Assignment_08

December 2, 2017

1 Assingment 08

1.0.1 RaviKiran Bhat

1.0.2 Rubanraj RaviChandran

1.0.3 Ramesh Kumar

2 Exercise 3

Tasks:

```
In [52]: Image(filename='fig2.png')
```

```
Out[52]:
```

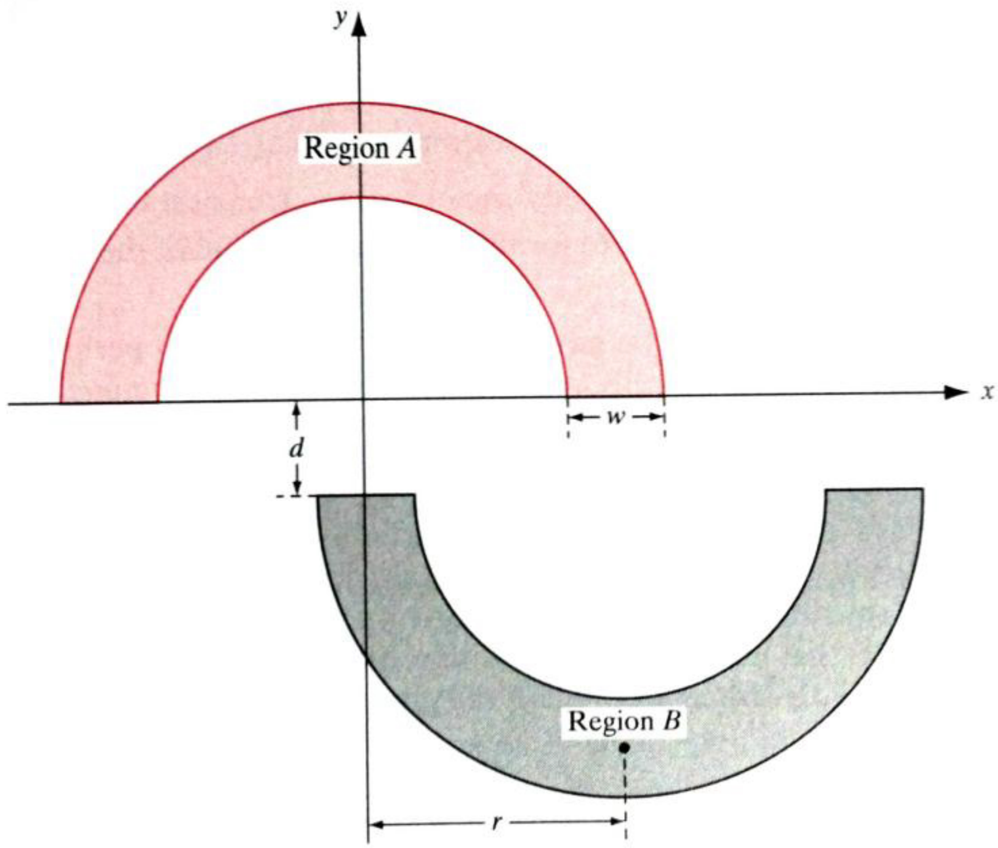


FIGURE 1.8 The double-moon classification problem.

In [53]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from IPython.display import Image
from sklearn import svm
from sklearn.cluster import KMeans
%matplotlib inline
```

In [54]:
```
class StateVectorMachine:

    def __init__(self,_radius,_width,_distance,_num_of_training_set,
                 _num_of_testing_set):
        self.radius = _radius
        self.width = _width
        self.distance = _distance
        self.num_of_training_set = _num_of_training_set
        self.num_of_testing_set = _num_of_testing_set
```

```python
def generate_sample(self,_class):
    random_theta = np.pi *  random.random()
    random_r = (self.width*random.random())+(self.radius-self.width)
    #Region one
    if _class is 1:
        x = random_r*np.cos(random_theta)
        y = random_r*np.sin(random_theta)
        return [x,y,1]
    else:
        #Region two
        random_theta += np.pi
        x = random_r*np.cos(random_theta)+(self.radius-(self.width/2.0))
        y = random_r*np.sin(random_theta)-self.distance
        return [x,y,2]

def get_samples(self,_flag):
    samples = np.empty((0,3))

    if _flag is "train":
        _no_of_samples = self.num_of_training_set
    else:
        _no_of_samples = self.num_of_testing_set

