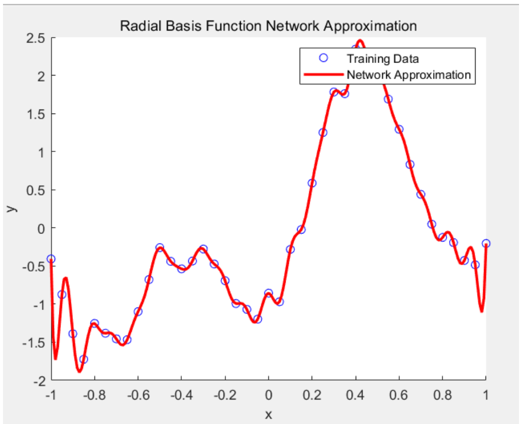
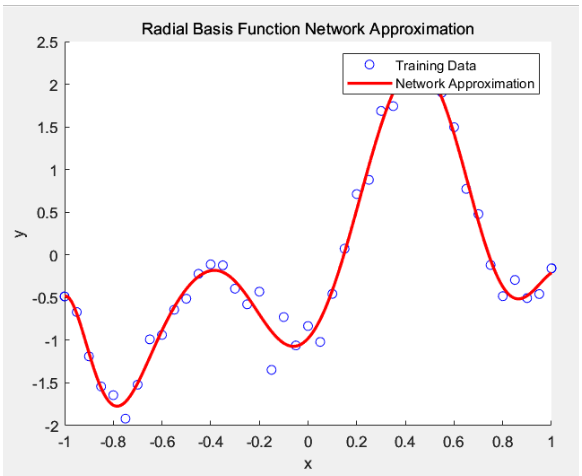
Q1:

a) Using exact interpolation method and Gaussian function with standard deviation of 0.1.



There was an obvious problem of being over-fitting. The noise signal largely spoiled the fitting results.

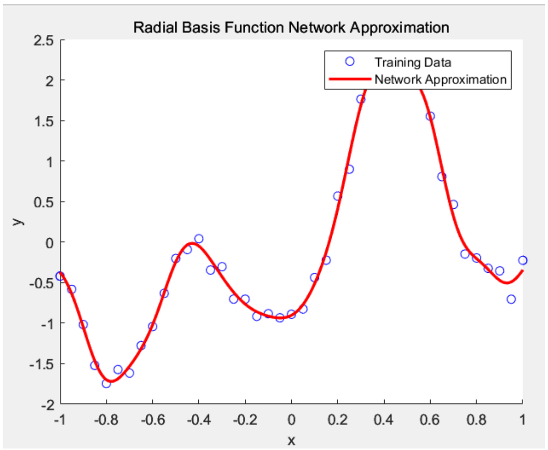
b) Using the method ‘Fixed centers selected at random.’

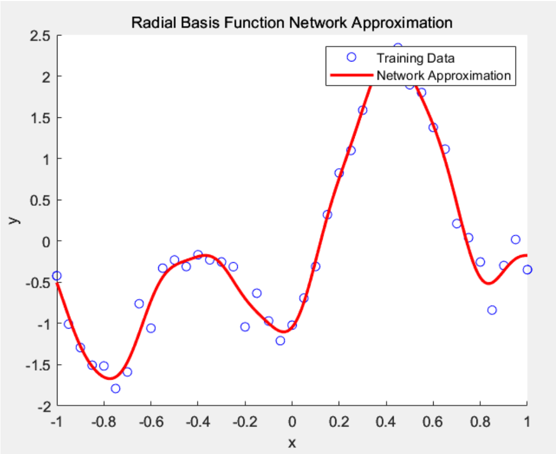


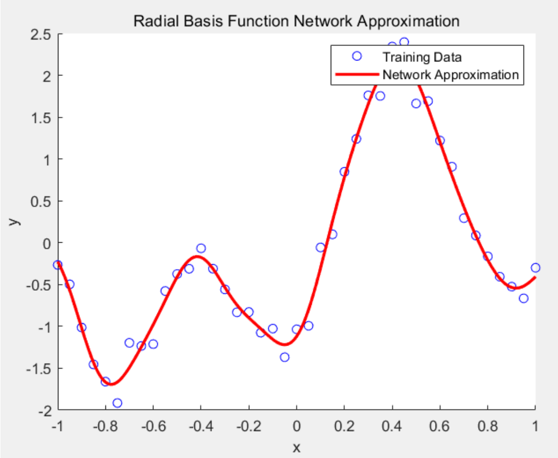
The noise didn’t cause the twisted curves. The fitting results performed well.

