NN_RubanrajRavichandran_13112017

November 19, 2017

```
In [8]: import numpy as np
        import sympy as sp
        import matplotlib.pyplot as plt
        import random
        %matplotlib inline
```

1 Exercise 2

For this task you have to program the back-propogation (BP) for multi layered perceptron (MLP). Design your implementation for general NN with arbitrary many hidden layers. The test case is as follows: 2-2-1 multi layered perceptron (MLP) with sigmoid activation function on XOR data.

a. Experiments with initial weights

b. Train the network with zero initial weights i.e. wij = 0.

ii. Train with random initial weights

```
In [9]: class NeuralNetwork:

            def __init__(self,
                         _no_input_neuron,
                         _hidden_config,
                         _no_output_neuron,
                         _input_data_set,
                         _desired_output,
                         _learning_rate,
                         _random_initial_weight):

                self.no_of_input_neurons = _no_input_neuron
                self._hidden_config = _hidden_config
                self.no_of_layers = len(_hidden_config)
                self.no_of_hidden_neurons = _hidden_config[0]
                self.no_of_output_neurons = _no_output_neuron

                self.input_data_set = _input_data_set
                self.desired_output = _desired_output
```

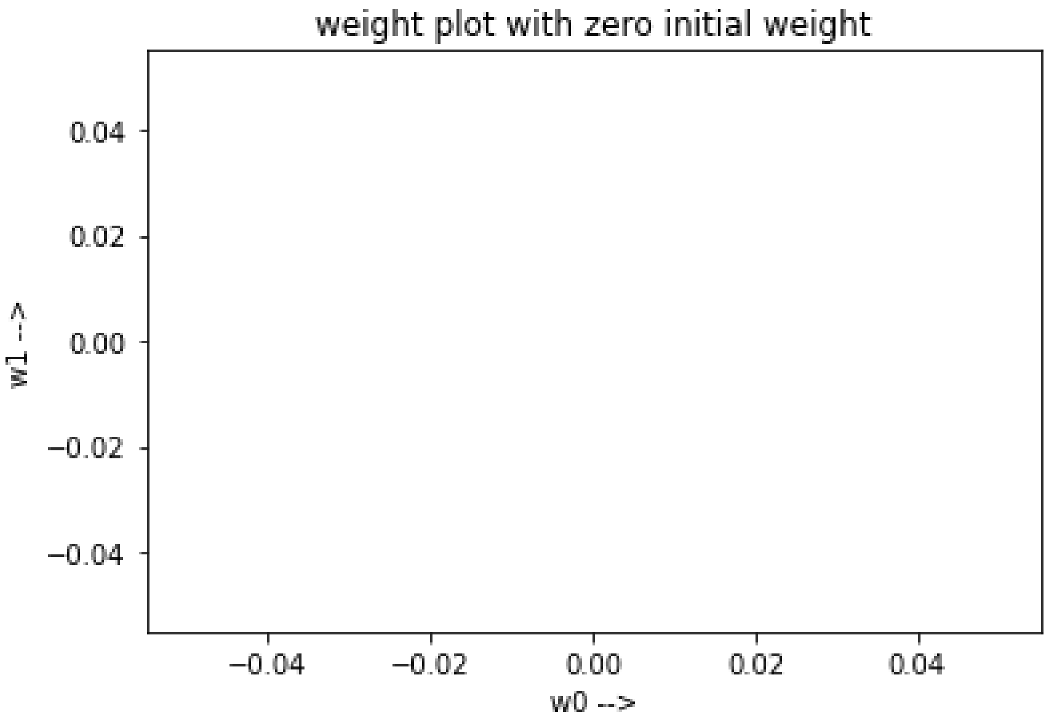
```
        self.learning_rate = _learning_rate
        self.random_initial_weight = _random_initial_weight

    def generate_weight(self,is_random,_size):
        return np.random.uniform(size=_size) if is_random else np.zeros(_size)

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

    def derivative_(self,x):
        return x * (1 - x)

    def local_field(self,x,w):
        return np.dot(x,w)

    def error(self,y):
        return self.desired_output - y

    def delta(self,sigma_tic,summed_error,flag = False):
        return summed_error * sigma_tic
#         return np.dot(sigma_tic,summed_error) if flag else sigma_tic*summed_error

    def backpropagation(self,fig_title):

        # weights from input layer to hidden layer
        Wh = self.generate_weight(self.random_initial_weight,(self.no_of_input_neurons,
        # weights from hidden layer to output layer
        Wz = self.generate_weight(self.random_initial_weight,(self._hidden_config[0],sel

        w_0 = []
        w_1 = []

        avg_error = float('inf')
        epochs = 0

        # In zero initial weight case the error will be always same,
        # so we need to break the loop after some maximum epoch limit
        while avg_error > 0.01 and epochs < 1000000:
            #activation result from hidden layer neurons
            H = self.sigmoid(np.dot(bp.input_data_set, Wh))
            #activatiion result from output neurons
            Z = self.sigmoid(np.dot(H, Wz))
            #error calculation
            E = self.desired_output - Z
            #calculating delta_j for output neuron
            dZ = E * self.derivative_(Z)
            #calculating delta_j for hidden neuron
```

```
            dH = dZ.dot(Wz.T) * self.derivative_(H)

            #updating weights using backpropagation
            Wz +=  self.learning_rate * H.T.dot(dZ)
            Wh +=  self.learning_rate * self.input_data_set.T.dot(dH)

            w_0.append(Wz[0,:][0])
            w_1.append(Wz[1,:][0])
            avg_error = (np.average(E**2))
            epochs += 1

        w_0 = np.asarray(w_0)
        w_1 = np.asarray(w_1)

        plt.plot(w_0,w_1)
        plt.xlabel('w0 -->')
        plt.ylabel('w1 -->')
        plt.title(fig_title)
        return E,Wh,Wz,epochs
```