    """
    - generating number of samples
    - half samples belongs to region A and
      remaining half samples belongs to region B
    """
    for i in range(_no_of_samples):
        sample = self.generate_sample(1 if (i<_no_of_samples/2) else 2)
        samples = np.vstack([samples,sample])

    #returning samples and desired output
    return samples[:,0:2],samples[:,2:3]

def plot(self,points,output,title):
    plt.grid(True)
    plt.title(title)
    plt.xlabel("x-->")
    plt.ylabel("y-->")
    for index,point in enumerate(points):
        if (output[index] == 1.0):
            plt.plot(point[0],point[1],'r+',label='region a')
        else:
            plt.plot(point[0],point[1],'b+',label='region b')
```

```python
    def get_center_of_cluster(self, X, k):

        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        return kmeans, kmeans.cluster_centers_


    def compute_variance(self, x):

        kmeans, mu = self.get_center_of_cluster(x, 6)
        avg_sq = np.zeros((6,2))
        count = np.zeros(6)
        variance = np.zeros((6,2))
        var = np.zeros(6)

        kmeans_labels = kmeans.labels_
        for idx in range(6):
            dists = []
            for i in range(len(x)):
                if kmeans_labels[i] == idx:
                    dists.append(np.linalg.norm(x[i] - mu[kmeans_labels[i]]))
            dists = np.asarray(dists)
            var[idx] = np.var(dists)

        print "centers ", mu
        print "var",var
        return var
```

3 Case1: d = 1.0

```python
In [55]: radius = 10.0
         width = 6.0
         distance = 1.0
         num_of_training_samples = 1000
         num_of_testing_samples = 3000

         state_vector_machine = StateVectorMachine(radius,
                                                   width,
                                                   distance,
                                                   num_of_training_samples,
                                                   num_of_testing_samples)
         training_input,desired_output = state_vector_machine.get_samples("train")
         variance = state_vector_machine.compute_variance(training_input)

         variances_over_d = np.array(variance)
```

```
centers  [[  0.99105201  -4.02655455]
 [ 12.33268807  -4.36436986]
```

```
 [  0.13975611   6.51418403]
 [  6.39628465  -7.72654146]
 [  5.98556064   3.18678587]
 [ -5.86311113   3.43667292]]
var [ 0.83415765  1.24169248  0.92608868  1.19929593  0.78870699  1.12045541]
```

4 Case2: d = 0.0

```
In [56]: distance = 0.0
         state_vector_machine = StateVectorMachine(radius,
                                                    width,
                                                    distance,
                                                    num_of_training_samples,
                                                    num_of_testing_samples)
         variance = state_vector_machine.compute_variance(training_input)
         variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[  7.08366721  -7.70491033]
 [ -5.81873738   3.48255094]
 [  6.04850862   3.13700492]
 [  1.16677716  -4.30878623]
 [  0.28353139   6.4919076 ]
 [ 12.55456169  -4.0587546 ]]
var [ 1.14648796  1.12637676  0.78228403  1.01771544  0.91356654  0.91819254]
```

5 case3: d = -1.0

