EE550 Implementation of Multilayer Perceptron Model with Backpropagation Algorithm Using Python

April 13, 2017

Mine Melodi Çalışkan - 2015705009

Multilayer Perceptron is a modification of the single perceptron which can separate data that are not linearly separable.

It has layers between inputs and outputs.Given data flows forward and training is done with the backpropagation algorithm.

In the feedforward algorithm the information moves only one direction as the sum of the multiplication of the neurons and the weights connecting the next layer calculated, it passes through the activation function to determine the output of the layer

BACKPROPAGATION ALGORITHM

This algorithm looks for the minimum of the error function in weight space using the method of gradient descent.

For given a training data set of input vector x and target output vector t , the algorithm back propagates the error by weighting it by the weights in the previous layer and the gradients of the associated activation functions.

After the back propagation the parameters are updated by using the calculated gradients.

The Activation Function

The activation function \$f \$ is non-linear, differentiable and bounded. In this implementation Sigmoid Activation Function is used.

$$\text{Sigmoid activation function: } f(x) = \frac{1}{1+e^{-x}}$$

$$\text{Derivative of Sigmoid activation function: } f'(x) = (\frac{1}{1+e^{-x}})(1 - \frac{1}{1+e^{-x}}) = f(x)(1-f(x))$$

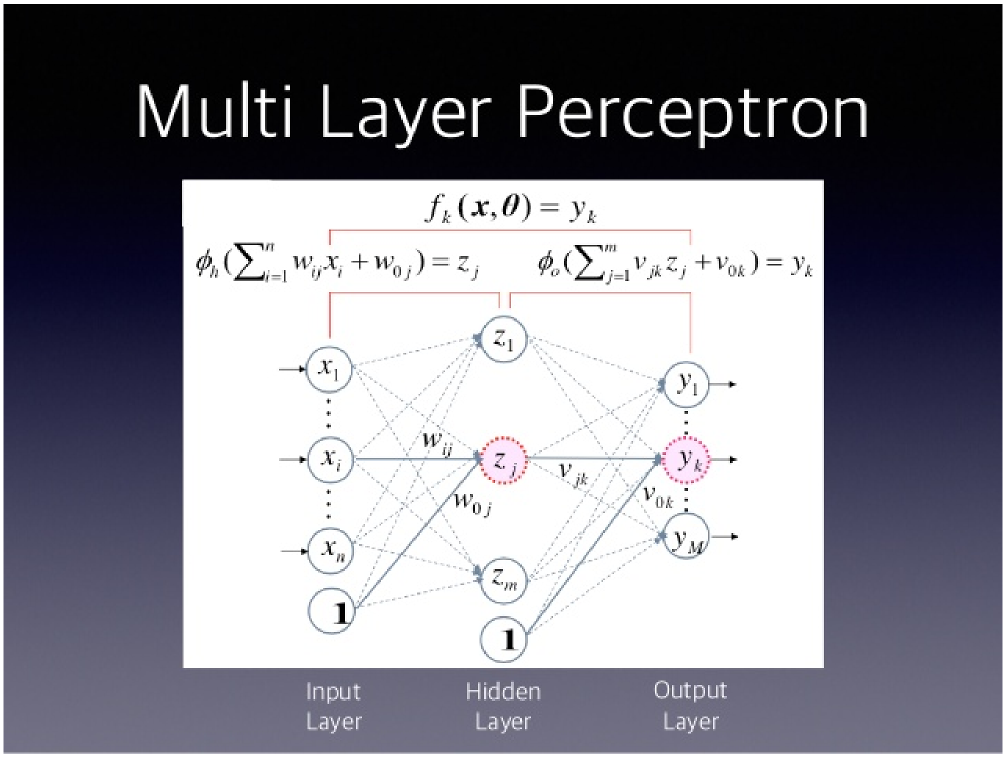
The Error Function

$$E = \frac{1}{2}\sum_{k \in K}(O_k - t_k)^2$$

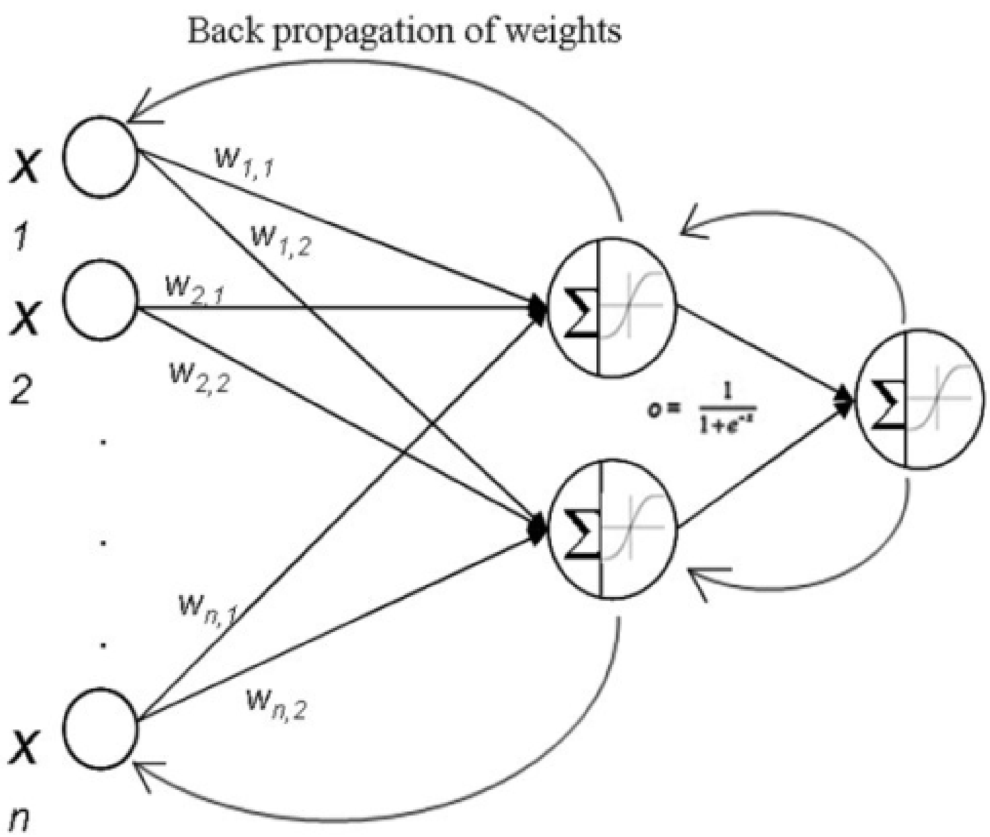
where t_k is the target value of node k and O_k is the output value of node k which is obtained by weighted input value and activation function.

To minimize the error, the best combination of weights should be found and for this purpose it's rate of change with respect to given connective weights should be determined. There exists two parts of the gradient computations depend on the layers.