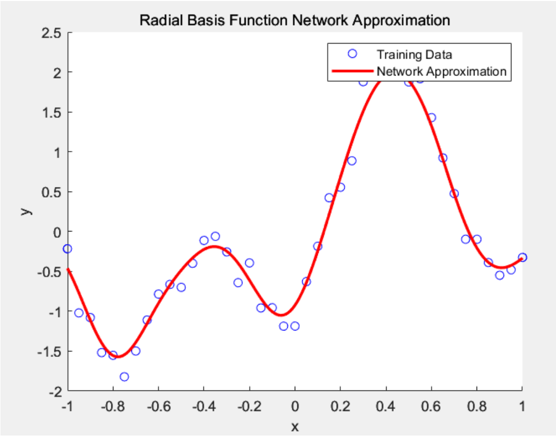
c)

set the regularization parameter (lambda)= 0.005 0.01 0.1 1.





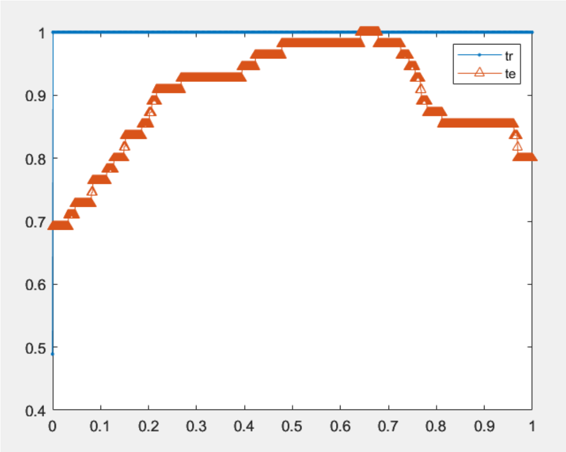




The regularization method helped to remove the noise signal and make the curve smooth. This algorithm showed the good fitting performances.

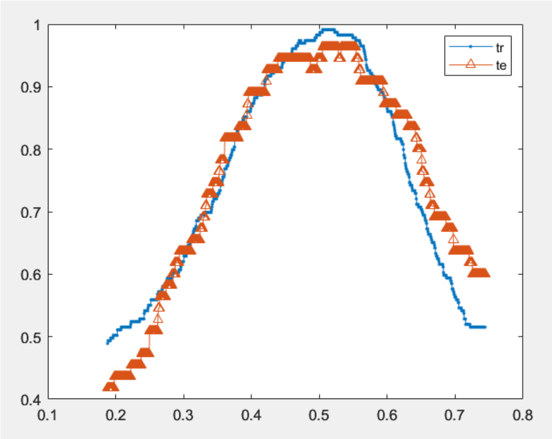
Q2:

a) Using exact interpolation method. Assume the RBF is Gaussian function with standard deviation 100. Vary the regularization parameter from 0 to 10. Besides using the evaluation method provided in the homework script, I used 'round( )' to get the final prediction results for further evaluations.



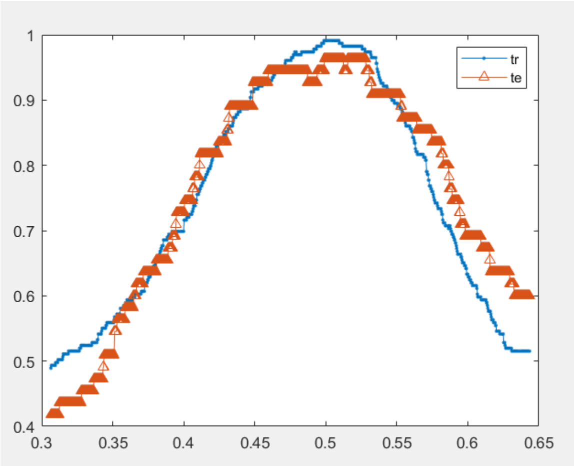

```
============test
Accuracy 0.96364
there are 31 successful predictions of 0 in 55 input cases
there are 22 successful predictions of 1 in 55 input cases
============
```

Regularization parameter = 0



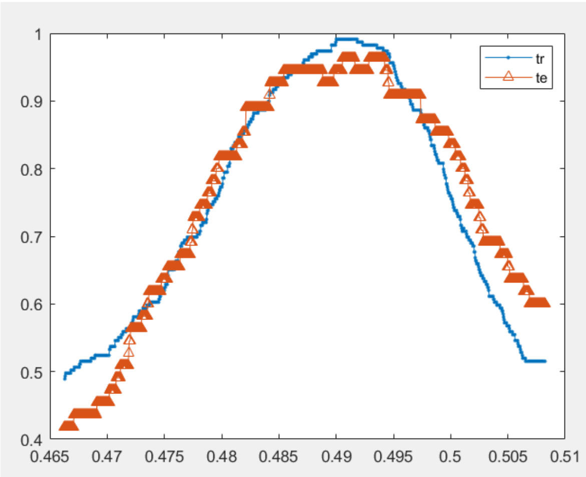

```
============test
Accuracy 0.94545
there are 31 successful predictions of 0 in 55 input cases
there are 21 successful predictions of 1 in 55 input cases
============
```