In [10]:
```
"""
parameters required to init NeuralNetwork class:
    1. number of input layer
    2. number of hidden neurons in each hidden layers
    3. input data set
    4. desired output set
    5. learning rate
    6. Random initial weight or zero initial weight (True or False)
"""
```

Out[10]: '\nparameters required to init NeuralNetwork class:\n   1. number of input layer\n

In [26]:
```
"""
case 1: start with zero weights
    If we start with zero weights, then all hidden neurons will get zero signal
    even the real input is some non-zero number.
    By using the given sample input, observed that the error is always same and the wei
    To avoid this symmetry condition, we always start learning process with random weig
    since the error is always same, we breaking the learning process after some number
"""
bp = NeuralNetwork(2,
                   [2],
                   1,
                   np.array([[0,0],[0,1],[1,0],[1,1]]),
                   np.array([[0],[1],[1],[0]]),
                   0.1,False)

error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("weight plo
```

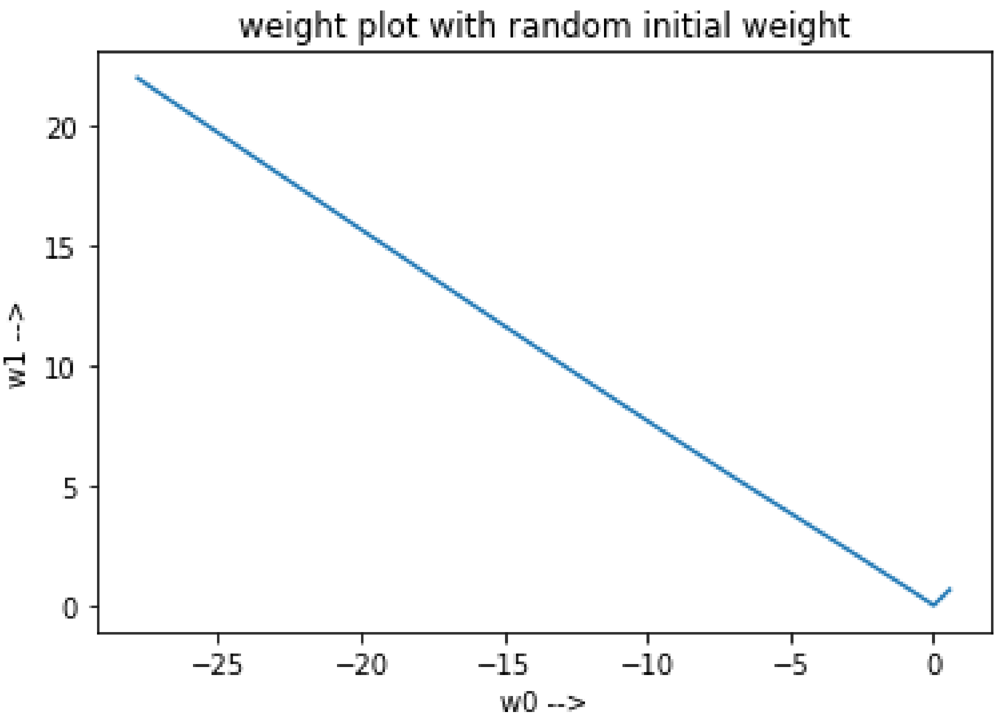



```
In [27]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]
weights from input to hidden layer :
[[ 0.  0.]
 [ 0.  0.]]
weights from hidden to output layer :
[[ 0.]
 [ 0.]]
Number of epochs :  1000000
```

```
In [28]: """
         case 2: start with random weights
                 If we start with random weights the error is reducing proportional to the learn
         """
         bp = NeuralNetwork(2,
```

```
                        [2],
                        1,
                        np.array([[0,0],[0,1],[1,0],[1,1]]),
                        np.array([[0],[1],[1],[0]]),
                        0.1,True)

error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("weight plo
```




```
In [29]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.05269223]
 [ 0.09943523]
 [ 0.09943517]
 [-0.1320927 ]]
weights from input to hidden layer :
[[ 0.90718117  7.35114361]
 [ 0.90718413  7.35228553]]
weights from hidden to output layer :
[[-27.80472916]
 [ 22.02638595]]
```