The cost function at the Output Layer (with index k+1):



MLP



BackPropagation

$$E^{k+1} = \frac{1}{2}\sum(t_j - {O_j}^{k+1})^2$$

t_j : Desired signals at the Output Layer${O_j}^{k+1}$: Actual output at the Output Layer ${O_j}^{k+1} = \gamma(w_1, w_2, \ldots, w_h, \vec{x}, \vec{t}$

In general, we deploy a single gradient descent rule to minimize such cost function.

$$\Delta\vec{w} = -\eta\frac{\partial E^{k+1}}{\partial\vec{w}} \text{ Here } \vec{w} \text{ involves all the weights.}$$

The update rule for all the weights (output weights and the hidden layer weights) can be written as:

$$\Delta {w_{ij}}^k = \eta {\delta_j}^{k+1} {O_i}^k \qquad (6)$$

$$\text{which is actually equal to: } \Delta {w_{ij}}^k = -\eta\frac{\partial E^{k+1}}{\partial {w_{ij}}^k}$$

$$\text{Let } {S_j}^{k+1} = \sum {w_{ij}}^k {O_i}^k$$

1) If we are at Output Layer:

$${\delta_j}^{k+1} = (t_j - {O_j}^{k+1}) f'({S_j}^{k+1}) \qquad (1)$$

** 2) If we are at any Hidden Layer (of index k+1) : **

$${\delta_j}^{k+1} = (\sum {\delta_l}^{k+2} {w_{jl}}^{k+1}) f'({S_j}^{k+1}) \qquad (2)$$

Local Minimum Problem

There is no guarantee of the convergence of the backpropagation algorithm.

It may get stuck at a local minimum point. To prevent this issue, we can apply following tuning methods.

Momentum term for update rule

$$\{\Delta w_i\}_r = -\eta\nabla E^{k+1} + \mu\{\Delta w_i\}_{r-1} \qquad (3)$$

where "r" is the iteration number in the training process and μ is momentum coefficient.

$\{\Delta w_i\}_{r-1}$ is the momentum term which basically determines the impact of past changes.

Then the effective learning rate can be made large without divergent oscillations.

Learning rate update rule with threshold

$$\text{If } \|\nabla E\| < \epsilon \qquad (4) \quad \text{then, do not update learning rate.}$$

$$\text{Else} \quad \Delta\eta = \begin{cases} \gamma & \nabla E < 0 \\ -\beta\eta & \nabla E > 0 \end{cases} \qquad (5)$$

In [16]:

```python
import numpy as np

class MultiLayerPerceptron:

    def __init__(self, network_size):

        """Initialize the network

        network_size=(n_input,n_hidden1,...,n_hiddenk, n_output)

        n_input: number of neurons in input layer

        n_hiddenj: number of hidden neurons in hidden layer j
                   where j=1,2,..k

        n_output:number of output neurons
        """

        self.indices=0
        self.shape=None
        self.weights=[]



        #set layer values
        self.indices = len(network_size) - 1
        self.shape = network_size


        #to store inputs and outputs after forward propagation
        self._S = []
        self._O = []

        #to store previous weight changes for momentum term
        self.prev_weight_change = []


        #Initialize weights

        layer_array=np.array([network_size[:-1], network_size[1:]]).T
        for (layerpair_1,layerpair_2) in layer_array:

            self.wi=np.zeros((layerpair_2,layerpair_1+1))

            for i in range(layerpair_2):
                for j in range(layerpair_1+1):
                    self.wi[i][j]=np.random.uniform(-1, 1)
```

```
            self.weights.append(self.wi)
            self.prev_weight_change.append(np.zeros((layerpair_2
                                                    ,layerpair_1+1)))


    #Forward Propagation
    def FeedForward(self, input):
        """Feed the network with inputs"""

        #Reset values
        self._S = []
        self._O = []

        #Feedforward
        for k in range(self.indices):

            # Determine layer inputs

            #if we are at the input layer
            if k == 0:
                #we also add bias
                input_with_bias=np.array([np.append(i,1) for i in input])
                S = self.weights[0].dot(input_with_bias.T)

            #else we are the hidden layer
            else:
                #we take the data from previous layer
                #hidden_input_with_bias
                b=np.ones([1, input.shape[0]])
                S = self.weights[k].dot(np.vstack([self._O[-1],b]))

            #layer inputs
            self._S.append(S)

            #layer outputs
            self._O.append(self.sigmoid(S))

        #return output from the last layer
        return self._O[-1].T


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

    #Derivative of Sigmoid
    def sigmoid_derivative(self,x):
            output = self.sigmoid(x)
```

```
            return output * (1 - output)


    #Backpropagation
    def BackPropagation(self, input, target, eta,momentum_coef):
        """
        Backpropagate the network for one epoch

        eta:learning rate
        momentum_coef: momentum coefficient

        """
        #to store deltas in Equation (1) and (2)
        delta = []

        # FeedForward the network
        self.FeedForward(input)

        #Compute deltas
        #start from Output Layer and move backwards
        for k in range(self.indices)[::-1]:

            #if we are at Output Layer
            if k== self.indices - 1:
                e= self._O[k]-target.T
                #Equation (1)
                output_delta=e*self.sigmoid_derivative(self._S[k])
                error = 0.5*np.sum(e**2)
                delta.append(output_delta)

            #else we are at hidden layer
            else:

                # delta_h--> following layer's delta
                delta_h = self.weights[k + 1].T.dot(delta[-1])
                f_deriv_S=self.sigmoid_derivative(self._S[k])
                #Equation (2)
                #takes all the but last rows that correspond to biases
                hidden_delta=delta_h[:-1, :]*f_deriv_S
                delta.append(hidden_delta)

        #Compute weight changes
        for k in range(self.indices):

            '''
            *get outputs of the layers

            *multiply all the outputs from previous layer
```

```
        by all of the deltas from the current layer

        *update the weights that connect
        previous layer to the current layer

        *return error
        '''

        if k == 0:
            # if we are in input layer
            #add biases also
            input_with_bias=np.array([np.append(i,1) for i in input])
            O= input_with_bias.T

        else:
            #output for previous layer
            #add biases also
            b=np.ones([1, self._O[k - 1].shape[1]])
            O = np.vstack([self._O[k - 1],b])



        #adapt index of delta for reverse order
        k_delta = self.indices - 1 - k


        #Equation (6)

        #take current deltas and multiply it
        #with previous layers' outputs
        delta_x_O=delta[k_delta][np.newaxis,:,:].transpose(2, 1, 0)\
                                      * O[np.newaxis,:,:].transpose(2, 0
        Delta_w_current=eta*np.sum(delta_x_O, axis = 0)


        momentum_effect= momentum_coef * self.prev_weight_change[k]

        #Equation(3)
        #update the weights
        Delta_w = Delta_w_current + momentum_effect

        self.weights[k] -= eta*Delta_w

        self.prev_weight_change[k] = Delta_w


    #returns error
    return error
```

```python
def train(self, patterns, epochs, eta, mu):

    #eta: learning rate
    #mu: momentum coefficient


    import pylab
    E=np.zeros(epochs)
    etas = []
    etas.append(eta)

    c=[]
    epoch=[]

    for n in range(1,epochs):
        cost = 0.0
        inputs=[]
        targets=[]
        for p in patterns:
            inputs.append(p[0])
            targets.append(p[1])

        inputs=np.array(inputs)
        targets=np.array(targets)

        #cost=self.BackPropagation(inputs,targets, eta, mu)

        #Update rule for Learning Rate "eta"
        E[0]=self.BackPropagation(inputs,targets, eta, mu)
        E_new=self.BackPropagation(inputs,targets, eta, mu)
        epsilon=0.0001
        if not abs(E_new-E[n-1])<epsilon: #Equation (4)
            #Equation (5)
            if E_new > E[n-1]:
                # Decrease learning rate
                eta = eta * 0.5
                E_new=self.BackPropagation(inputs,targets, eta, mu)

            elif E_new < E[n-1]:
                #Increase learning rate
                eta = eta * 1.05

        etas.append(eta)
        E[n] = E_new
```

```
            cost =cost +self.BackPropagation(inputs,targets, eta, mu)
            c.append(cost)
            epoch.append(n)
            threshold=0.01

            #terminate if cost is less then the threshold=0.01
            if cost<threshold:
                break

        #print learning rate list
        #print etas

        #Plot the cost function value vs the number of epochs
        pylab.plot(epoch, c)
        pylab.xlabel('Number of Epochs')
        pylab.ylabel('Cost')
        pylab.show()

    def test(self, patterns,plot=False,input_index=False):
        inputs=[]
        targets=[]
        outputs=[]

        #get inputs and targets in given patterns
        for p in patterns:
            inputs.append(p[0])
            targets.append(p[1])

        inputs=np.array(inputs)
        targets=np.array(targets)
        #print inputs
        if not input_index:
            for i in range(len(inputs)):

                print "Input:",inputs[i],'->',\
                "Desired:",targets[i],',',\
                "Output:",self.FeedForward(inputs)[i]

                outputs.append(self.FeedForward(inputs)[i])
        else:
            for i in range(len(inputs)):

                print "Input:",i,'->',\
                "Desired:",targets[i],',',\
                "Output:",self.FeedForward(inputs)[i]
```

```
                outputs.append(self.FeedForward(inputs)[i])


        #plotting
        if plot:
            import matplotlib.pyplot as plt
            y=np.amax(inputs)
            x=np.linspace(0, y, len(targets))
            plt.scatter(x,targets, c='b')
            plt.scatter(x,outputs, c='r')
            plt.title('Comparison of Results')
            plt.show()
```