Regularization parameter = 0.05



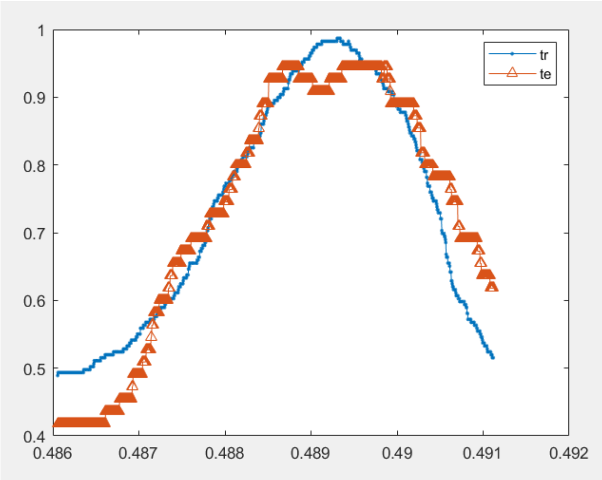

```
============test
Accuracy 0.96364
there are 32 successful predictions of 0 in 55 input cases
there are 21 successful predictions of 1 in 55 input cases
============
```

Regularization parameter = 0.1




```
============test
Accuracy 0.83636
there are 33 successful predictions of 0 in 55 input cases
there are 13 successful predictions of 1 in 55 input cases
============
```

Regularization parameter = 1



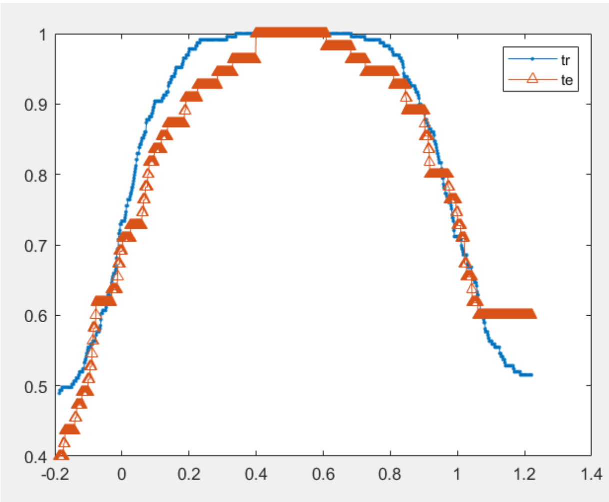

```
============test
Accuracy 0.6
there are 33 successful predictions of 0 in 55 input cases
there are 0 successful predictions of 1 in 55 input cases
============
```

Regularization parameter = 1

Basically, for the exact interpolation method, the regularization didn't help to improve the performances.

b)

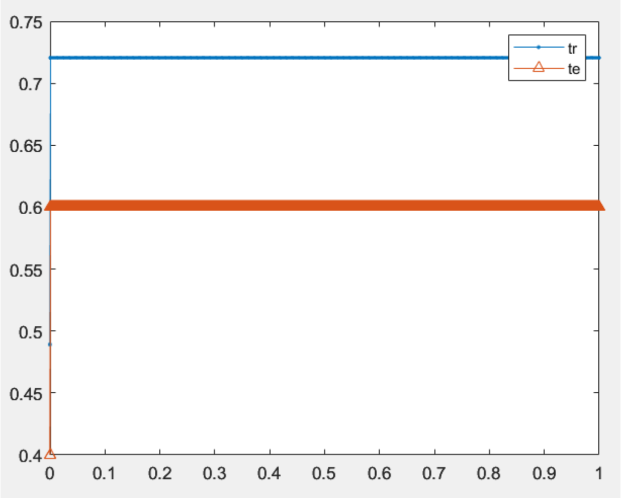
Using the method 'Fixed centers selected at random.' The original value of width was 10.



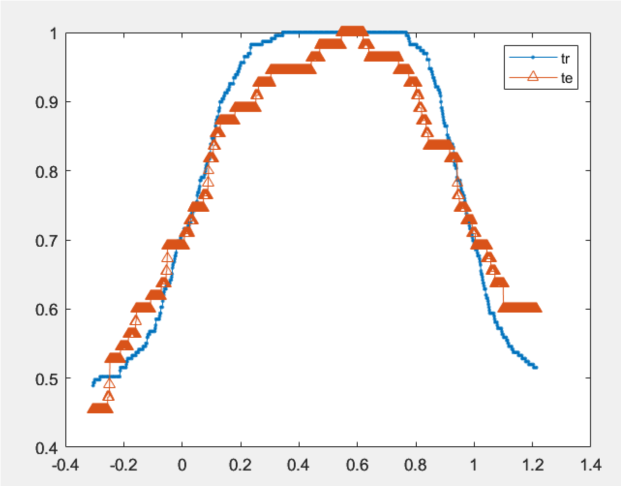

```
============test
Accuracy 1
there are 33 successful predictions of 0 in 55 input cases
there are 22 successful predictions of 1 in 55 input cases
============
```

