Semih Akbayrak

# EE550 Report

## HW3

In this homework, we are expected to design a multilayer perceptron structure which will create layers and neurons according to the user's desires.

First of all to use in hidden layers, I defined two nonlinear function, sigmoid and its derivative.

```
#Nonlinear function sigmoid and its derivative
def sigmoid(x):
    x = np.array(x)
    return 1/(1+np.exp(-x))

def drvsigmoid(x):
    x = np.array(x)
    return sigmoid(x)*(1-sigmoid(x))
```

Then I started to design MLP class, this class can be used to create an MLP object with different number of neurons and layers. It accepts array as an input, as an example: if you would like to create an MLP with 1 input layer, 1 output layer and 2 hidden layers with 2 neurons for input, 9 and 10 neurons for layers, 1 neuron for output layer, then you need to use [2,9,10,1] as the input. This class firstly starts to create weight data structure with appropriate indexing and initialize these weights.

```
#MLP class which includes several functions
class MLP:
    #structurel function of MLP which is used to to create new MLP object
    def __init__(self, layers):
        self.layers = layers
        self.weights = []
        #weight array starts to be created by for loops
        for i in range(len(self.layers)-1):
            self.weights.append([])
            for j in range(self.layers[i]):
                self.weights[i].append([])
                for k in range(self.layers[i+1]):
```

```
                r = (2*np.random.random()-1)
                self.weights[i][j].append(r) #initial weights specified
```

In order to design forward propagation, I defined another function. It takes an array like [[0,1],[1]] which includes both input and expected output. First layer's neurons don't do anything, they just let inputs to go as the outputs of first layer. Then by multiplying them with convenient weights and by using sigmoid to compute the hidden neurons' outputs, the algorithm comes to output layer. In the beginning, I designed output layer's neurons as nonlinear function as well, but I couldn't get the correct result. So in the last layer, I just multiply outputs with weights and add them up without putting them to sigmoid function.

```
    #starts from input layer and propagate to output layer
    def propagate_forward(self,X):
        self.outputs = []
        self.inputs = []
        self.inputs.append(X[0])
        self.outputs.append(self.inputs[0]) #output of first layer, simply equal to input
        for i in range(len(self.layers)-1):
            self.inputs.append([])
            for j in range(self.layers[i+1]):
                add = 0
                for k in range(self.layers[i]):
                    add = add + self.outputs[i][k]*self.weights[i][k][j]
                self.inputs[i+1].append(add) #net input to a neuron
            if i == len(self.layers)-1:
                self.outputs.append(self.inputs[i+1]) #output of last layer, which is a linear function
            else:
                self.outputs.append(sigmoid(self.inputs[i+1])) #output of hidden layers acquired by using nonlinear sigmoid
```

Now, let me start to tell about the learning part. In this part, the algorithm propagate backward and by using the datas from the layers it propagated, it updates the weights. Apart from other layers, firstly I compute the sigma belongs to output layer. Because I can compute the cost function in there directly. Then by using this data, I compute other sigmas as well. It needs to be noticed that, I designed sigmas array by starting from the last layer so its element order is different from the weight, output and input arrays.

```
    #This is where learning starts and weights update themselves
    def propagate_backward(self,X,nuu=0.4):
        self.expected = np.array(X[1])
        self.out = np.array(self.outputs[len(self.layers)-1])
        drv = np.array(drvsigmoid(self.inputs[len(self.layers)-1]))
        self.error = (self.expected - self.out)
        sigma = self.error*drv #sigma belongs to last layer
        self.sigmas = []
        self.sigmas.append(sigma)
        #loop to create sigma array
        for i in range(len(self.layers)-1):
            self.sigmas.append([])
            p = len(self.layers)-(i+2)
            for j in range(self.layers[p]):
                add = 0
                for k in range(self.layers[len(self.layers)-(i+1)]):
                    add = add + self.sigmas[i][k]*self.weights[p][j][k]
                add = add*drvsigmoid(self.inputs[p][j])
                self.sigmas[i+1].append(add)
        self.deltaw = []
        #loop to create deltaw array, change in weights
        for i in range(len(self.layers)-1):
            j = len(self.layers)-(i+2)
            s = np.array(self.sigmas[i])
            sm = np.mat(s)
            o = np.array(self.outputs[j])
            om = np.mat(o)
            wm = om.T*sm
            wm = np.array(wm)
            wm = nuu*wm
            self.deltaw.append(wm)
        for i in range(len(self.layers)-1):
            p = len(self.layers)-(i+2)
            for j in range(self.layers[p]):
                self.weights[p][j] = self.weights[p][j] + self.deltaw[i][j]
```