Remark: In the following examples the number of layers and the number of neurons in each layer can be changed.

0.1 Test the model with XOR function

0.1.1 Plot cost function vs epochs.

0.1.2 Terminate the update rule if the cost function reaches a certain threshold (i.e 0.01)

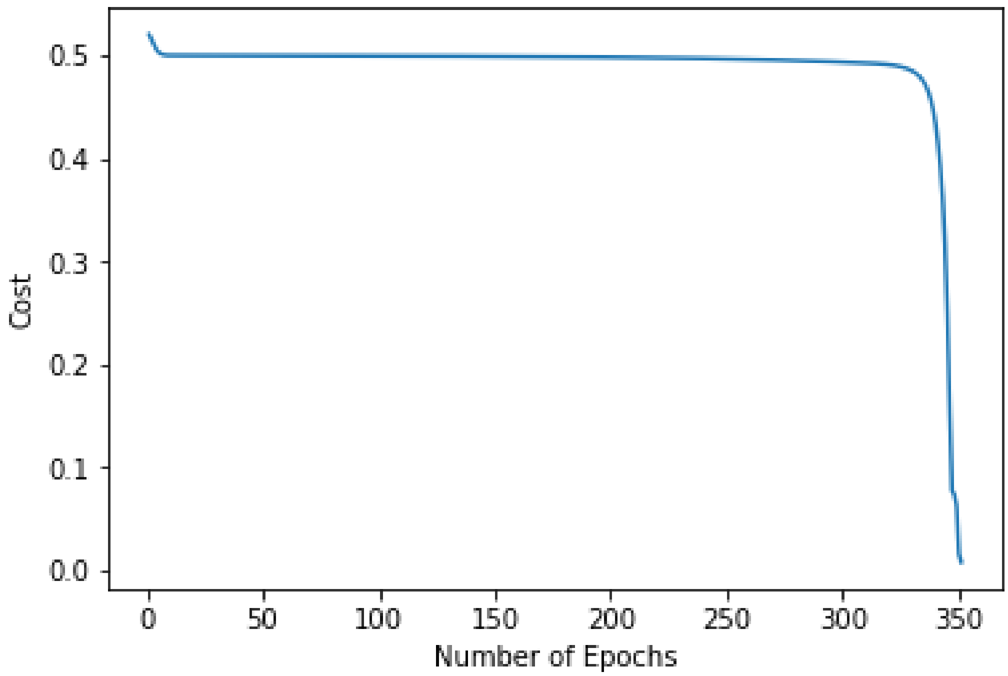
0.1.3 Check the model with 4 inputs and write the output for each input.

```
In [17]: def XOR():
             XOR_= [
                 [[0,0], [0]],
                 [[0,1], [1]],
                 [[1,0], [1]],
                 [[1,1], [0]]
             ]


             # Multilayer Perceptron model with:
             #2 input neurons
             #2 hidden layers with 3 neurons
             #1 output neuron

             network_form2=(2,3,3,1)
             MLP2=MultiLayerPerceptron(network_form2)
             print "Performance of the MLP with 2 hidden layers with 3 neurons"
             eta=0.2
             mu=0.7
             epochs=100000
             #train
             MLP2.train(XOR_,epochs,eta,mu)
             #test
             MLP2.test(XOR_)
```

Remark In the results we expect not exactly the target values but close to the target values.

```
In [18]: if __name__ == "__main__":
             XOR()
```

```
Performance of the MLP with 2 hidden layers with 3 neurons
```




```
Input: [0 0] -> Desired: [0] , Output: [ 0.04914958]
Input: [0 1] -> Desired: [1] , Output: [ 0.93731628]
Input: [1 0] -> Desired: [1] , Output: [ 0.93652772]
Input: [1 1] -> Desired: [0] , Output: [ 0.07407472]
```