The value of width = 10

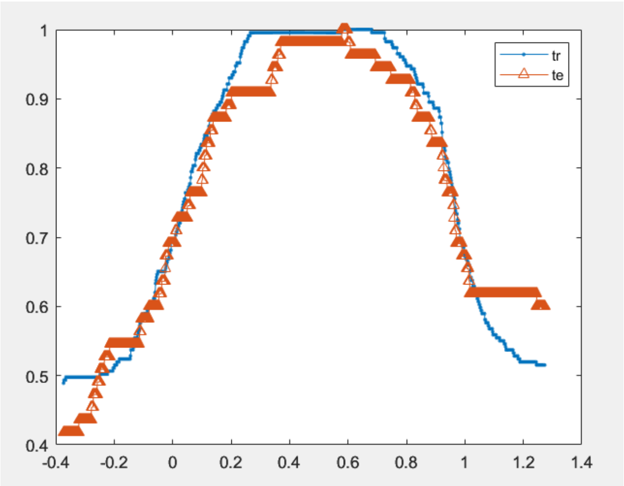
The performance was very good with a large platform. Compared with the predictions got from question a), the algorithm was not so good because it only employed 100 centers in 'Fixed center selected at random'.



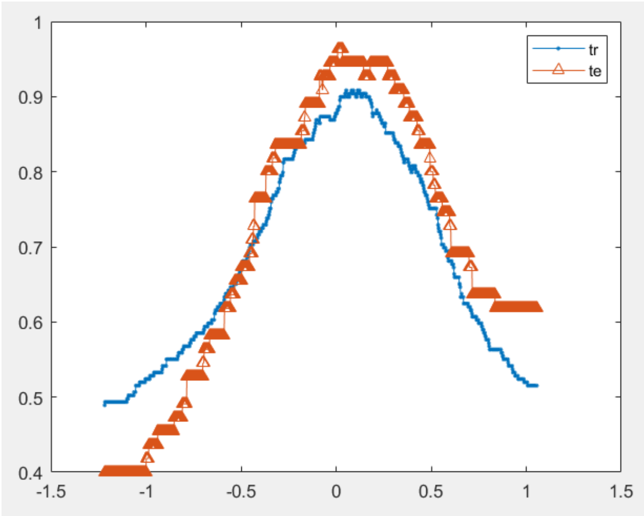
The value of width = 1



The value of width = 100



The value of width = 1000

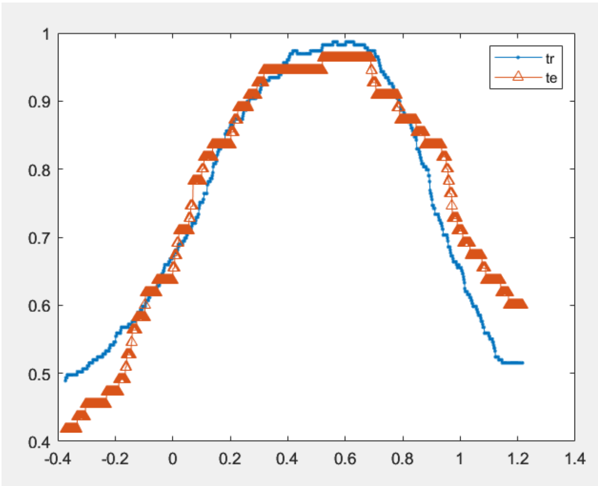


The value of width = 10000

Very small value of width would cause the failure of RBFNs. Very large one would decrease the accuracy dramatically.

c)

Try classical “K-Mean Clustering” method with two centers.




```
============test
Accuracy 0.94545
there are 30 successful predictions of 0 in 55 input cases
there are 22 successful predictions of 1 in 55 input cases
============
```

The platform was a little narrow compared with above trials. But generally speaking, the prediction performances were good.

It was very hard to visualize the obtained centers because the centers were N dimensional vectors which could not be plotted.

