**import** numpy **as** np   
   
**from** scipy **import** optimize  
  
  
**class** Neural\_Network(object):  
 **def** \_\_init\_\_(self):  
   
 self.inputLayerSize = 2  
 self.outputLayerSize = 1  
 self.hiddenLayerSize = 3  
   
 self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)  
 self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)  
   
 **def** forward(self, X):  
   
 self.z2 = np.dot(X, self.W1)  
 self.a2 = self.sigmoid(self.z2)  
 self.z3 = np.dot(self.a2, self.W2)  
 yHat = self.sigmoid(self.z3)   
 **return** yHat  
   
 **def** sigmoid(self, z):  
   
 **return** 1/(1+np.exp(-z))  
   
 **def** sigmoidPrime(self,z):  
   
 **return** np.exp(-z)/((1+np.exp(-z))\*\*2)  
   
 **def** costFunction(self, X, y):  
   
 self.yHat = self.forward(X)  
 J = 0.5\*sum((y-self.yHat)\*\*2)  
 **return** J  
   
 **def** costFunctionPrime(self, X, y):  
 self.yHat = self.forward(X)  
   
 delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))  
 dJdW2 = np.dot(self.a2.T, delta3)  
   
 delta2 = np.dot(delta3, self.W2.T)\*self.sigmoidPrime(self.z2)  
 dJdW1 = np.dot(X.T, delta2)   
   
 **return** dJdW1, dJdW2  
   
 **def** getParams(self):  
 params = np.concatenate((self.W1.ravel(), self.W2.ravel()))  
 **return** params  
   
 **def** setParams(self, params):  
 W1\_start = 0  
 W1\_end = self.hiddenLayerSize \* self.inputLayerSize  
 self.W1 = np.reshape(params[W1\_start:W1\_end], (self.inputLayerSize , self.hiddenLayerSize))  
 W2\_end = W1\_end + self.hiddenLayerSize\*self.outputLayerSize  
 self.W2 = np.reshape(params[W1\_end:W2\_end], (self.hiddenLayerSize, self.outputLayerSize))  
   
 **def** computeGradients(self, X, y):  
 dJdW1, dJdW2 = self.costFunctionPrime(X, y)  
 **return** np.concatenate((dJdW1.ravel(), dJdW2.ravel()))  
   
  
  
  
**class** trainer(object):  
 **def** \_\_init\_\_(self, N):  
 self.N = N  
   
 **def** callbackF(self, params):  
 self.N.setParams(params)  
 self.J.append(self.N.costFunction(self.X, self.y))   
   
 **def** costFunctionWrapper(self, params, X, y):  
 self.N.setParams(params)  
 cost = self.N.costFunction(X, y)  
 grad = self.N.computeGradients(X,y)  
   
 **return** cost, grad  
   
 **def** train(self, X, y):  
 self.X = X  
 self.y = y  
  
 self.J = []  
   
 params0 = self.N.getParams()  
  
 options = {**'maxiter'**: 200, **'disp'** : **True**}  
 \_res = optimize.minimize(self.costFunctionWrapper, params0, jac=**True**, method=**'BFGS'**, \  
 args=(X, y), options=options, callback=self.callbackF)  
  
 self.N.setParams(\_res.x)  
 self.optimizationResults = \_res  
  
  
  
  
  
  
  
X = np.array(([3,5] , [5,1] , [10,2]) , dtype = float)  
y = np.array(([90],[10],[30]) , dtype = float)  
  
X = X/np.amax(X , axis=0)  
y=y/100  
  
print(y)  
nn = Neural\_Network()  
  
cost1=nn.costFunction(X,y);  
dJdW1, dJdW2=nn.costFunctionPrime(X,y);  
  
print();  
*#print(dJdW1);*print(**"\n\n"**)  
  
nn.W1=nn.W1-3\*dJdW1;  
nn.W2=nn.W2-3\*dJdW2;  
  
  
*#nn.W1=nn.W1-3\*dJdW1;  
#nn.W2=nn.W2-3\*dJdW2;*cost2=nn.costFunction(X,y);  
  
  
  
print(cost1);  
print(**"now change!!!"**);  
print(cost2);  
print(**'\n'**)  
print(nn.forward(X))  
  
  
  
  
  
  
T=trainer(nn)  
T.train(X , y)  
p=nn.forward(X)  
print(p)