```
In [57]: distance = -1.0
         state_vector_machine = StateVectorMachine(radius,
                                                    width,
                                                    distance,
                                                    num_of_training_samples,
                                                    num_of_testing_samples)
         variance = state_vector_machine.compute_variance(training_input)
         variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[  1.16677716  -4.30878623]
 [ 12.55456169  -4.0587546 ]
 [ -5.86311113   3.43667292]
 [  0.13975611   6.51418403]
 [  5.98556064   3.18678587]
 [  7.08366721  -7.70491033]]
var [ 1.01771544  0.91819254  1.12045541  0.92608868  0.78870699  1.14648796]
```

6 case4: d = -2.0

In [58]:
```
distance = -2.0
state_vector_machine = StateVectorMachine(radius,
                                          width,
                                          distance,
                                          num_of_training_samples,
                                          num_of_testing_samples)
variance = state_vector_machine.compute_variance(training_input)
variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[  7.08366721  -7.70491033]
 [  0.06369367   6.52365857]
 [ -5.89665272   3.37130106]
 [  5.98556064   3.18678587]
 [  1.16677716  -4.30878623]
 [ 12.55456169  -4.0587546 ]]
var [ 1.14648796  0.94770405  1.11582425  0.78870699  1.01771544  0.91819254]
```

7 case5: d = -3.0

In [59]:
```
distance = -3.0
state_vector_machine = StateVectorMachine(radius,
                                          width,
                                          distance,
                                          num_of_training_samples,
                                          num_of_testing_samples)
variance = state_vector_machine.compute_variance(training_input)
variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[  0.13975611   6.51418403]
 [  7.08366721  -7.70491033]
 [  1.16677716  -4.30878623]
 [  5.98556064   3.18678587]
 [ -5.86311113   3.43667292]
 [ 12.55456169  -4.0587546 ]]
var [ 0.92608868  1.14648796  1.01771544  0.78870699  1.12045541  0.91819254]
```

8 case6 d = -4.0

In [60]:
```
distance = -4.0
state_vector_machine = StateVectorMachine(radius,
                                          width,
                                          distance,
                                          num_of_training_samples,
```

```
                                                    num_of_testing_samples)
         variance = state_vector_machine.compute_variance(training_input)
         variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[ -5.81873738   3.48255094]
 [  7.08366721  -7.70491033]
 [  6.04850862   3.13700492]
 [ 12.55456169  -4.0587546 ]
 [  1.16677716  -4.30878623]
 [  0.28353139   6.4919076 ]]
var [ 1.12637676  1.14648796  0.78228403  0.91819254  1.01771544  0.91356654]
```

9 case7 d= -5.0

```
In [61]: distance = -5.0
         state_vector_machine = StateVectorMachine(radius,
                                                   width,
                                                   distance,
                                                   num_of_training_samples,
                                                   num_of_testing_samples)
         variance = state_vector_machine.compute_variance(training_input)
         variances_over_d = np.row_stack((variances_over_d,variance))
```

```
centers  [[ -5.86311113   3.43667292]
 [  5.98556064   3.18678587]
 [  7.08366721  -7.70491033]
 [  0.13975611   6.51418403]
 [  1.16677716  -4.30878623]
 [ 12.55456169  -4.0587546 ]]
var [ 1.12045541  0.78870699  1.14648796  0.92608868  1.01771544  0.91819254]
```

10 case8: d = -6.0

```
In [62]: distance = -6.0
         state_vector_machine = StateVectorMachine(radius,
                                                   width,
                                                   distance,
                                                   num_of_training_samples,
                                                   num_of_testing_samples)
         variance = state_vector_machine.compute_variance(training_input)
         variances_over_d = np.row_stack((variances_over_d,variance))
```

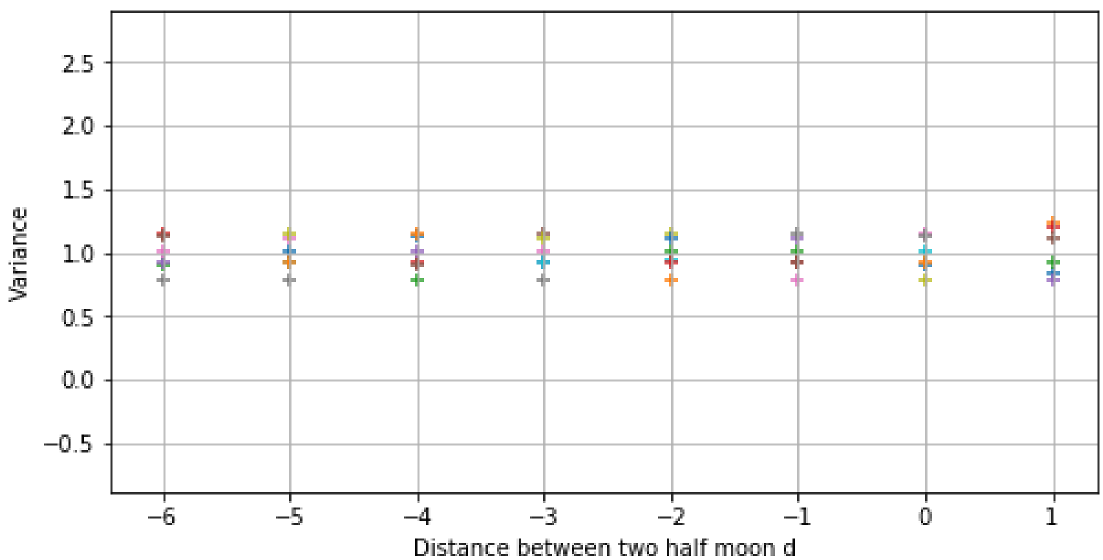
```
centers  [[  0.24507747   6.47571268]
 [  7.08366721  -7.70491033]
 [ 12.55456169  -4.0587546 ]
```