centers 2x784 double

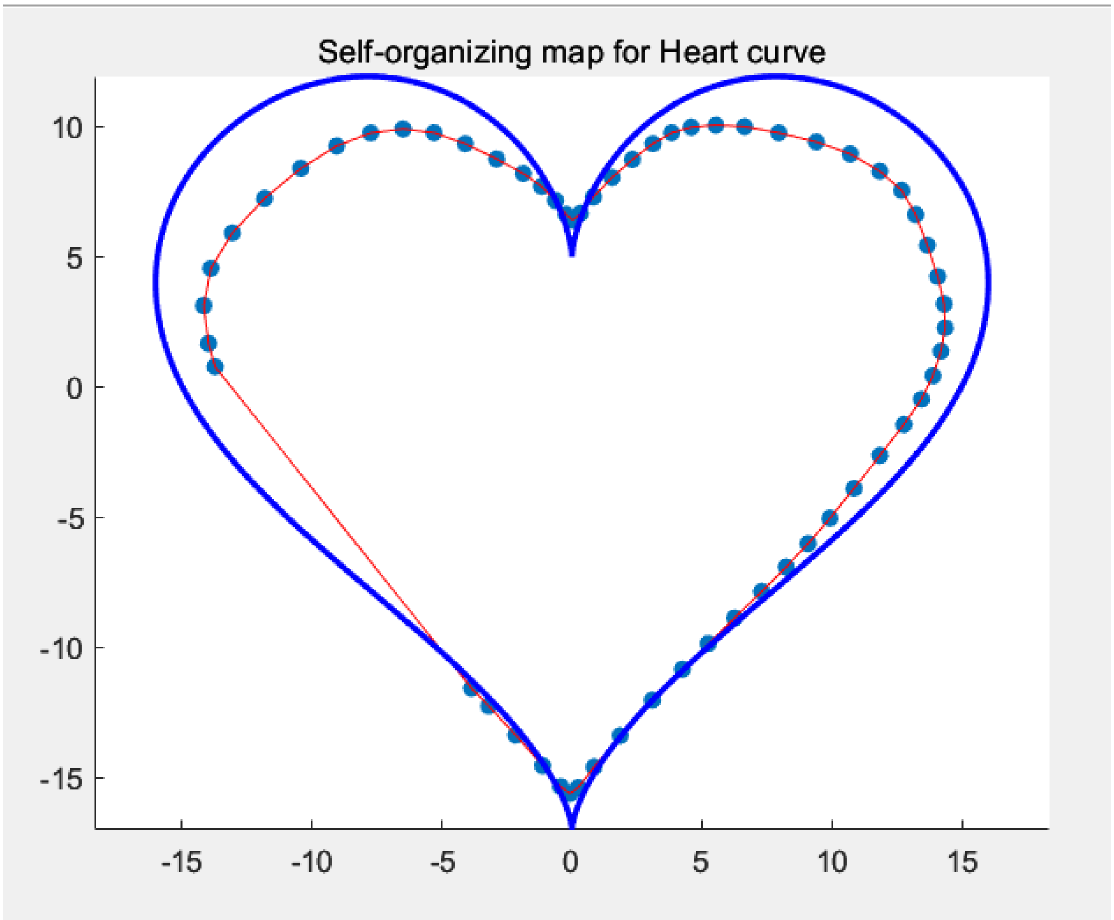
I didn’t compare the centers with the mean of images’ values from 2 different classes. But it could be predicted that these two numbers were different. The k-means centers were got from iterations and were not easily influenced by some particular images' values. The mean number revealed the whole situation, which was similar to the batch mode.

Q3

a)

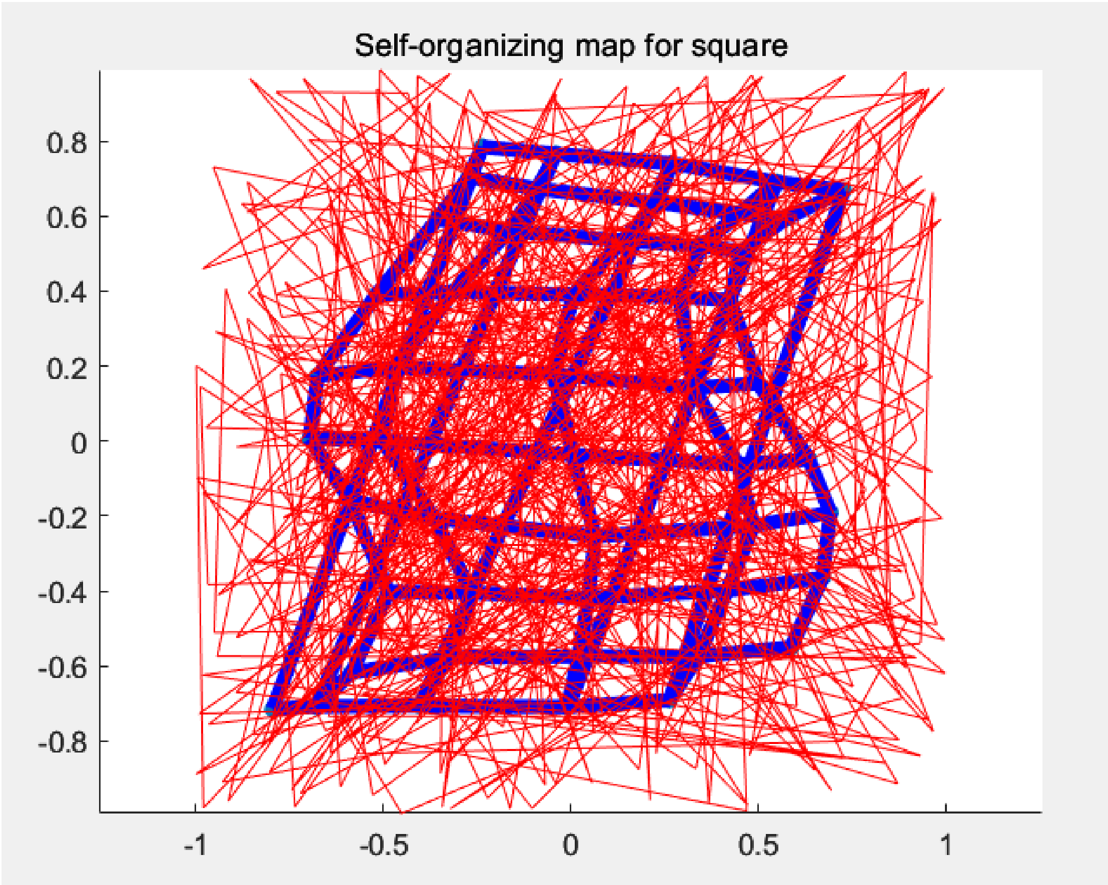
I implemented a SOM that mapped a 1-dimensional output layer of 60 neurons to a "heart curve". I adjusted some parameters for this better performance.

```
% Set up the SOM parameters
num_neurons = 60;
input_dim = 2;
num_epochs = 800;
learning_rate_initial = 0.1;
sigma_initial=30;
```