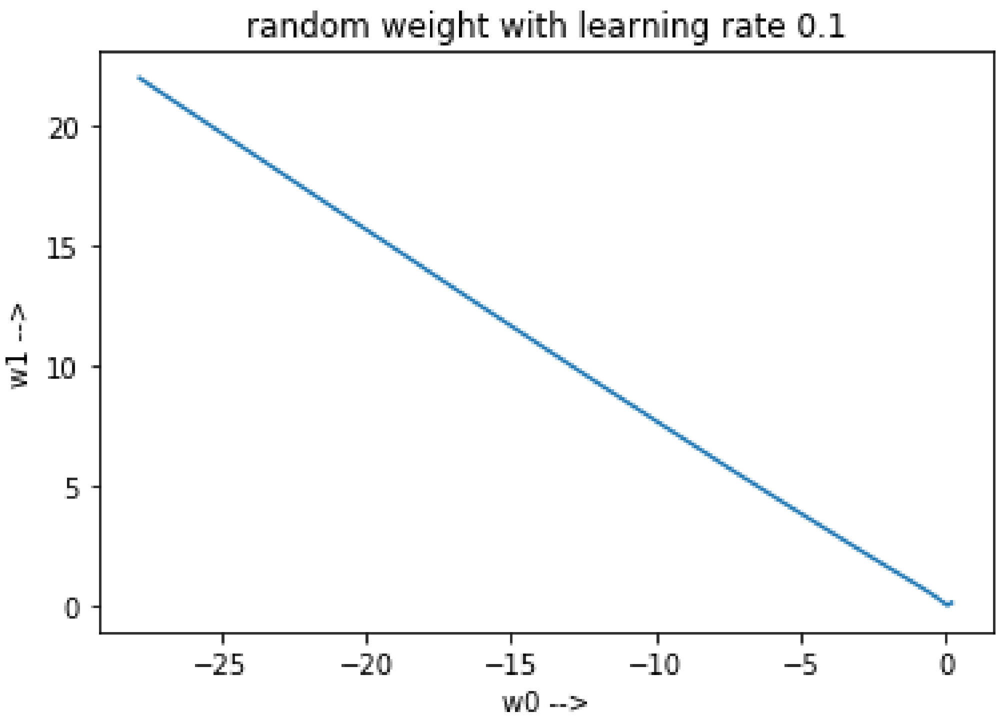
```
Number of epochs :  54515
```

Compare and comment on the convergence.

b. Experiment with different learning rates e.g. 0.1, 0.3, 0.5, 0.9..

Compare the convergence and plot some resulting surfaces. You are not allowed to use any neural network toolbox for this solution.