```
 [ -5.85363735   3.46265303]
 [  1.16677716  -4.30878623]
 [  6.04850862   3.13700492]]
var [ 0.91171193  1.14648796  0.91819254  1.13402508  1.01771544  0.78228403]
```

11 Plot of d vs variance:

```
In [66]: d = [1,0,-1,-2,-3,-4,-5,-6]
         fig, ax = plt.subplots(figsize=(8,4))
         ax.axis('equal')
         ax.grid()
         for i,val in enumerate(variances_over_d):
             for var in val:
                 ax.scatter(d[i],var, marker='+')
         ax.set_xlabel("Distance between two half moon d")
         ax.set_ylabel("Variance")
```

```
Out[66]: Text(0,0.5,u'Variance')
```



12 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.

(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 [34]: class RBFN:

             def __init__(self,
                           _no_of_hidden_neuron,
                          _num_of_samples,
                          _min_range,
                          _max_range,
                          _function_number,
                          _num_of_test_samples):

                 self.no_of_hidden_neuron = _no_of_hidden_neuron
                 self.num_of_samples = _num_of_samples
                 self.num_of_test_samples = _num_of_test_samples
                 self.min_range = _min_range
                 self.max_range = _max_range
                 self.function_number = _function_number

                 self.samples = self.generate_samples(_num_of_samples,_min_range,_max_range,_fun
                 self.input_data = self.samples[:,0:1]
                 self.target_data = self.samples[:,1:2]

             def generate_samples(self,number_of_samples,min_range,max_range,function_number):
                 samples = np.zeros((0,2))
                 for i in range(number_of_samples):
                     x = random.uniform(min_range, max_range)
                     samples = np.vstack([samples,[x,self.generate_target(function_number,x)]])
                 return samples

             # Given functions
             def generate_target(self,function_number,x):
                 if function_number is 1:
                     return 1/x
                 elif function_number is 2:
                     return np.log10(x)
                 elif function_number is 3:
                     return np.exp(-x)
```

```
        elif function_number is 4:
            return np.sin(x)

    # calculating cluster centers using Kmean
    def get_center_of_cluster(self, X):

        kmeans = KMeans(n_clusters=self.no_of_hidden_neuron)
        kmeans.fit(X)
        return kmeans, kmeans.cluster_centers_

    def calculate_variance(self,X,center):
        return [np.sum((X - center)**2)/(X.shape[0])]

    # calculating variances for all hidden neurons
    def get_variances(self,X,centers):
        variances = np.empty((0,1))
        for center in centers:
            variances = np.vstack([variances,self.calculate_variance(X,center)])
        return variances

    def gaussian_activation(self,X,center,variance):
        gaussian = np.empty((0,1))
        for x in X:
            result = [np.exp((-0.5 * (np.linalg.norm((x-center)))**2)/variance )]
            gaussian = np.vstack([gaussian,result])
        return gaussian

    def calculate_weight(self,gaussian,target):
        return np.dot(np.linalg.pinv(gaussian),target)

    def calcualate_f_x(self,num_of_input,weights,gaussians):
        f_x = np.zeros((num_of_input,1))
        for i in range(len(weights)):
            result = weights[i] * gaussians[i]
            f_x = f_x + result
        return f_x

    def train(self):
        self.weights = []
        gaussian_activations = []

        #Unsupervised phase, calculating means and variances
        kmeans,self.centers = self.get_center_of_cluster(self.input_data)
        self.variances = self.get_variances(self.input_data,self.centers)

        #calculating gauss activation results
        for i in range(0,self.no_of_hidden_neuron):
            gaussian_activations.append(self.gaussian_activation(self.input_data,self.c
```

```
        #Supervised phase, calculating weights
        for gaussian_activation in gaussian_activations:
            self.weights.append(self.calculate_weight(gaussian_activation,self.target_d

        #Computing final output
        f_x = self.calcualate_f_x(self.num_of_samples,self.weights,gaussian_activations
        return f_x

    def test(self):
        test_gaussian_activations = []

        test_samples = self.generate_samples(self.num_of_test_samples,_min_range,_max_r
        test_input_data = test_samples[:,0:1]
        test_target_data = test_samples[:,1:2]

        for i in range(0,self.no_of_hidden_neuron):
            test_gaussian_activations.append(self.gaussian_activation(test_input_data,
                                                                      self.centers[i],
                                                                      self.variances[i]
        f_x = self.calcualate_f_x(self.num_of_test_samples,self.weights,test_gaussian_a

        #compute mean square error from final output and desired output
        error = f_x - test_target_data
        squared_error = error ** 2
        mean_squared_error = np.mean(squared_error)

        return f_x,test_target_data, mean_squared_error
```

