CSE 5526 Programming Assignment 2

Summary Report

In this project, I implemented a RBF network with the LMS rule to solve the function approximation problem. As required, my RBF neural network consists of one input variable, hidden nodes as the first layer and one output node as the second layer. The weights and biases between the hidden layer and output layer are initialized to random numbers between -1 and 1. And Gaussian function is used as the local basis function.

Additionally, online update is applied in my neural network, in which weight adjustment occurs after the presentation of each pattern.

The sample input values are taken randomly from a uniform distribution in the interval [0.0, 1.0]. The output values are generated by 75 sample data points by the function h(x)= 0.5 + 0.4*sin(2*pi*x) with added uniform noise in the interval [-0.1, 0.1].

The Gaussian centers are determined by the K-means algorithm, and the Gaussian widths are set for each cluster accordingly first. I also use the same variance for all cluster as described in the lecture.

In this project, I used the learning rate of 0.01 and 0.02 to conduct the experiments and used the number of Gaussian centers of 2,4,7,11 and 16. The number of epochs is 100. When the number of Gaussian centers is 2, the result is best. From the figures as follows, we can also see that the approximated function for number of centers 2 is the best. The fluctuation represents the balance between bias and variance. Comparing the total costs of learning rate of 0.01 and 0.02, we can find that the total cost of larger learning rate after 100 epochs is smaller than the cost of small learning rate, which means that converging rate is faster for large learning rate. The total cost results for different variances case are shown in the following Table 1 and Table 2.

Table 1. Experiment results for different number of centers (gama=0.01)

Number of centers	Total cost
2	0.19305677
4	0.6476884
7	0.47469544
11	0.45124812
16	0.51406316

Table 2. Experiment results for different number of centers (gama=0.02)

Number of centers	Total cost
2	0.18587388
4	0.57868345
7	0.47375369
11	0.33072235
16	0.33351759

The plots for the sample data points for different variance case, the original function and the function generated by the RBF network are shown as follows:

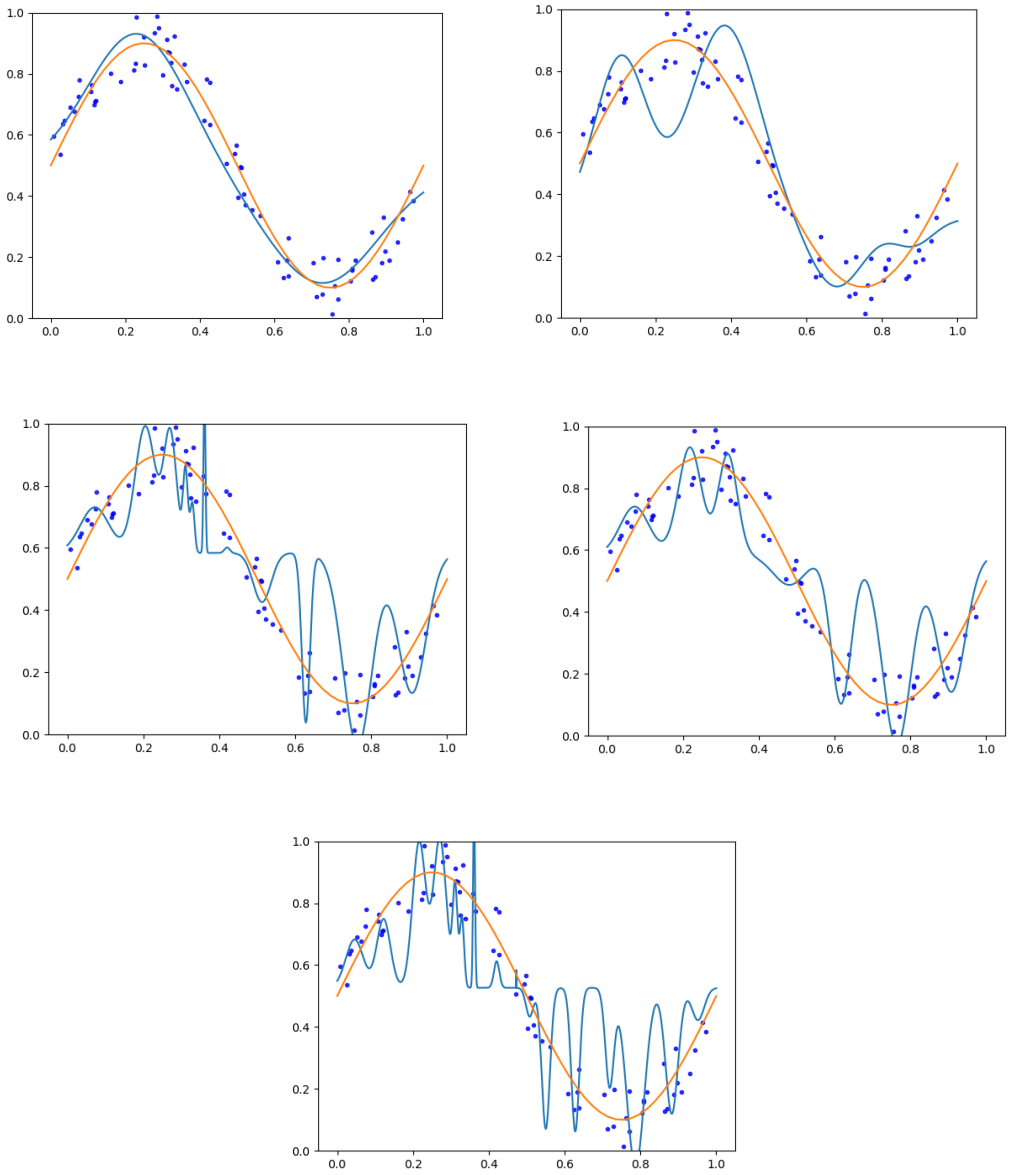

Fig1. Plots of different number of centers for gama=0.01

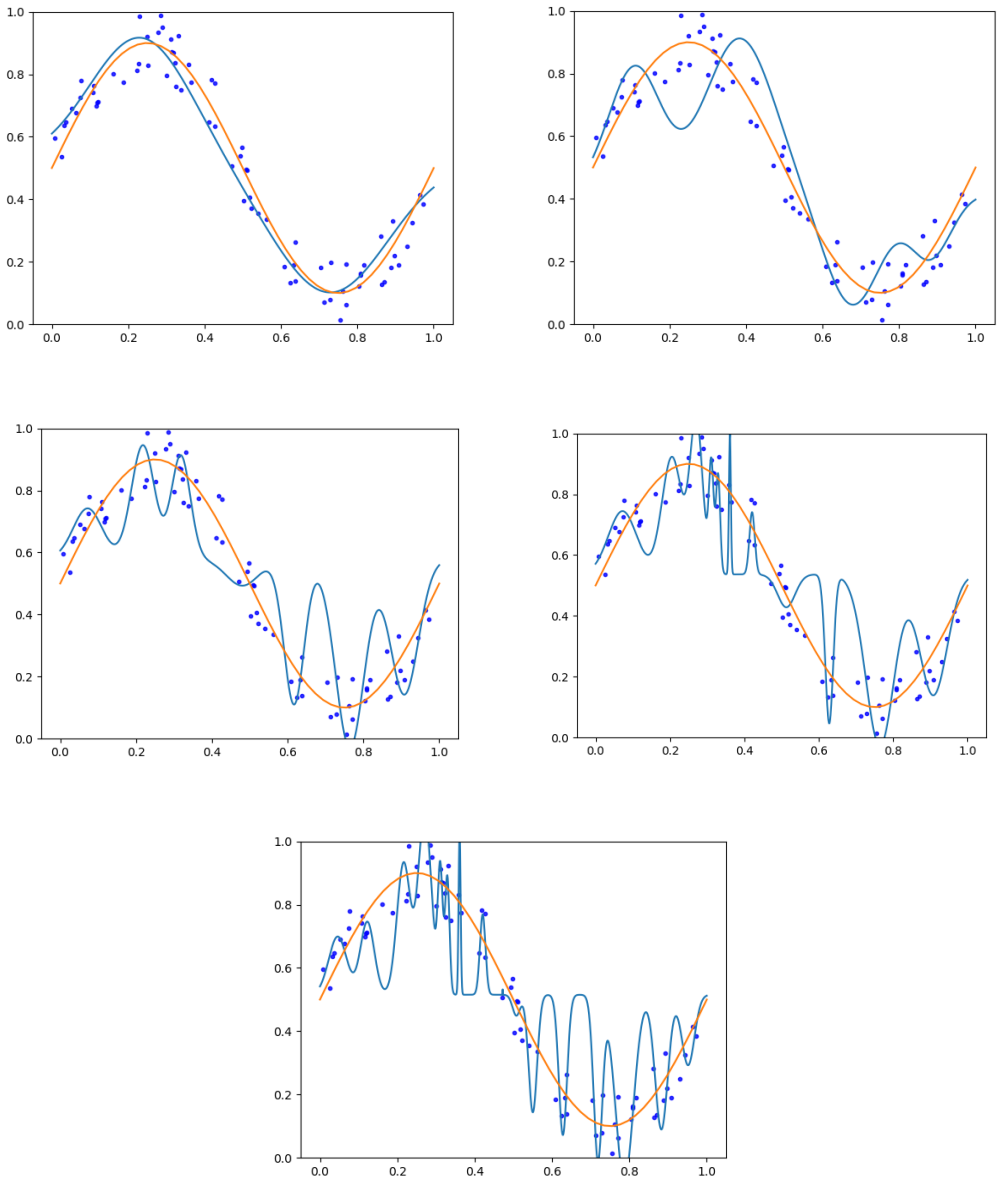

Fig2. Plots of different number of centers for gama=0.02

In the figures, blue dots stand for the sample data, blue line is the function generated by RBF network, and orange line represents the original function.

I also did the experiments using the same variance of all clusters. The total cost results are shown below:

Table 3. Experiment results for different number of centers (gama=0.01)

Number of centers	Total cost
2	0.3125
4	0.5192
7	0.1703
11	0.1312
16	0.1683

Table 4. Experiment results for different number of centers (gama=0.02)

Number of centers	Total cost
2	0.3160
4	0.4044
7	0.1292
11	0.1302
16	0.1584

The plots for the sample data points for same variance case, the original function and the function generated by the RBF network are shown as follows:

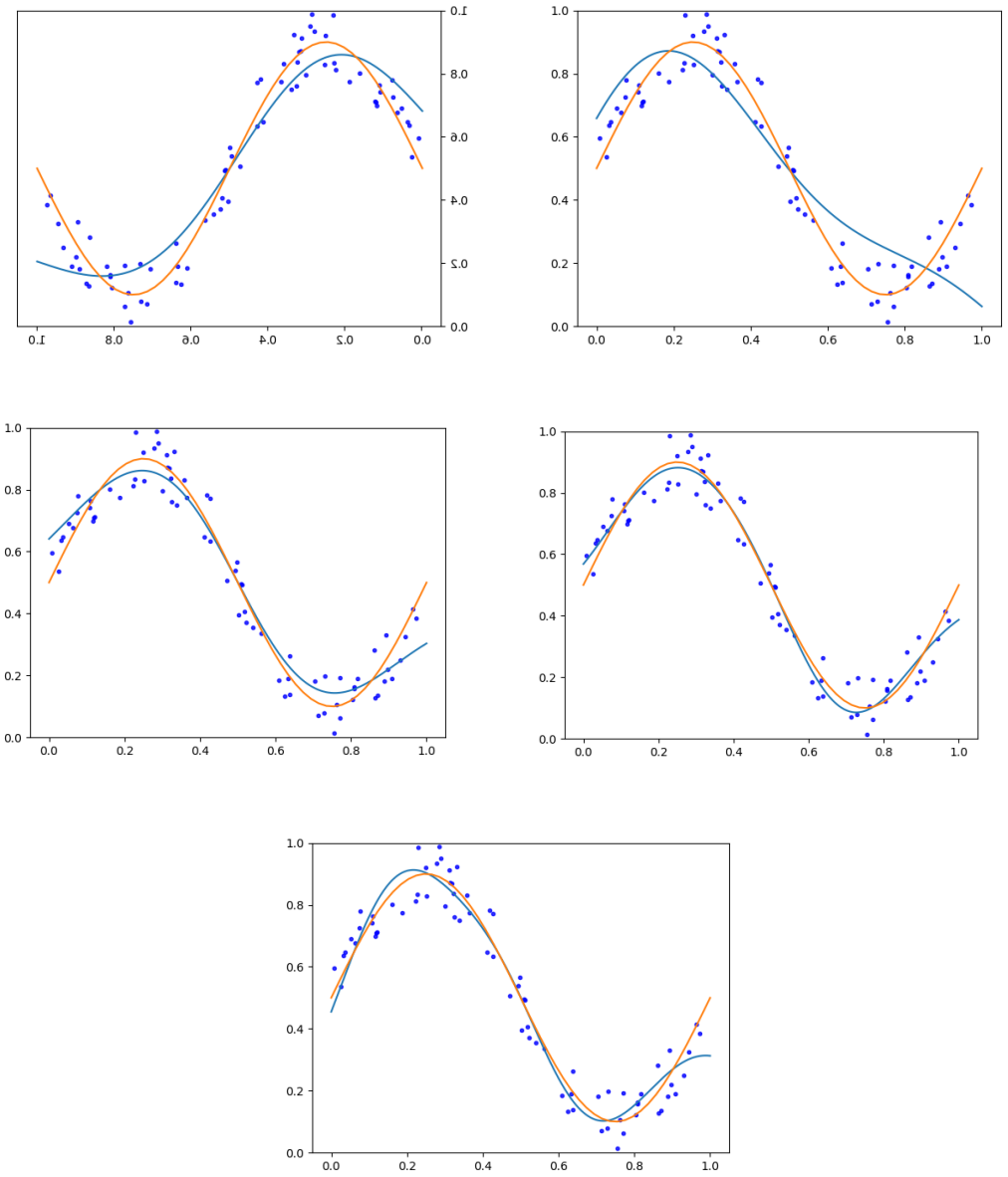

Fig3. Plots of different number of centers for gama=0.01

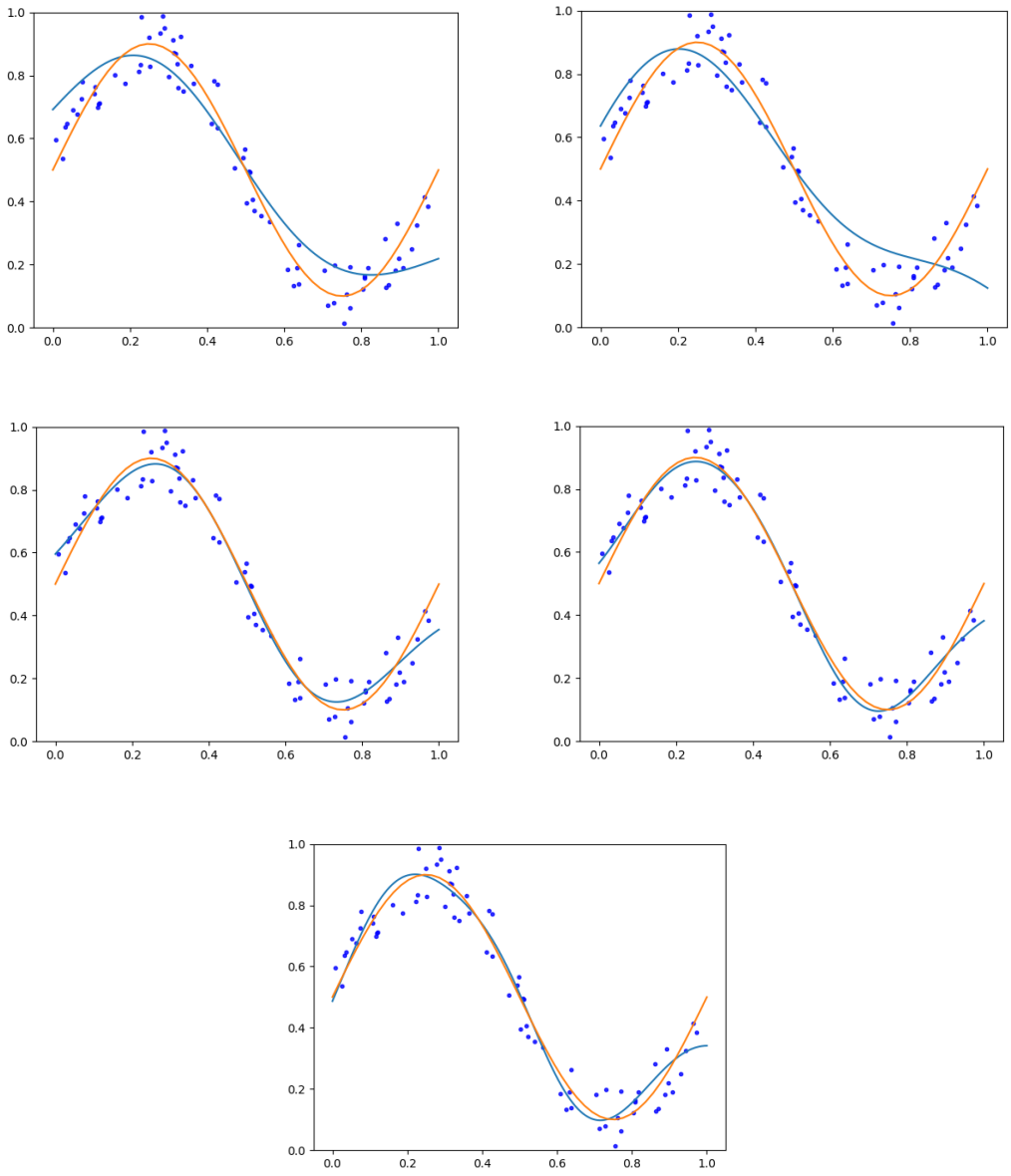

Fig4. Plots of different number of centers for gama=0.02

In the same variance case, we can also see the same trend of total cost as increasing the number of centers. And larger gama generates faster converging rate. Comparing different variance and same variance case, I found that the fitting results of same variance case is significantly better than different variance case from both the total cost value and the plots above. The converging rate of same variance case is faster. The fitting figure for same variance case is smoother.