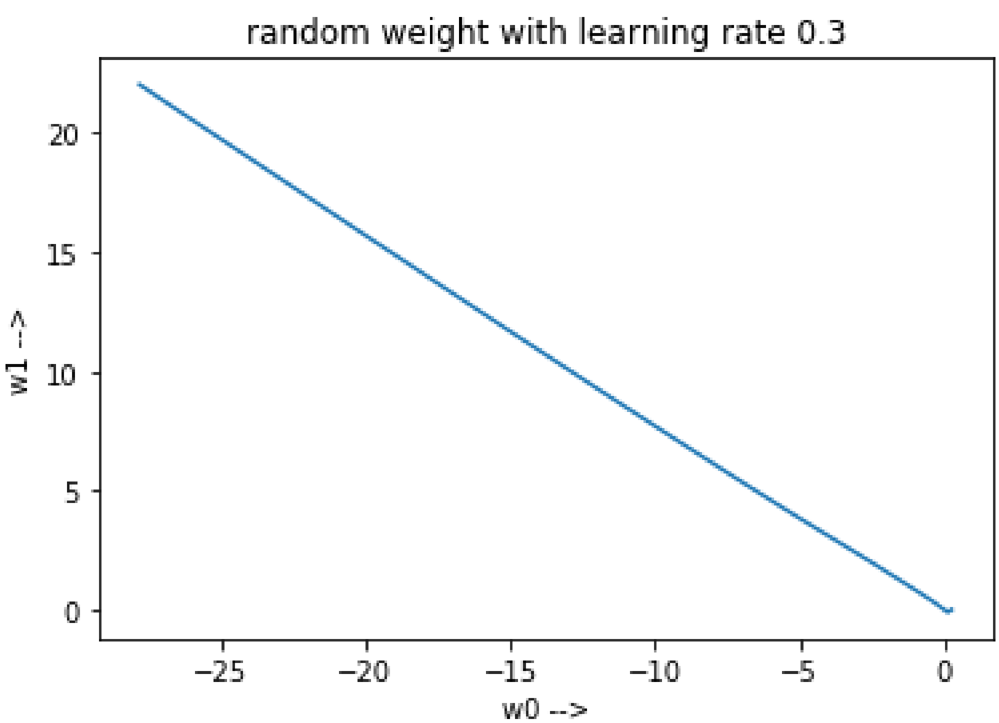
```
In [30]: """
         case b (i): start with random weights and learning rate is 0.1
         """
         bp = NeuralNetwork(2,
                            [2],
                            1,
                            np.array([[0,0],[0,1],[1,0],[1,1]]),
                            np.array([[0],[1],[1],[0]]),
                            0.1,True)
         error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("random wei
```




```
In [31]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.05269191]
 [ 0.09943465]
 [ 0.09943461]
 [-0.13209194]]
weights from input to hidden layer :
[[ 0.90718192  7.35169621]
 [ 0.90718381  7.35242463]]
weights from hidden to output layer :
[[-27.80478279]
 [ 22.02642687]]
Number of epochs :  51702
```