13 Function 1/x with different number of hidden neurons

```
In [38]: _num_of_train_samples = 2000
         _num_of_test_samples = 1000
         _min_range = 1
         _max_range = 100
         _function_number = 1

         hidden_neurons = [3,4,5,6,7]

         for hidden_neuron in hidden_neurons:
             rb = RBFN(hidden_neuron,_num_of_train_samples,_min_range,_max_range,_function_numbe

             #Training Phase
             print "training started for function f(x) = 1/x with " + str(hidden_neuron) + " hid
             rb.train()
             print "training finished"
```

```
        #Testing phase
        print "testing started for function f(x) = 1/x with " + str(hidden_neuron) + " hidd
        f_x,test_target_data, mse = rb.test()
        print "testing finished"
        print "mean squared error ", mse
        print "===========================================================================
```

```
training started for function f(x) = 1/x with 3 hidden neurons
training finished
testing started for function f(x) = 1/x with 3 hidden neurons
testing finished
mean squared error  0.0113139525128
=====================================================================================
training started for function f(x) = 1/x with 4 hidden neurons
training finished
testing started for function f(x) = 1/x with 4 hidden neurons
testing finished
mean squared error  0.0172609910191
=====================================================================================
training started for function f(x) = 1/x with 5 hidden neurons
training finished
testing started for function f(x) = 1/x with 5 hidden neurons
testing finished
mean squared error  0.0282153044964
=====================================================================================
training started for function f(x) = 1/x with 6 hidden neurons
training finished
testing started for function f(x) = 1/x with 6 hidden neurons
testing finished
mean squared error  0.0323429394507
=====================================================================================
training started for function f(x) = 1/x with 7 hidden neurons
training finished
testing started for function f(x) = 1/x with 7 hidden neurons
testing finished
mean squared error  0.0440672754296
=====================================================================================
```

14 Function exp(-x) with different number of hidden neurons

```
In [43]: _num_of_train_samples = 2000
         _num_of_test_samples = 1000
         _min_range = 1
         _max_range = 10
         _function_number = 3
```

```
    hidden_neurons = [3,4,5,6,7]

    for hidden_neuron in hidden_neurons:
        rb = RBFN(hidden_neuron,_num_of_train_samples,_min_range,_max_range,_function_numbe

        #Training Phase
        print "training started for function f(x) = exp(-x) with " + str(hidden_neuron) + "
        rb.train()
        print "training finished"

        #Testing phase
        print "testing started for function f(x) = exp(-x) with " + str(hidden_neuron) + "
        f_x,test_target_data, mse = rb.test()
        print "testing finished"
        print "mean squared error ", mse
        print "=============================================================================

training started for function f(x) = exp(-x) with 3 hidden neurons
training finished
testing started for function f(x) = exp(-x) with 3 hidden neurons
testing finished
mean squared error  0.00783856034326
=====================================================================================
training started for function f(x) = exp(-x) with 4 hidden neurons
training finished
testing started for function f(x) = exp(-x) with 4 hidden neurons
testing finished
mean squared error  0.0114087166973
=====================================================================================
training started for function f(x) = exp(-x) with 5 hidden neurons
training finished
testing started for function f(x) = exp(-x) with 5 hidden neurons
testing finished
mean squared error  0.0161574000354
=====================================================================================
training started for function f(x) = exp(-x) with 6 hidden neurons
training finished
testing started for function f(x) = exp(-x) with 6 hidden neurons
testing finished
mean squared error  0.0223797260317
=====================================================================================
training started for function f(x) = exp(-x) with 7 hidden neurons
training finished
testing started for function f(x) = exp(-x) with 7 hidden neurons
testing finished
mean squared error  0.0332260588892
=====================================================================================
```