0.2 Use the model to approximate a non-linear function.

0.2.1

$$y = sin(x) \ \textbf{for x in} \ [0, 2\pi]$$

0.2.2

$$\textbf{Take zero mean 0.1 variance noise } y = sin(x + noise)$$

0.2.3 Create 100 data points for training and 25 data points for testing.

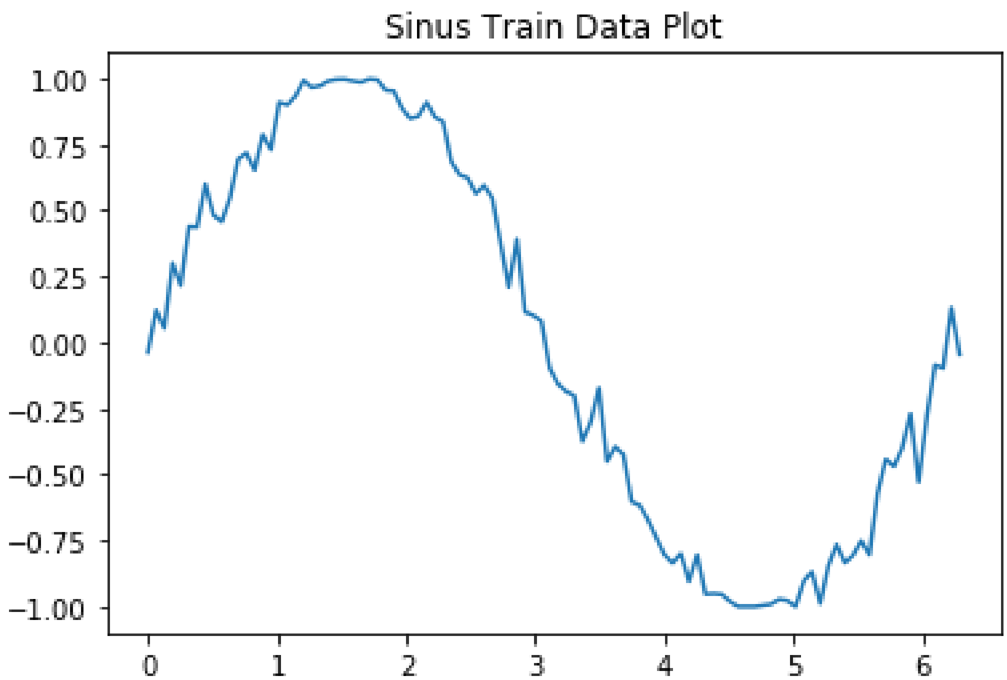
0.2.4 Implement the algorithm

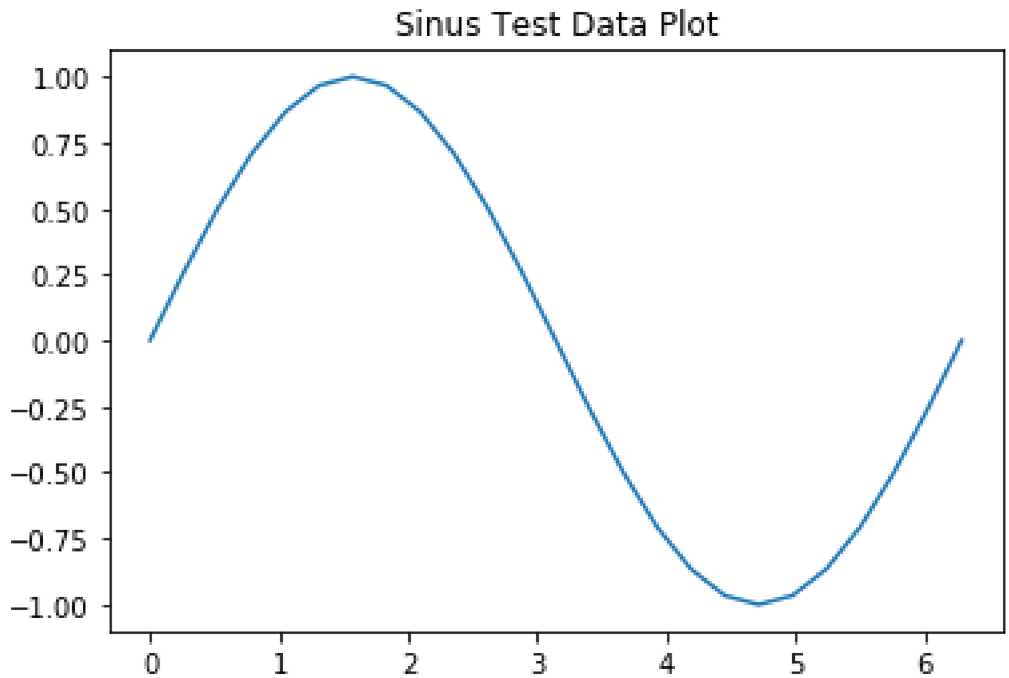
```
In [19]: def SinApproximation():
             import matplotlib.pyplot as plt
```

```python
import numpy as np


np.random.seed(578)

mu, sigma = 0, 0.1

#zero-mean Gaussian noise with standard deviation 0.1
noise1= np.random.normal(mu, sigma,100)

#inputs for train
trainSinus = np.linspace(0, 2*np.pi, 100)

#desired outputs for train
targetTrainSinus=np.sin(trainSinus+noise1)

#plot tranining noisy data
plt.plot(trainSinus,targetTrainSinus)
plt.title('Sinus Train Data Plot')
plt.show()

#inputs for test
testSinus = np.linspace(0, 2*np.pi, 25)

#desired outputs for test
targetTestSinus=np.sin(testSinus)

#plot test data points
plt.plot(testSinus,targetTestSinus)
plt.title('Sinus Test Data Plot')
plt.show()

#make patterns with inputs and targets
#for training
SinusTrainPatterns=[[[j]] for j in trainSinus]

for i in range(len(trainSinus)):
    SinusTrainPatterns[i].append([targetTrainSinus[i]])

#make patterns with inputs and targets
#for testing
SinusTestPatterns=[[[j]] for j in testSinus]

for i in range(len(testSinus)):
    SinusTestPatterns[i].append([targetTestSinus[i]])
```