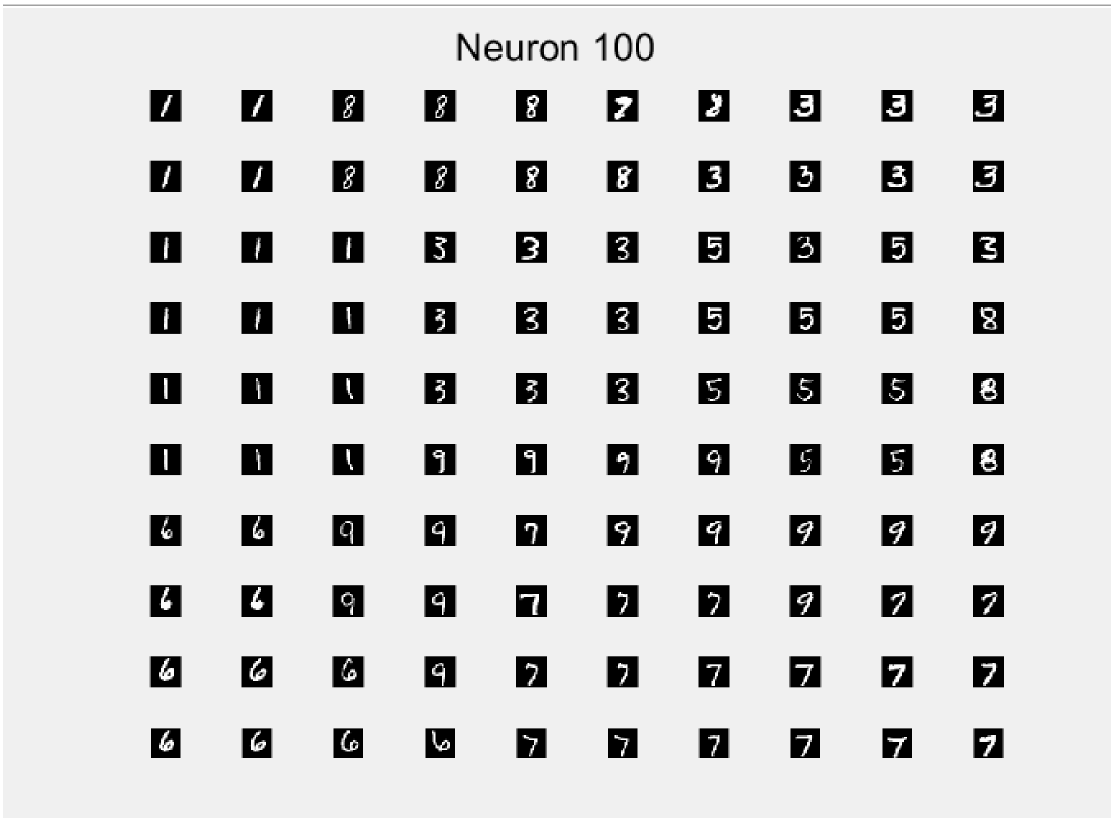


b)

Mapping a 2-dimensional output layer of 25 (i.e. 5×5) neurons to a "square".