In the taining part, the system uses both forward propagation and backward propagation, respectively. To complete one epoch, it uses all the sample patterns and updates weights. For XOR we have 4 sample patterns, and let's say system reached the desired error in 3000 epoch, this means that the algorithm forward and backward propagated 12000 times to reach desired level. To increase the convergence I added momentum term to weight changes. As well as, I used adaptive parameters for the appropriate update direction.

```
    #function for training
    def train(self,X,epochs=10000,nuu=0.4,muu=0.2,error=0.01):
        self.errorlist = []
        for i in range(epochs):
            print i
            self.errorfunc = 0
            for j in range(len(X)):
                self.propagate_forward(X[j])
                self.propagate_backward(X[j])
                for k in range(len(self.layers)-1):
                    for l in range(self.layers[len(self.layers)-(k+2)]):
                        self.deltaw[k][l] = self.deltaw[k][l] + muu*self.deltaw[k][l] #momentum term
                self.errorfunc = (self.errorfunc + (np.array(self.error))**2)
            self.errorlist.append(self.errorfunc)
            print self.errorlist[i]
            count = 0
            zerocount = 0
            if i>0:
                for j in range(self.layers[len(self.layers)-1]):
                    if (self.errorlist[i]-self.errorlist[i-1])[j]<0:
                        count = count + 1
                    elif (self.errorlist[i]-self.errorlist[i-1])[j]==0:
                        zerocount = zerocount + 1
                #Adaptive parameters
                if count==self.layers[len(self.layers)-1]:
                    nuu = nuu + 0.005
                else:
                    nuu = nuu - 0.01*nuu
                if zerocount==self.layers[len(self.layers)-1]:
                    break
            errorcount = 0
            for j in range(self.layers[len(self.layers)-1]):
                    if self.errorfunc[j]<error:
                        errorcount = errorcount + 1
            if errorcount == self.layers[len(self.layers)-1]:
                plt.figure()
                plt.plot(self.errorlist)
                plt.show()
                break
```

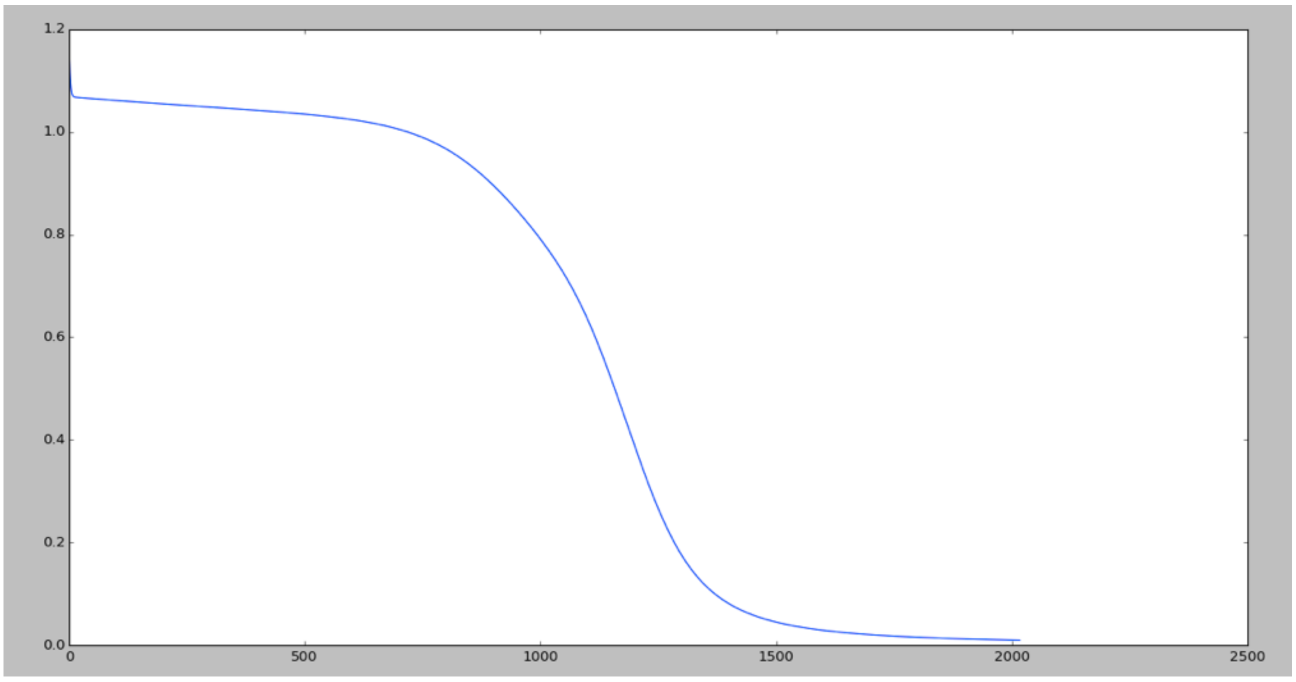
## Below function is for testing

```
#function for testing, very similar to forward propagating
    def test(self,X):
        self.outputs = []
        self.inputs = []
        self.inputs.append(X[0])
        self.outputs.append(self.inputs[0])
        for i in range(len(self.layers)-1):
            self.inputs.append([])
            for j in range(self.layers[i+1]):
                add = 0
                for k in range(self.layers[i]):
                    add = add + self.outputs[i][k]*self.weights[i][k][j]
                self.inputs[i+1].append(add)
            if i == len(self.layers)-1:
                self.outputs.append(self.inputs[i+1])
            else:
                self.outputs.append(sigmoid(self.inputs[i+1]))
        print self.inputs[0]
        print self.outputs[len(self.layers)-1]
```