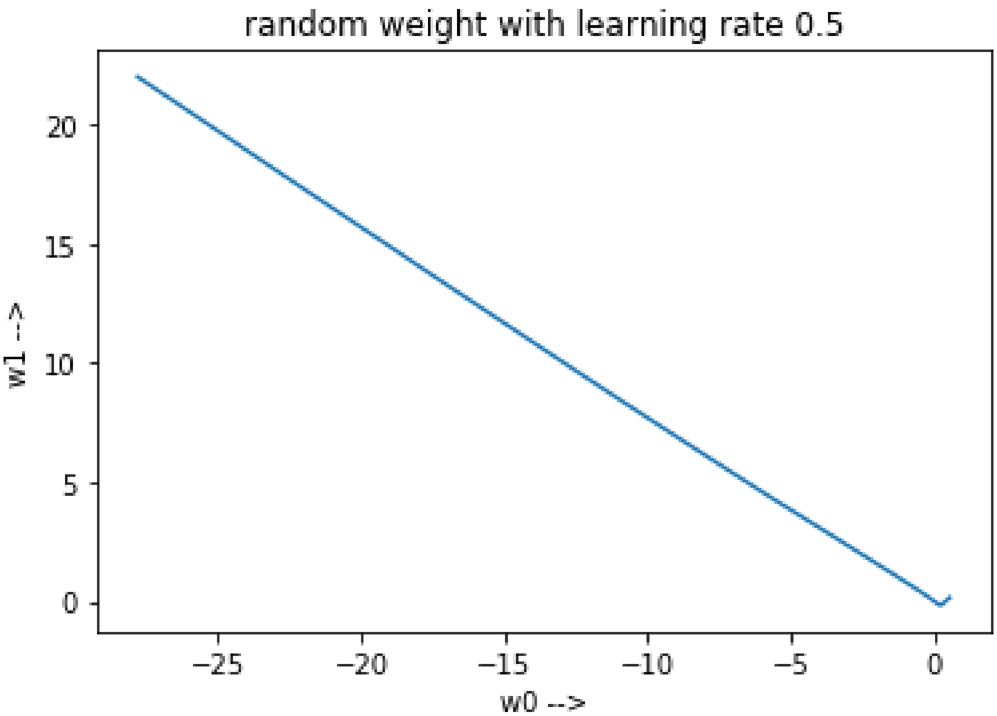
In [32]:
```
"""
case b (i): start with random weights and learning rate is 0.3
"""
bp = NeuralNetwork(2,
                   [2],
                   1,
                   np.array([[0,0],[0,1],[1,0],[1,1]]),
                   np.array([[0],[1],[1],[0]]),
                   0.3,True)

error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("random wei
```



```
In [33]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.05269187]
 [ 0.09943471]
 [ 0.09943442]
 [-0.13209185]]
weights from input to hidden layer :
[[ 0.90717796  7.34964572]
 [ 0.90719064  7.35453819]]
weights from hidden to output layer :
[[-27.80512237]
 [ 22.02670231]]
Number of epochs :  18444
```

```
In [34]: """
         case b (i): start with random weights and learning rate is 0.5
         """
         bp = NeuralNetwork(2,
                            [2],
                            1,
                            np.array([[0,0],[0,1],[1,0],[1,1]]),
                            np.array([[0],[1],[1],[0]]),
                            0.5,True)
         error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("random wei
```

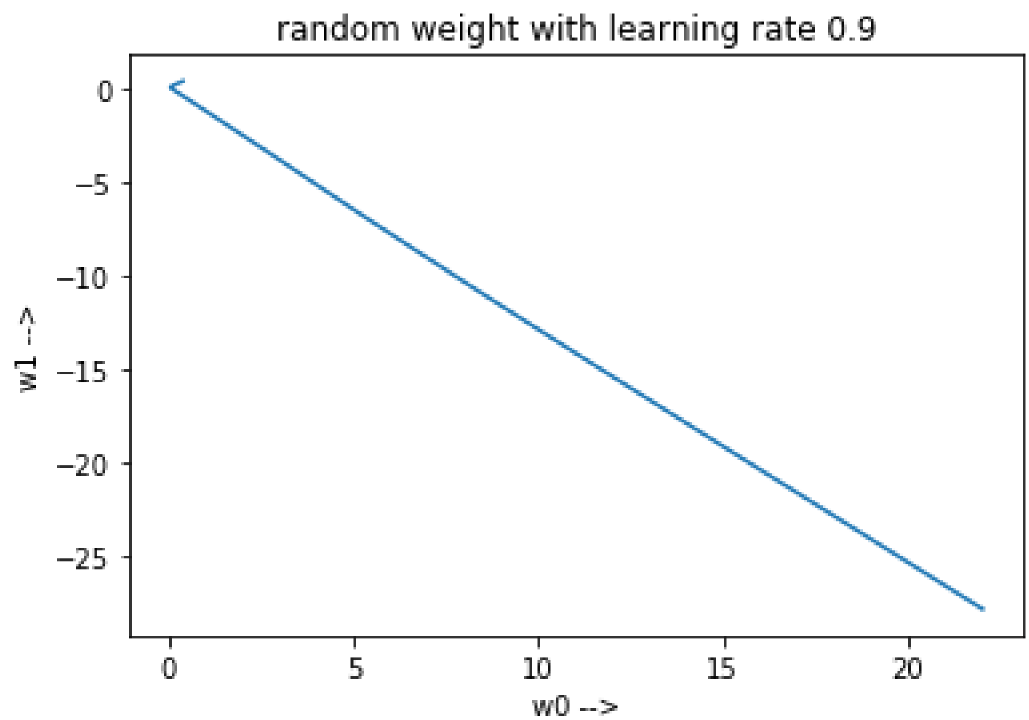



```
In [35]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.05268999]
 [ 0.09943201]
 [ 0.0994319 ]
 [-0.13208844]]
weights from input to hidden layer :
[[ 0.90718501  7.35099816]
 [ 0.90718984  7.35286328]]
weights from hidden to output layer :
[[-27.80585887]
 [ 22.02730133]]
Number of epochs :  10752
```

```
In [36]: """
         case b (i): start with random weights and learning rate is 0.9
         """
         bp = NeuralNetwork(2,
                            [2],
```

```
                                          1,
                                          np.array([[0,0],[0,1],[1,0],[1,1]]),
                                          np.array([[0],[1],[1],[0]]),
                                          0.9,True)

        error,weights_hidden_layer,weights_output_layer,epochs = bp.backpropagation("random wei
```




```
In [37]: print "Error : \n", error
         print "weights from input to hidden layer : \n", weights_hidden_layer
         print "weights from hidden to output layer : \n", weights_output_layer
         print "Number of epochs : ", epochs
```

```
Error :
[[-0.05269155]
 [ 0.09943413]
 [ 0.09943414]
 [-0.13209129]]
weights from input to hidden layer :
[[ 7.352197    0.90718899]
 [ 7.35204041  0.90718858]]
weights from hidden to output layer :
[[ 22.02756127]
 [-27.80618086]]
Number of epochs :  5752
```