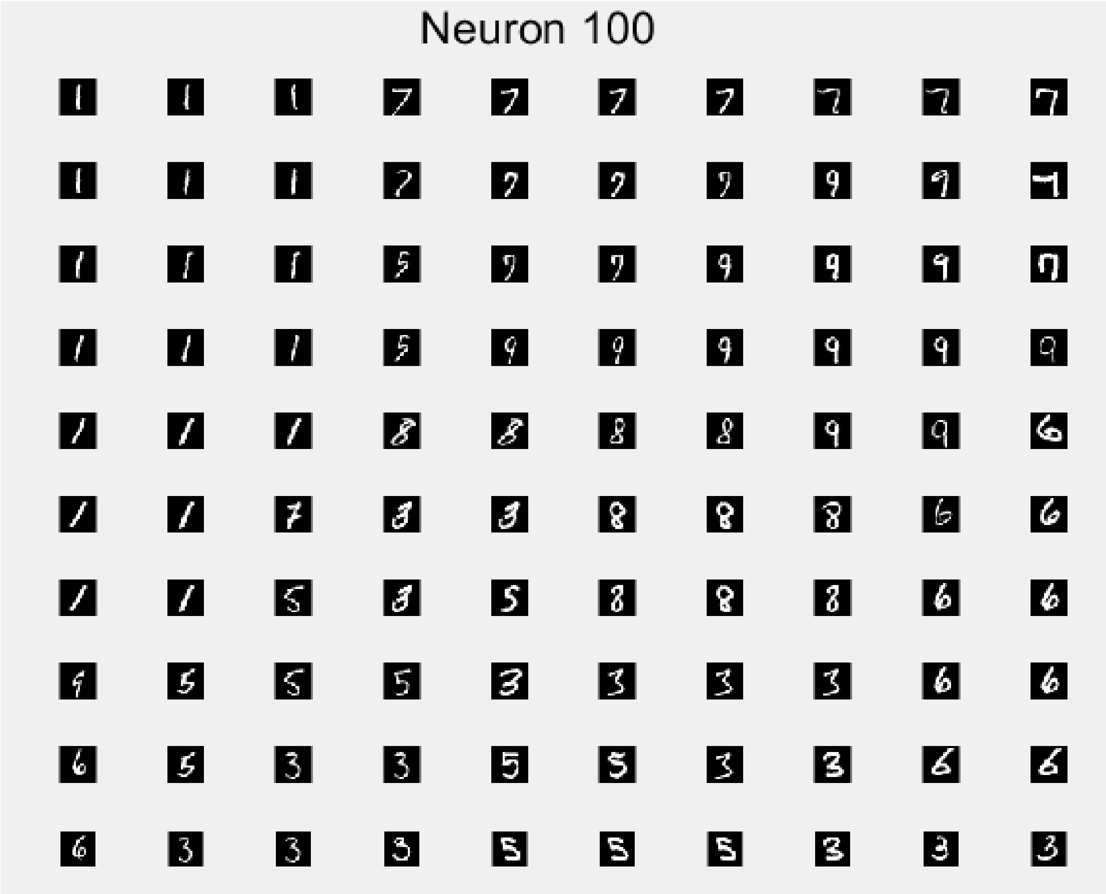
C1)



It could be easily discovered that the similarity among the neighboring images.

C2)

```
============test
Accuracy 0.7052
there are 31 successful predictions of 1 in 173 input cases
there are 0 successful predictions of 2 in 173 input cases
there are 16 successful predictions of 3 in 173 input cases
there are 0 successful predictions of 4 in 173 input cases
there are 11 successful predictions of 5 in 173 input cases
there are 19 successful predictions of 6 in 173 input cases
there are 16 successful predictions of 7 in 173 input cases
there are 12 successful predictions of 8 in 173 input cases
there are 17 successful predictions of 9 in 173 input cases
============train
Accuracy 0.20482
there are 28 successful predictions of 1 in 664 input cases
there are 0 successful predictions of 2 in 664 input cases
there are 22 successful predictions of 3 in 664 input cases
there are 0 successful predictions of 4 in 664 input cases
there are 13 successful predictions of 5 in 664 input cases
there are 18 successful predictions of 6 in 664 input cases
there are 18 successful predictions of 7 in 664 input cases
there are 12 successful predictions of 8 in 664 input cases
there are 25 successful predictions of 9 in 664 input cases
```



For the training dataset, the accuracy was quite low, although the semantic map had possible hidden pattern. But, for the testing dataset, the neuron could still stand for some characteristics inside the content of images. So, for the testing dataset, the accuracy could be larger than 70%.