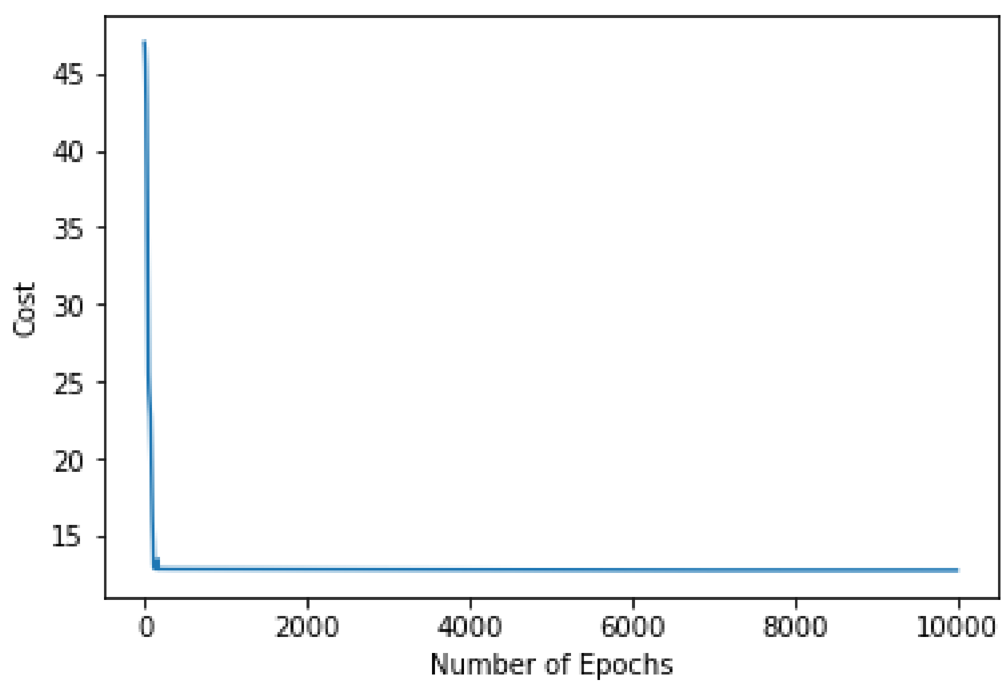
```
#Multilayer Perceptron model with:
#1 input neuron
#1 hiddden layer with 6 neurons
#and 1 output neuron

network_form=(1,6,1)
MLP = MultiLayerPerceptron(network_form)
print "Performance of the MLP with 1 hidden layer with 6 neurons"
eta=0.002
mu=0.7
epochs=10000

# train
MLP.train(SinusTrainPatterns,epochs,eta,mu)

# test
MLP.test(SinusTestPatterns,True)
```

```
In [20]: if __name__ == "__main__":
             SinApproximation()
```

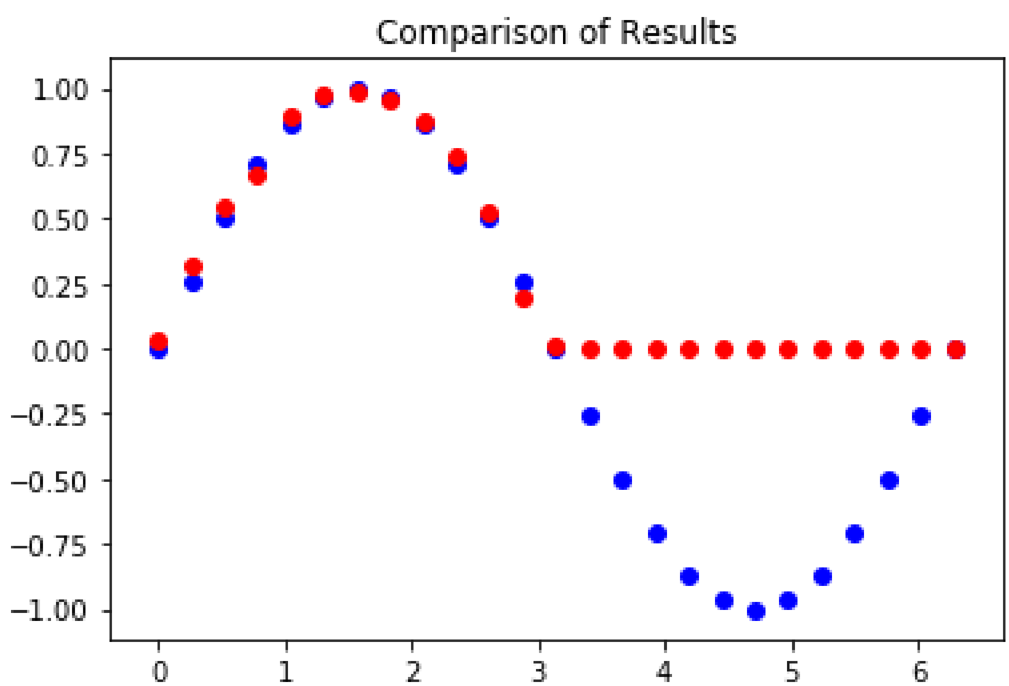




Performance of the MLP with 1 hidden layer with 6 neurons



```
Input: [ 0.] -> Desired: [ 0.] , Output: [ 0.03045874]
Input: [ 0.26179939] -> Desired: [ 0.25881905] , Output: [ 0.31428978]
Input: [ 0.52359878] -> Desired: [ 0.5] , Output: [ 0.54065067]
Input: [ 0.78539816] -> Desired: [ 0.70710678] , Output: [ 0.66888789]
Input: [ 1.04719755] -> Desired: [ 0.8660254] , Output: [ 0.8895553]
Input: [ 1.30899694] -> Desired: [ 0.96592583] , Output: [ 0.97856384]
Input: [ 1.57079633] -> Desired: [ 1.] , Output: [ 0.98128915]
Input: [ 1.83259571] -> Desired: [ 0.96592583] , Output: [ 0.95353919]
Input: [ 2.0943951] -> Desired: [ 0.8660254] , Output: [ 0.87376717]
Input: [ 2.35619449] -> Desired: [ 0.70710678] , Output: [ 0.73673453]
Input: [ 2.61799388] -> Desired: [ 0.5] , Output: [ 0.52892373]
Input: [ 2.87979327] -> Desired: [ 0.25881905] , Output: [ 0.19944895]
Input: [ 3.14159265] -> Desired: [  1.22464680e-16] , Output: [ 0.01516905]
Input: [ 3.40339204] -> Desired: [-0.25881905] , Output: [ 0.00039971]
Input: [ 3.66519143] -> Desired: [-0.5] , Output: [  2.38785201e-05]
Input: [ 3.92699082] -> Desired: [-0.70710678] , Output: [  5.89887771e-06]
Input: [ 4.1887902] -> Desired: [-0.8660254] , Output: [  3.40719816e-06]
Input: [ 4.45058959] -> Desired: [-0.96592583] , Output: [  2.78860948e-06]
Input: [ 4.71238898] -> Desired: [-1.] , Output: [  2.58631158e-06]
Input: [ 4.97418837] -> Desired: [-0.96592583] , Output: [  2.50545503e-06]
Input: [ 5.23598776] -> Desired: [-0.8660254] , Output: [  2.46535151e-06]
Input: [ 5.49778714] -> Desired: [-0.70710678] , Output: [  2.44084944e-06]
Input: [ 5.75958653] -> Desired: [-0.5] , Output: [  2.42349183e-06]
Input: [ 6.02138592] -> Desired: [-0.25881905] , Output: [  2.41017013e-06]
Input: [ 6.28318531] -> Desired: [ -2.44929360e-16] , Output: [  2.39956466e-06]
```