By using the code segment below, I get the result for XOR, and the test results are very satisfactory

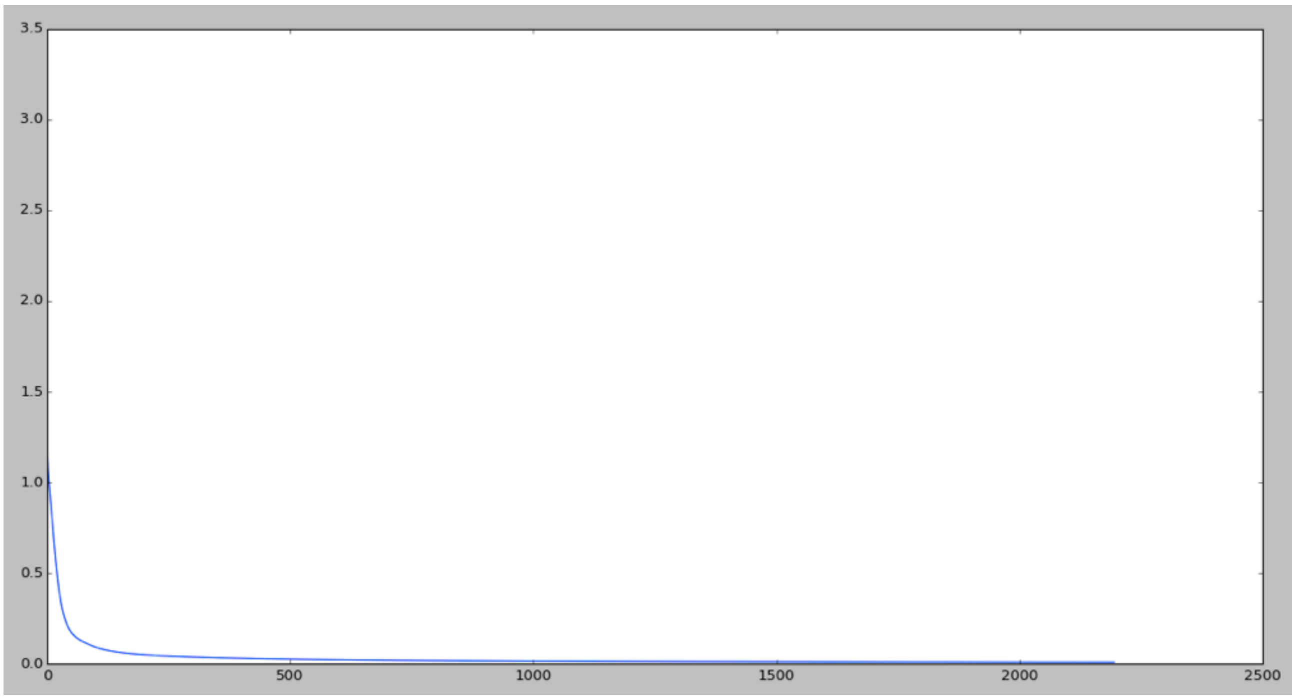
```
mlp = MLP([2,8,9,1])
X = [[[0,0],[0]],
[[0,1],[1]],
[[1,0],[1]],
[[1,1],[0]]]
mlp.train(X)
print 'Training is over'
mlp.test([[0,0]])
mlp.test([[0,1]])
mlp.test([[1,0]])
mlp.test([[1,1]])
```

And corresponding error plot below, with x-axis number of epochs and y-axis error.