Source Code:

```
#####          CSE 5526 Introduction to Neural Networks          #####
#####               Programming Assignment 2      RBF               #####
#####                              Online Learning                          #####

import numpy as np
import matplotlib.pyplot as plt

class Cluster:
    def __init__(self,data,num_elements):
        self.data=data
        self.num_elements=num_elements
    def __getdata__(self): # get the input data of this input node
        return self.data

def find_cluster(x,center_list,num_cluster):
    # Find the index of cluster in which x is
    # Besides the index of cluster, also return the distance list to each of the centers
    dist_list=[]
    for i in range(num_cluster):
        dist=abs(x-center_list[i])
        dist_list.append(dist)
    min_dist=min(dist_list)
    index_cluster=dist_list.index(min_dist)
    return (index_cluster,dist_list)

def Kmeans(k,input_x,sample_size):
    a = np.arange(75)
    np.random.shuffle(a)
    index_center_list=a[0:k]
    center_list=np.array([input_x[index_center] for index_center in index_center_list]).reshape(k,)
    delta_list=np.array([-1.0 for i in range(k)]).reshape(k,)
    index_cluster_sample=np.ones(sample_size)*(-1)

    while True:
        # calculate the index of cluster for each of the input patterns
        for i in range(sample_size):
            (index_cluster_sample[i],dist_test)=find_cluster(input_x[i],center_list,k)
            pass

        center_list_new=np.ones(k,)*(-99)
        # Update cluster centers
        for j in range(k):
            sum_x=0
```

```
                count_j=0
                for i in range(sample_size):
                    if index_cluster_sample[i]==j:
                        sum_x=sum_x+input_x[i]
                        count_j=count_j+1
                center_list_new[j]=sum_x/count_j
            dif0=abs(center_list_new[0]-center_list[0])
            dif1=abs(center_list_new[1]-center_list[1])
            print('dif0 %s, dif1 %s\n '%(dif0, dif1))
            if np.array_equal(center_list_new,center_list):
                break
            center_list=center_list_new
        # calculate the variance of each cluster(delta)
        delta_sum=0
        num_cluster_nonzero=0
        for j in range(k):
            sum_dif_2=0
            count_j=0
            for i in range(sample_size):
                if index_cluster_sample[i]==j:
                    sum_dif_2=sum_dif_2+(input_x[i]-center_list[j])**2
                    count_j=count_j+1
            if count_j!=0:
                delta=sum_dif_2/count_j
                delta_list[j]=np.sqrt(delta)
                delta_sum=delta_sum+delta
                num_cluster_nonzero=num_cluster_nonzero+1

        # update the delta in list if the count is zero
        for j in range(k):
            if delta_list[j]==-1 or delta_list[j]==0:
                delta=delta_sum/num_cluster_nonzero
                delta_list[j]=delta

        return (center_list_new,delta_list)

def max_distance(center_list,k):
        dmax=0
        for i in range(k):
            for j in range(k):
                if i!=j:
                    d=abs(center_list[i]-center_list[j])
                    if d>dmax:
                        dmax=d
```

```
    return dmax

# Activation functions
def gaussian(x,xj,delta):
    dif_2=(x-xj)**2
    phi=np.exp(-dif_2/(2*delta**2))
    return phi

# Derivative of activation function
def gaussianDer(x,xj,delta):
    coef=-abs(x-xj)/(delta**2)
    dif_2=(x-xj)**2
    expv=-dif_2/(2*delta**2)
    phi_prime=coef*np.exp(expv)
    return phi_prime

# Forward Process between input and hidden layer
def Forward_in2hi(input_x, xj_list,delta_list,k):
    result = [gaussian(input_x,xj_list[i],delta_list[i]) for i in range(k)]
    result = np.array(result).reshape(k,1)
    return result

# Forward Process between hidden and output layer
def Forward_hi2op(w_hi2op, b_op, yj,k):
    yj=yj.reshape(k,1)
    w_hi2op = w_hi2op.reshape(k,)
    yj = yj.reshape(k,)
    tmp = np.sum(w_hi2op * yj) + b_op
    return (tmp)

# The whole forward process
def Forward(input_x, xj_list,delta_list, w_hi2op, b_op,k):
    yj = Forward_in2hi(input_x, xj_list,delta_list,k)
    act_output = Forward_hi2op(w_hi2op, b_op, yj,k)
    return act_output

# BackPropogation between output layer and hidden layer
def BackProp_op2hi(input_x, xj_list,w_hi2op, b_op, exp_output, act_output,k):
    yj= Forward_in2hi(input_x,xj_list,delta_list, k)
    act_output = Forward_hi2op(w_hi2op, b_op, yj,k)
    # vk = sum(np.multiply(w_hi2op.reshape(k,), yj.reshape(k,)))
    dw_hi2op = -(exp_output-act_output) * yj
    dw_hi2op=np.array(dw_hi2op).reshape(k,1)
    db_op =    -(exp_output-act_output) * 1
```

```
    return (dw_hi2op, db_op)

# Update the weights based on several parameters
def Weight_update( w_hi2op, b_op , dw_hi2op, db_op, gama=0.02):
    dw_hi2op = gama * dw_hi2op
    db_op = gama * db_op
    return (w_hi2op-dw_hi2op,b_op- db_op,dw_hi2op, db_op)

# Cost/loss function
def CostFunction(act_output,exp_output):
    ESquare=((act_output-exp_output)**2)*0.5
    return ESquare

# Initialize the input and output data for training samples
sample_size=75
np.random.seed(42)
noise=np.random.uniform(-0.1,0.1,sample_size).reshape(sample_size,1)
input_x=np.random.uniform(0,1,sample_size).reshape(sample_size,1)
output_h=0.4*np.sin(2*np.pi*input_x)+0.5+noise

K=[2,4,7,11,16]
lr=[0.01,0.02]
k=K[4]
l=lr[1]
# get the centers and deltas for each of the clusters
(center_list,delta_list)=Kmeans(k,input_x,sample_size)

# This part is for the same variance for all clusters
dmax=max_distance(center_list,k)
delta=dmax/(np.sqrt(2*k))
delta_list=[ delta for i in range(len(delta_list))]

# initialize weights and bias
np.random.seed(42)
w_hi2op = np.random.rand(k,1)*2-1
b_op = np.random.rand()*2-1

for index_epoch in range(100):
    cost_total = 0
    dw_hi2op = np.array([0.0 for i in range(k)]).reshape(k,1)
    db_op=0
    for index_sample in range(sample_size):
        x=input_x[index_sample]
        exp_output=output_h[index_sample]
```

```
            # update weights and bias
            act_output = Forward(x, center_list,delta_list, w_hi2op, b_op,k)
            (dw_hi2op,db_op)=BackProp_op2hi(x,center_list,w_hi2op,b_op,exp_output,act_output,k)
            (w_hi2op, b_op, dw_hi2op, db_op) = Weight_update(w_hi2op, b_op, dw_hi2op,
db_op,gama=l)
        # Calculate the total cost for each of the sample data
        for index_sample in range(sample_size):
            x=input_x[index_sample]
            exp_output=output_h[index_sample]
            act_output = Forward(x, center_list,delta_list, w_hi2op, b_op,k)
            cost_total += CostFunction(act_output, exp_output)

print('Gama is %s, num of clusters is %s' % (l, k))
print('Index of epoch %s: Total cost is %s' % (index_epoch, cost_total))

# Draw the function generated by the RBF network
input_x_final=np.linspace(0,1,1000)
final_output=[]
for index_sample in range(len(input_x_final)):
        x=input_x_final[index_sample]
        act_output = Forward(x, center_list,delta_list, w_hi2op, b_op,k)
        final_output.append(act_output)
final_output=np.array(final_output)
plt.plot(input_x_final,final_output)

# Draw the sample points
ax = plt.gca()
ax.scatter(list(input_x),list(output_h) , color='blue', marker='.', alpha=0.8)

#Draw the original function
x_list=np.linspace(0,1)
y_list=0.4*np.sin(2*np.pi*x_list)+0.5
plt.plot(x_list, y_list)
ax.set_ylim([0, 1])
plt.show()
```