0.3 Apply the algorithm to Iris Data Set.

0.3.1 There 150 sample patterns. Pick 125 for training (randomly). Test the system with the rest 25 sample patterns.

0.3.2 Pick 3 outputs, one for each class of flowers, e.g:

$(y_1, y_2, y_3) = (1, 0, 0) \Rightarrow Class\ \ A(y_1, y_2, y_3) = (0, 1, 0) \Rightarrow Class\ \ B(y_1, y_2, y_3) = (0, 0, 1) \Rightarrow Class\ \ C$

Iris Data Set
The iris dataset contains measurements for 150 iris flowers from three different species.

```
The three classes in the Iris dataset are:
* setosa (n=50)
* versicolor (n=50)
* virginica (n=50)

And the four features of in Iris dataset are:
* sepal length
* sepal width
* petal length
* petal width
```

We store iris dataset in form of a 150×4 matrix where the columns are the different features, and every row represents a separate flower sample.
To implement the algorithm we convert target flowers names to numeric values.

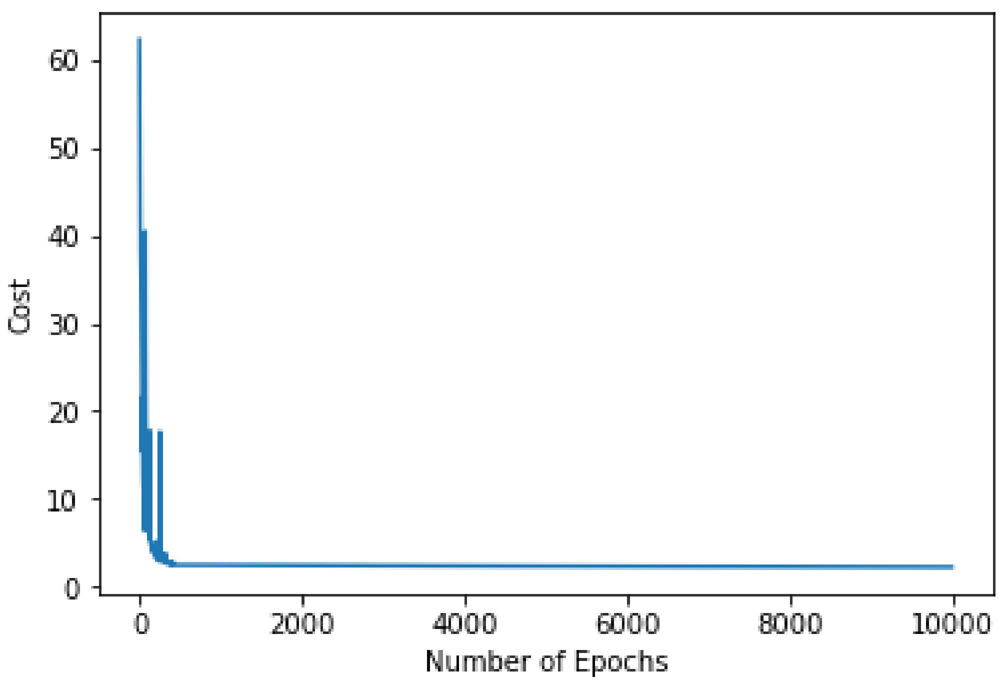
```
In [23]: def Classify_IrisFlowers():
             import csv
             import random

             random.seed(123)

             #Load iris dataset
             with open('data/iris.csv') as csvfile:
                 csvreader = csv.reader(csvfile)
                 #skips the header
                 next(csvreader, None)
                 dataset = list(csvreader)

             #Change string targets to numeric as:
             #Setosa=[1,0,0]
             #Versicolor=[0,1,0]
             #Virginica=[0,0,1]
             for row in dataset:
                 row[4] = ["setosa", "versicolor", "virginica"].index(row[4])
                 row[:4] = [float(row[j]) for j in xrange(len(row))]
                 if row[4]==0:
```

```
        row[4]=[1,0,0]
    elif row[4]==1:
        row[4]=[0,1,0]
    else:
        row[4]=[0,0,1]

#Split data to features and targets
#X is input
#y is target output

#shuffle data set
random.shuffle(dataset)

#data for training
datatrain = dataset[:125]

#data for testing
datatest = dataset[25:]

#inputs for training
train_X = [data[:4] for data in datatrain]

#targets for training
train_y = [data[4] for data in datatrain]

#inputs for testing
test_X = [data[:4] for data in datatest]

#targets for testing
test_y = [data[4] for data in datatest]

#rearrange training data form
datas=[]
for i in range(len(train_X)):
    datas.append([train_X[i]])
    datas[i].append(train_y[i])


# Multilayer Perceptron model with:
#4 input neurons
#1 hidden layer with 4 neurons
#and 3 output neurons

network_form=(4, 4,3)
MLP = MultiLayerPerceptron(network_form)
print "Performance of the MLP with 1 hidden layer with 4 neurons"
eta=0.02
mu=0.7
```

```
        epochs=10000

        # train
        MLP.train(datas,epochs,eta,mu)

        #rearrange testing data form
        testdatas=[]
        for i in range(len(test_X)):
            testdatas.append([test_X[i]])
            testdatas[i].append([test_y[i]])

        # test
        MLP.test(testdatas,input_index=True)
```

```
In [24]: if __name__ == "__main__":
             Classify_IrisFlowers()
```

```
Performance of the MLP with 1 hidden layer with 4 neurons
```




```
Input: 0 -> Desired: [[0 0 1]] , Output: [ 0.00312864  0.06063626  0.94978737]
Input: 1 -> Desired: [[0 0 1]] , Output: [ 0.00695222  0.45965394  0.55328942]
```

```
Input: 2 -> Desired: [[1 0 0]] , Output: [ 0.97104782  0.03794766  0.00103204]
Input: 3 -> Desired: [[0 0 1]] , Output: [ 0.00247965  0.04905451  0.97524005]
Input: 4 -> Desired: [[0 0 1]] , Output: [ 0.00242732  0.03630542  0.97754375]
Input: 5 -> Desired: [[0 1 0]] , Output: [ 0.02668848  0.96947732  0.01329617]
Input: 6 -> Desired: [[1 0 0]] , Output: [ 0.96948952  0.04226283  0.00104081]
Input: 7 -> Desired: [[0 1 0]] , Output: [ 0.02193264  0.95994777  0.02462573]
Input: 8 -> Desired: [[0 1 0]] , Output: [ 0.02368657  0.97044072  0.0186885 ]
Input: 9 -> Desired: [[0 0 1]] , Output: [ 0.0025451   0.03483118  0.97426659]
Input: 10 -> Desired: [[1 0 0]] , Output: [ 0.96969483  0.03869301  0.00105419]
Input: 11 -> Desired: [[0 0 1]] , Output: [ 0.00297782  0.0636466   0.95625252]
Input: 12 -> Desired: [[0 0 1]] , Output: [ 0.00281079  0.05171718  0.96392163]
Input: 13 -> Desired: [[0 0 1]] , Output: [ 0.00250182  0.03053721  0.97590736]
Input: 14 -> Desired: [[0 0 1]] , Output: [ 0.00245969  0.05198246  0.97566816]
Input: 15 -> Desired: [[0 0 1]] , Output: [ 0.00269205  0.04921328  0.96841725]