2 observations with different learning rate

By increasing the learning rate, observed that error and number of epochs is reducing significantly faster. In case one, with learning rate 0.1 the error is [[-0.03347088] [0.07031518] [0.07031523] [-0.09374767]] and with learning rate 0.9 the error is reduced to [[-0.00719604][0.02115464] [0.02115463][-0.02845579]].

But we are not sure, the network found the optimal solution with the huge learning rate or it stuck with local minima.

3 Exercise 3

Investigate the use of back-propagation learning using a sigmoidal nonlinearity to achieve one-toone mappings, as described here:

1. F(x) = 1/x 1<=x<=100
2. F(x) = log10(x) 1<=x<=10
3. F(x) = exp(-x) 1<=x<=10
4. F(x) = sin(x) 0<=x<=pi/2

(a) Set up two sets of data, one for network training, and the other for testing.

(b) Use the training data set to compute the synaptic weights of the network, assumed to have a single hidden layer.

```
In [39]: bp = NeuralNetwork(2,
                            [2],
                            1,
                            np.array([[0,0],[0,1],[1,0],[1,1]]),
                            np.array([[0],[1],[1],[0]]),
                            0.1,True)
```

```
In [40]: # Generate data based on minimum and maximum value, n is no. of samples
         def get_data(minimum,maximum,n):
             data = np.zeros((1,n))
             for i in range(n):
                 data[0,i] = random.uniform(minimum, maximum)
             return data

         #Training phase
         #Wh and Wz are hidden and output layer weights
         def training(training_set,desired,Wh,Wz):

             epoch_e = []
             squared_error = []
             epoch_count = 0
             while(True):
```

```
        #Forward pass
        vh = bp.local_field(training_set.T,Wh)
        sigmoid = bp.sigmoid(vh)
        vo = bp.local_field(sigmoid,Wz)
        y = bp.sigmoid(vo)
        e = desired.T - y
        epoch_e.append(e)

        #Backward pass
        dZ = e * bp.derivative_(y)
        dH = dZ.dot(Wz.T) * bp.derivative_(sigmoid)
        Wz +=  -eta * np.dot(sigmoid.T,dZ)
        Wh +=  -eta * training_set.dot(dH)

        if epoch_count>0:
            squared_error.append((epoch_e[epoch_count]**2-epoch_e[epoch_count-1]**2))
            # If average squared error is less than 0.01, we stop adjustment
            if  (np.average(squared_error[epoch_count-1])) < 0.01:
                break

        epoch_count +=1
    print "Training phase"
    print "Number of epochs it took: ", epoch_count
    return (Wh,Wz)

# Test data using weight adjusted during training phase
def testing(test_data,Wh,Wz):
    print "Testing"
    test_vh = bp.local_field(test_data.T,Wh)
    test_sigmoid = bp.sigmoid(test_vh)
    test_vo = bp.local_field(test_sigmoid,Wz)
    output_mapping = bp.sigmoid(test_vo)
    return output_mapping


'''
To compute accuracy:
    i. We check how much test data is classified correctly,
     based on weights adjusted during training phase
'''
def compute_accuracy(training_data,test_set,
                     desired,expected,
                     nHidden,nOutput):


    Wh = np.random.rand(1, nHidden)
    Wz = np.random.rand(nHidden,nOutput)
```

```
    Wh,Wz = training(training_data,desired,Wh,Wz)
    print "Adjusted weights from training phase :"
    print "hidden weights ", Wh
    print "Output weights ", Wz
    #we use sample weights for testing
    actual = testing(test_set,Wh,Wz)
```