15 Function log10(x) with different number of hidden neurons

In [39]:
```
_num_of_train_samples = 2000
_num_of_test_samples = 1000
_min_range = 1
_max_range = 10
_function_number = 2

hidden_neurons = [3,4,5,6,7]

for hidden_neuron in hidden_neurons:
    rb = RBFN(hidden_neuron,_num_of_train_samples,_min_range,_max_range,_function_numbe

    #Training Phase
    print "training started for function f(x) = log(x) with " + str(hidden_neuron) + "
    rb.train()
    print "training finished"

    #Testing phase
    print "testing started for function f(x) = log(x) with " + str(hidden_neuron) + " h
    f_x,test_target_data, mse = rb.test()
    print "testing finished"
    print "mean squared error ", mse
    print "=============================================================================
```

```
training started for function f(x) = log(x) with 3 hidden neurons
training finished
testing started for function f(x) = log(x) with 3 hidden neurons
testing finished
mean squared error  1.31266426843
=====================================================================================
training started for function f(x) = log(x) with 4 hidden neurons
training finished
testing started for function f(x) = log(x) with 4 hidden neurons
testing finished
mean squared error  3.0081643624
=====================================================================================
training started for function f(x) = log(x) with 5 hidden neurons
training finished
testing started for function f(x) = log(x) with 5 hidden neurons
testing finished
mean squared error  5.49446828447
=====================================================================================
training started for function f(x) = log(x) with 6 hidden neurons
training finished
```

```
testing started for function f(x) = log(x) with 6 hidden neurons
testing finished
mean squared error  8.82782689141
=====================================================================================
training started for function f(x) = log(x) with 7 hidden neurons
training finished
testing started for function f(x) = log(x) with 7 hidden neurons
testing finished
mean squared error  12.0094634936
=====================================================================================
```

16 Function sin(x) with different number of hidden neurons

In [42]:
```
_num_of_train_samples = 2000
_num_of_test_samples = 1000
_min_range = 1
_max_range = np.pi/2
_function_number = 4

hidden_neurons = [3,4,5,6,7]

for hidden_neuron in hidden_neurons:
    rb = RBFN(hidden_neuron,_num_of_train_samples,_min_range,_max_range,_function_numbe

    #Training Phase
    print "training started for function f(x) = sin(x) with " + str(hidden_neuron) + " 
    rb.train()
    print "training finished"

    #Testing phase
    print "testing started for function f(x) = sin(x) with " + str(hidden_neuron) + " h
    f_x,test_target_data, mse = rb.test()
    print "testing finished"
    print "mean squared error ", mse
    print "==========================================================================
```

```
training started for function f(x) = sin(x) with 3 hidden neurons
training finished
testing started for function f(x) = sin(x) with 3 hidden neurons
testing finished
mean squared error  2.31725339956
=====================================================================================
training started for function f(x) = sin(x) with 4 hidden neurons
training finished
testing started for function f(x) = sin(x) with 4 hidden neurons
testing finished
```

```
mean squared error  5.79562117483
=====================================================================================
training started for function f(x) = sin(x) with 5 hidden neurons
training finished
testing started for function f(x) = sin(x) with 5 hidden neurons
testing finished
mean squared error  10.4273695468
=====================================================================================
training started for function f(x) = sin(x) with 6 hidden neurons
training finished
testing started for function f(x) = sin(x) with 6 hidden neurons
testing finished
mean squared error  16.4923516727
=====================================================================================
training started for function f(x) = sin(x) with 7 hidden neurons
training finished
testing started for function f(x) = sin(x) with 7 hidden neurons
testing finished
mean squared error  23.7574919794
=====================================================================================
```