Input: 16 -> Desired: [[0 1 0]] , Output: [ 0.02661435  0.97105107  0.01341772]
Input: 17 -> Desired: [[0 0 1]] , Output: [ 0.00584916  0.28775921  0.6956182 ]
Input: 18 -> Desired: [[0 0 1]] , Output: [ 0.00565384  0.27910268  0.71809233]
Input: 19 -> Desired: [[1 0 0]] , Output: [ 0.97090634  0.03862985  0.00103136]
Input: 20 -> Desired: [[0 1 0]] , Output: [ 0.00307053  0.07581616  0.95123095]
Input: 21 -> Desired: [[0 0 1]] , Output: [ 0.00491165  0.18533902  0.80590022]
Input: 22 -> Desired: [[1 0 0]] , Output: [ 0.97119447  0.03696305  0.00103424]
Input: 23 -> Desired: [[0 1 0]] , Output: [ 0.02588986  0.96924477  0.01456453]
Input: 24 -> Desired: [[0 1 0]] , Output: [ 0.02416925  0.96593209  0.0179909 ]
Input: 25 -> Desired: [[0 0 1]] , Output: [ 0.00494015  0.24625799  0.79651359]
Input: 26 -> Desired: [[0 1 0]] , Output: [ 0.01085525  0.77279626  0.21345886]
Input: 27 -> Desired: [[0 1 0]] , Output: [ 0.02416897  0.96411535  0.01794241]
Input: 28 -> Desired: [[0 1 0]] , Output: [ 0.02612262  0.96997199  0.01423851]
Input: 29 -> Desired: [[0 0 1]] , Output: [ 0.00249533  0.04207078  0.97521061]
Input: 30 -> Desired: [[1 0 0]] , Output: [ 0.97114601  0.0372969   0.00103346]
Input: 31 -> Desired: [[0 1 0]] , Output: [ 0.02516234  0.96158355  0.01610685]
Input: 32 -> Desired: [[0 0 1]] , Output: [ 0.00241124  0.02723628  0.97868429]
Input: 33 -> Desired: [[1 0 0]] , Output: [ 0.96965275  0.0431143   0.00103391]
Input: 34 -> Desired: [[0 0 1]] , Output: [ 0.0024097   0.02733749  0.97871577]
Input: 35 -> Desired: [[1 0 0]] , Output: [ 0.97093575  0.03685448  0.00103985]
Input: 36 -> Desired: [[0 0 1]] , Output: [ 0.00241673  0.03373197  0.97801675]
Input: 37 -> Desired: [[0 0 1]] , Output: [ 0.00240568  0.02953354  0.97863859]
Input: 38 -> Desired: [[1 0 0]] , Output: [ 0.97005196  0.03931739  0.00104429]
Input: 39 -> Desired: [[1 0 0]] , Output: [ 0.97113842  0.03795457  0.00103025]
Input: 40 -> Desired: [[0 1 0]] , Output: [ 0.02711703  0.97527497  0.01274563]
Input: 41 -> Desired: [[1 0 0]] , Output: [ 0.96886648  0.04106533  0.001058  ]
Input: 42 -> Desired: [[0 1 0]] , Output: [ 0.0219823   0.95252709  0.02470238]
Input: 43 -> Desired: [[0 1 0]] , Output: [ 0.02775786  0.97551139  0.01255327]
Input: 44 -> Desired: [[0 0 1]] , Output: [ 0.00241897  0.03478273  0.97788026]
Input: 45 -> Desired: [[0 0 1]] , Output: [ 0.00248194  0.03452708  0.9761497 ]
Input: 46 -> Desired: [[0 1 0]] , Output: [ 0.02432118  0.96535124  0.01753487]
Input: 47 -> Desired: [[0 0 1]] , Output: [ 0.00252876  0.04210388  0.9742283 ]
Input: 48 -> Desired: [[0 1 0]] , Output: [ 0.02380517  0.96863443  0.01845134]
Input: 49 -> Desired: [[1 0 0]] , Output: [ 0.97094895  0.04036096  0.00102215]
```

```
Input: 50 -> Desired: [[1 0 0]] , Output: [ 0.97134007  0.03686107  0.00103193]
Input: 51 -> Desired: [[0 1 0]] , Output: [ 0.02430328  0.96706867  0.01774725]
Input: 52 -> Desired: [[1 0 0]] , Output: [ 0.97096124  0.03877505  0.00102958]
Input: 53 -> Desired: [[0 0 1]] , Output: [ 0.00247373  0.03828457  0.97610328]
Input: 54 -> Desired: [[0 1 0]] , Output: [ 0.03154618  0.96813943  0.01215259]
Input: 55 -> Desired: [[0 0 1]] , Output: [ 0.00243131  0.04232166  0.97703732]
Input: 56 -> Desired: [[0 1 0]] , Output: [ 0.02687762  0.97102841  0.01317747]
Input: 57 -> Desired: [[0 0 1]] , Output: [ 0.00307855  0.05985289  0.95216143]
Input: 58 -> Desired: [[1 0 0]] , Output: [ 0.96938766  0.03927437  0.00105706]
Input: 59 -> Desired: [[1 0 0]] , Output: [ 0.97109066  0.03662043  0.00103806]
Input: 60 -> Desired: [[0 1 0]] , Output: [ 0.03796179  0.95901626  0.01151728]
Input: 61 -> Desired: [[1 0 0]] , Output: [ 0.97053097  0.04109772  0.00102666]
Input: 62 -> Desired: [[0 1 0]] , Output: [ 0.0208297   0.95091878  0.02913204]
Input: 63 -> Desired: [[0 0 1]] , Output: [ 0.00286679  0.05250837  0.96171629]
Input: 64 -> Desired: [[1 0 0]] , Output: [ 0.97106532  0.03739741  0.00103451]
Input: 65 -> Desired: [[1 0 0]] , Output: [ 0.96943761  0.04290104  0.00103885]
Input: 66 -> Desired: [[0 1 0]] , Output: [ 0.02552403  0.96831213  0.01494689]
Input: 67 -> Desired: [[1 0 0]] , Output: [ 0.97055547  0.03994403  0.00103166]
Input: 68 -> Desired: [[1 0 0]] , Output: [ 0.97055055  0.03875025  0.0010376 ]
Input: 69 -> Desired: [[1 0 0]] , Output: [ 0.97050116  0.04149774  0.00102537]
Input: 70 -> Desired: [[0 0 1]] , Output: [ 0.00244294  0.0516523   0.97617409]
