axis=1,keepdims=True)

    def train(self,x,y,epochs):
        for i in range(epochs):
            activations=self.forward(x)
            m=x.shape[1]
            self.backward(activations,y,m)
            if(i%1000==0):
                print("Error at %d epoch : "%(i),np.sum(activations[-1]-y.T))

    def predict(self,x):
        predictions=[]
        for input in x:
            prediction = self.forward(np.array(input))
            predictions.append(prediction[-1])
        return predictions
```

In [12]:
```
import numpy as np

class ANN:
    def __init__(self,input_size,hidden_layers,hidden_neurons,output_size,learning_rate):
        self.weights=[]
        self.bias=[]
        self.learning_rate=learning_rate
        self.hidden_layers=hidden_layers;

        for i in range(hidden_layers+1):
            if i==0:
                self.weights.append(np.random.randn(hidden_neurons,input_size))
                self.bias.append(np.full((hidden_neurons,1),1))
            elif i==hidden_layers:
                self.weights.append(np.random.randn(output_size,hidden_neurons))
                self.bias.append(np.full((output_size,1),1))
            else:
                self.weights.append(np.random.randn(hidden_neurons,hidden_neurons))
                self.bias.append(np.full((hidden_neurons,1),1))

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