It was also wanted from us to use MLP for f(x) = 1/x, and the below code and plot belongs to that function. You can see in there I randomly specify the the numbers for training and testing. And as it was said, I use 85 points for training and 15 for testing.

```
mlp2 = MLP([1,10,1])
X = []
my_list = list(xrange(1,101))
random.shuffle(my_list)
for i in range (85):
    X.append([])
    for j in range(2):
        X[i].append([])
        if j ==0:
            X[i][j].append(my_list[i])
        else:
            X[i][j].append(1.0/my_list[i])
mlp2.train(X)
A = []
for i in range (100):
    A.append([])
    for j in range(2):
        A[i].append([])
        if j ==0:
            A[i][j].append(my_list[i])
        else:
            A[i][j].append(1.0/my_list[i])
print 'Training is over'
for i in range (15):
    j = i+85
    mlp2.test(A[j])
```



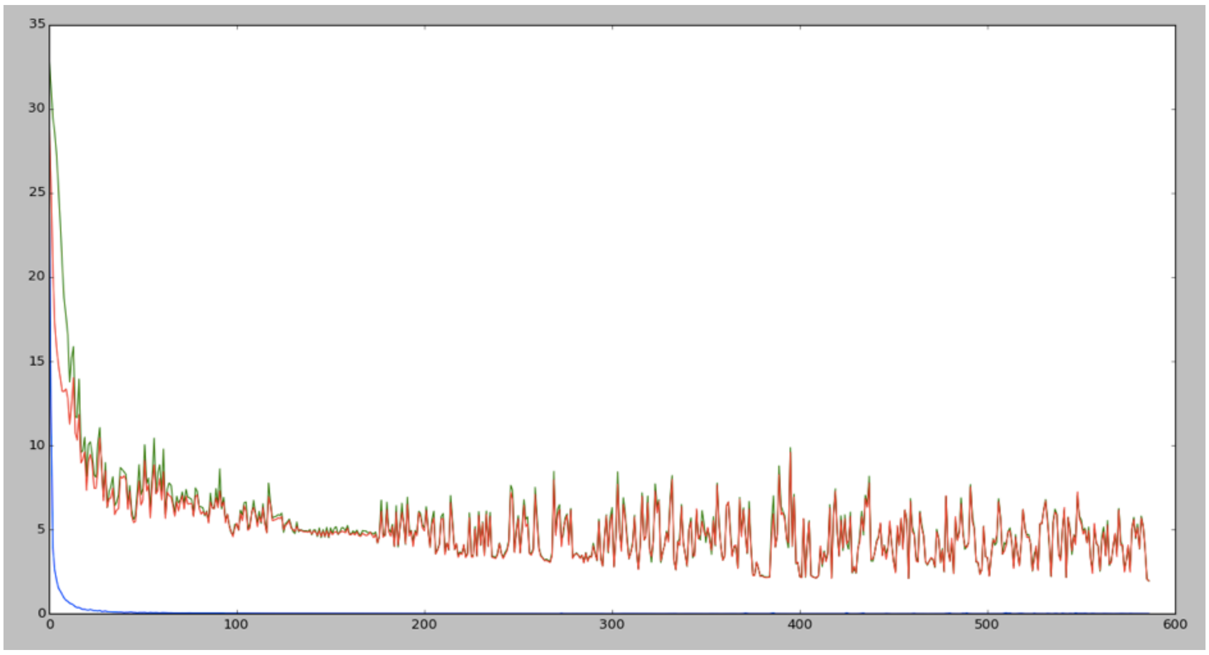
In the iris data set, I made some kind of changes to transform the dataset appropriate to my function parameters and also I tagged Iris-setosa with [1,0,0], Iris-versicolor with [0,1,0] and Iris-virginica with [0,0,1]. Therefore my inputs to train function became

```
X = [[[5.1,3.5,1.4,0.2],[1,0,0]],
[[4.9,3.0,1.4,0.2],[1,0,0]],
[[4.7,3.2,1.3,0.2],[1,0,0]],
[[4.6,3.1,1.5,0.2],[1,0,0]],
[[5.0,3.6,1.4,0.2],[1,0,0]],
[[5.4,3.9,1.7,0.4],[1,0,0]],
…..
…..
…..]
```

As it is wanted, I use 95% of them for training and 5% of them for testing. Again test results are very satisfactory, the algorithm tags the flowers correctly. This time I specify error as 2, because the number of samples is huge, 2 is enough for correct tagging and to find error below 2, algorithm needs much more epochs.

```
mlp3 = MLP([4,9,10,3])
X = [[[5.1,3.5,1.4,0.2],[1,0,0]],
[[4.9,3.0,1.4,0.2],[1,0,0]],
[[4.7,3.2,1.3,0.2],[1,0,0]],
[[4.6,3.1,1.5,0.2],[1,0,0]],
[[5.0,3.6,1.4,0.2],[1,0,0]],
[[5.4,3.9,1.7,0.4],[1,0,0]],
…..
…..
…..]
my_list = list(xrange(0,150))
random.shuffle(my_list)
TS = []
for i in range(142):
  TS.append(X[my_list[i]])

mlp3.train(TS,10000,0.4,0.2,2)
print 'Training is over'
A = []
for i in range(8):
  A .append(X[my_list[i+142]])
  mlp3.test(A[i])
```



This time, because my output layer consists 3 neurons, I got 3 different error function and their values in the plot is above.