Input: 71 -> Desired: [[1 0 0]] , Output: [ 0.97132834  0.03540824  0.00103989]
Input: 72 -> Desired: [[0 1 0]] , Output: [ 0.02911336  0.96547511  0.01291818]
Input: 73 -> Desired: [[0 1 0]] , Output: [ 0.01927373  0.94517453  0.03723408]
Input: 74 -> Desired: [[0 1 0]] , Output: [ 0.0273434   0.97157317  0.01301556]
Input: 75 -> Desired: [[0 0 1]] , Output: [ 0.00520974  0.22721261  0.77125364]
Input: 76 -> Desired: [[0 1 0]] , Output: [ 0.0251825   0.97073182  0.01543195]
Input: 77 -> Desired: [[1 0 0]] , Output: [ 0.9696707   0.04234018  0.00103708]
Input: 78 -> Desired: [[1 0 0]] , Output: [ 0.97021041  0.04159183  0.00103042]
Input: 79 -> Desired: [[0 1 0]] , Output: [ 0.00740734  0.4318783   0.50841014]
Input: 80 -> Desired: [[0 1 0]] , Output: [ 0.02624917  0.96721755  0.01401592]
Input: 81 -> Desired: [[0 0 1]] , Output: [ 0.00244325  0.05506386  0.9759895 ]
Input: 82 -> Desired: [[0 0 1]] , Output: [ 0.00280084  0.052902    0.96420387]
Input: 83 -> Desired: [[0 1 0]] , Output: [ 0.02689783  0.9747937   0.01309112]
Input: 84 -> Desired: [[0 1 0]] , Output: [ 0.02676288  0.96965098  0.0133298 ]
Input: 85 -> Desired: [[0 1 0]] , Output: [ 0.02436592  0.96023484  0.01770659]
Input: 86 -> Desired: [[0 1 0]] , Output: [ 0.02592146  0.96832779  0.01434581]
Input: 87 -> Desired: [[1 0 0]] , Output: [ 0.97014612  0.04116756  0.0010336 ]
Input: 88 -> Desired: [[1 0 0]] , Output: [ 0.97120464  0.03826886  0.00102739]
Input: 89 -> Desired: [[0 0 1]] , Output: [ 0.00366361  0.08433358  0.9182904 ]
Input: 90 -> Desired: [[0 1 0]] , Output: [ 0.02578613  0.96587597  0.01494643]
Input: 91 -> Desired: [[0 0 1]] , Output: [ 0.00423169  0.15043814  0.87066106]
Input: 92 -> Desired: [[1 0 0]] , Output: [ 0.96739646  0.04485299  0.00106721]
Input: 93 -> Desired: [[0 1 0]] , Output: [ 0.02656894  0.97073327  0.01354494]
Input: 94 -> Desired: [[0 0 1]] , Output: [ 0.00264417  0.03633674  0.97105271]
Input: 95 -> Desired: [[0 1 0]] , Output: [ 0.02907031  0.97322789  0.01242564]
Input: 96 -> Desired: [[0 0 1]] , Output: [ 0.00247116  0.03659158  0.97629333]
Input: 97 -> Desired: [[1 0 0]] , Output: [ 0.97131946  0.03593242  0.00103723]
```

```
Input: 98 -> Desired: [[0 1 0]] , Output: [ 0.02733046  0.97003686  0.0131    ]
Input: 99 -> Desired: [[0 0 1]] , Output: [ 0.00243576  0.03769241  0.97721758]
Input: 100 -> Desired: [[1 0 0]] , Output: [ 0.97086491  0.03928896  0.00102891]
Input: 101 -> Desired: [[0 0 1]] , Output: [ 0.00240076  0.02926662  0.97878767]
Input: 102 -> Desired: [[1 0 0]] , Output: [ 0.971103    0.03673116  0.00103724]
Input: 103 -> Desired: [[0 0 1]] , Output: [ 0.00256462  0.06512507  0.97173412]
Input: 104 -> Desired: [[0 0 1]] , Output: [ 0.00246357  0.03155184  0.97689619]
Input: 105 -> Desired: [[0 1 0]] , Output: [ 0.02636231  0.97252705  0.01347918]
Input: 106 -> Desired: [[1 0 0]] , Output: [ 0.96721996  0.05127263  0.00104378]
Input: 107 -> Desired: [[1 0 0]] , Output: [ 0.97050116  0.04149774  0.00102537]
Input: 108 -> Desired: [[0 1 0]] , Output: [ 0.02690947  0.97509439  0.01313331]
Input: 109 -> Desired: [[1 0 0]] , Output: [ 0.97021695  0.04007711  0.00103745]
Input: 110 -> Desired: [[0 1 0]] , Output: [ 0.02352155  0.95940413  0.02015463]
Input: 111 -> Desired: [[1 0 0]] , Output: [ 0.97114214  0.03761339  0.00103191]
Input: 112 -> Desired: [[1 0 0]] , Output: [ 0.97138074  0.03518338  0.00104008]
Input: 113 -> Desired: [[1 0 0]] , Output: [ 0.97101856  0.03719176  0.00103648]
Input: 114 -> Desired: [[0 0 1]] , Output: [ 0.00247373  0.03828457  0.97610328]
Input: 115 -> Desired: [[1 0 0]] , Output: [ 0.97134571  0.03633976  0.00103455]
Input: 116 -> Desired: [[0 0 1]] , Output: [ 0.00246288  0.03087336  0.97697405]
Input: 117 -> Desired: [[1 0 0]] , Output: [ 0.97043802  0.04030168  0.00103218]
Input: 118 -> Desired: [[0 1 0]] , Output: [ 0.01600846  0.88781208  0.06934888]
Input: 119 -> Desired: [[1 0 0]] , Output: [ 0.97090436  0.03658285  0.00104189]
Input: 120 -> Desired: [[0 0 1]] , Output: [ 0.00449797  0.15379634  0.84763793]
Input: 121 -> Desired: [[1 0 0]] , Output: [ 0.97135679  0.03466781  0.00104341]
Input: 122 -> Desired: [[0 1 0]] , Output: [ 0.02635079  0.97763669  0.01386742]
Input: 123 -> Desired: [[1 0 0]] , Output: [ 0.97059606  0.04169341  0.00102267]
Input: 124 -> Desired: [[1 0 0]] , Output: [ 0.97082396  0.03880944  0.00103205]
```