In [41]:
```
#Generate Training data

#get_data takens minimum, maximum, number_of_samples
#f(x) = 1/x
training_set1 = get_data(1,100,200)
desired1 = np.asarray([1.0/x for x in training_set1])

#f(x) = log_10(x)
training_set2 = get_data(1,10,20)
desired2 = np.asarray([np.log10(x) for x in training_set2])

#f(x) = exp(-x)
training_set3 = get_data(1,10,20)
desired3 = np.asarray([np.exp(-x) for x in training_set3])

#f(x) = sin(x)
training_set4 = get_data(1,45,20)
desired4 = np.asarray([np.sin(x) for x in training_set4])

#Generate Test sets
# we generate half numebr of samples for test as compare to training
test_set1 = get_data(1,100,100)
expected1 = np.asarray([1.0/x for x in test_set1])

test_set2 = get_data(1,10,10)
expected2 = np.asarray([np.log10(x) for x in test_set2])

test_set3 = get_data(1,10,10)
expected3 = np.asarray([np.exp(-x) for x in test_set3])

test_set4 = get_data(1,45,10)
expected4 = np.asarray([np.sin(np.radians(x)) for x in test_set4])


nHLayers = 1 #hidden layers
nOutput = 1 #hidden neurons
eta = 0.3 #learning rate
nHidden = [3] # number of hidden  neurons
```

(c) Evaluate the computation accuracy of the network by using the test data. Use a single hidden layer but with a variable number of hidden neurons. Investigate how the network performance is affected by varying the size of the hidden layer.

```
In [42]: #i) f(x) = 1/x
         for i in range(len(nHidden)):
             compute_accuracy(training_set1,test_set1,desired1,expected1,nHidden[i],nOutput)
```

```
Training phase
Number of epochs it took:  2
Adjusted weights from training phase :
hidden weights  [[ 0.60347235  3.94090292  1.13123281]]
Output weights  [[ 7.94291958]
 [ 6.8479807 ]
 [ 7.94948768]]
Testing
```

```
In [43]: #i) f(x) = log(x)
         for i in range(len(nHidden)):
             compute_accuracy(training_set2,test_set2,desired2,expected2,nHidden[i],nOutput)
```

```
Training phase
Number of epochs it took:  1
Adjusted weights from training phase :
hidden weights  [[ 0.75769781  0.99959185  0.44084522]]
Output weights  [[ 0.90777845]
 [ 0.20905923]
 [ 0.86811297]]
Testing
```

```
In [44]: #i) f(x) = exp(-x)
         for i in range(len(nHidden)):
             compute_accuracy(training_set3,test_set3,desired3,expected3,nHidden[i],nOutput)
```

```
Training phase
Number of epochs it took:  5
Adjusted weights from training phase :
hidden weights  [[ 0.76421993  0.80474788  0.58514857]]
Output weights  [[ 1.74636817]
 [ 1.88717087]
 [ 1.25108536]]
Testing
```

```
In [45]: #i) f(x) = sin(x)
         for i in range(len(nHidden)):
             compute_accuracy(training_set4,test_set4,desired4,expected4,nHidden[i],nOutput)
```

```
Training phase
Number of epochs it took:  3
Adjusted weights from training phase :
hidden weights  [[ 0.55568158  0.86125208  1.00000122]]
Output weights  [[ 1.76624946]
 [ 1.60054639]
 [ 1.08855542]]
Testing
```