Attachment Q1: a) c)

```
clear all
clc

noise=randn(1,41);
i=1;
for x=-1:0.05:1
    y=1.2*sin(pi*x)-cos(2.4*pi*x);
    train_set(i,1)=x;
    train_set(i,2)=y+noise(i)/6;
    i=i+1;
end

i=1;
for x=-1:0.01:1
    y=1.2*sin(pi*x)-cos(2.4*pi*x);
```

```
    test_set(i,1)=x;
    test_set(i,2)=y;
    i=i+1;
end

% Define the training inputs and targets
x_train = train_set(:,1);
y_train = train_set(:,2);

% Define the radial basis function with a Gaussian activation function
rbf = @(x, c, sigma) exp(-(x-c).^2/(2*sigma^2));

% Define the number of radial basis functions to use
num_rbfs = size(train_set,1);

% Select the centers of the radial basis functions randomly from the
training inputs
centers = datasample(x_train, num_rbfs, 'Replace', false);

% Define the standard deviation of the Gaussian activation function
sigma = 0.1;

% Compute the radial basis function values for each training input and
center
phi = zeros(length(x_train), num_rbfs);
for i = 1:num_rbfs
    phi(:,i) = rbf(x_train, centers(i), sigma);
end

% Solve for the network weights using the Moore-Penrose pseudoinverse
lambda=1;
w=inv(phi'*phi+lambda*eye(size(phi'*phi,1),size(phi'*phi,2)))*phi'*y_train;

% Define the test inputs and compute the corresponding radial basis
function values
x_test = test_set(:,1);
phi_test = zeros(length(x_test), num_rbfs);
for i = 1:num_rbfs
    phi_test(:,i) = rbf(x_test, centers(i), sigma);
end

% Compute the network output for the test inputs using the learned weights
and radial basis functions
y_test = phi_test * w;
```

```
% Plot the training and testing data, as well as the network approximation
figure;
hold on;
plot(x_train, y_train, 'bo');
plot(x_test, y_test, 'r-', 'LineWidth', 2);
legend('Training Data', 'Network Approximation');
xlabel('x');
ylabel('y');
title('Radial Basis Function Network Approximation');
```

Attachment 2: Q1 b)

```
clear all
clc

noise=randn(1,41);
i=1;
for x=-1:0.05:1
    y=1.2*sin(pi*x)-cos(2.4*pi*x);
    train_set(i,1)=x;
    train_set(i,2)=y+noise(i)/6;
    i=i+1;
end

i=1;
for x=-1:0.01:1
    y=1.2*sin(pi*x)-cos(2.4*pi*x);
    test_set(i,1)=x;
    test_set(i,2)=y;
    i=i+1;
end

% Define the training inputs and targets
x_train = train_set(:,1);
y_train = train_set(:,2);

% Define the radial basis function with a Gaussian activation function
rbf = @(x, c, sigma) exp(-(x-c).^2/(2*sigma^2));

% Define the number of radial basis functions to use
num_rbfs = 15;
```

```
% Select the centers of the radial basis functions randomly from the
training inputs
centers = datasample(x_train, num_rbfs, 'Replace', false);

% Define the standard deviation of the Gaussian activation function
sigma = (max(centers)-min(centers))/sqrt(2*num_rbfs);

% Compute the radial basis function values for each training input and
center
phi = zeros(length(x_train), num_rbfs);
for i = 1:num_rbfs
    phi(:,i) = rbf(x_train, centers(i), sigma);
end

% Solve for the network weights using the Moore-Penrose pseudoinverse
w = pinv(phi) * y_train;

% Define the test inputs and compute the corresponding radial basis
function values
x_test = test_set(:,1);
phi_test = zeros(length(x_test), num_rbfs);
for i = 1:num_rbfs
    phi_test(:,i) = rbf(x_test, centers(i), sigma);
end

% Compute the network output for the test inputs using the learned weights
and radial basis functions
y_test = phi_test * w;

% Plot the training and testing data, as well as the network approximation
figure;
hold on;
plot(x_train, y_train, 'bo');
plot(x_test, y_test, 'r-', 'LineWidth', 2);
legend('Training Data', 'Network Approximation');
xlabel('x');
ylabel('y');
title('Radial Basis Function Network Approximation');
```

Attachment 3: Q2 a) b)

```
clear all
clc
```

```
load MNIST_database.mat;
% train_data = training data, 784x1000 matrix
% train_classlabel = the labels of the training data, 1x1000 vector
% test_data = test data, 784x250 matrix
% train_classlabel = the labels of the test data, 1x250 vector


trainIdx = find(train_classlabel==2 | train_classlabel==4); % find the
location of classes 0, 1, 2
Train_ClassLabel = train_classlabel(trainIdx)';
for tmp=1:length(Train_ClassLabel)
    if Train_ClassLabel(tmp)==2
        Train_ClassLabel(tmp)=0
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    else
        Train_ClassLabel(tmp)=1
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    end
end
Train_Data = train_data(:,trainIdx);

trainIdx2 = find(test_classlabel==2 | test_classlabel==4); % find the
location of classes 0, 1, 2
Test_ClassLabel = test_classlabel(trainIdx2)';
for tmp=1:length(Test_ClassLabel)
    if Test_ClassLabel(tmp)==2
        Test_ClassLabel(tmp)=0
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    else
        Test_ClassLabel(tmp)=1
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    end
end
Test_Data = test_data(:,trainIdx2);

% Define the training inputs and targets
x_train = Train_Data;
y_train = Train_ClassLabel;
```

```
% Define the radial basis function with a Gaussian activation function
rbf = @(x, sigma) exp(-x.^2/(2*sigma^2));

% Define the number of radial basis functions to use
num_rbfs = 100;

% Select the centers of the radial basis functions randomly from the
training inputs
[centers,idx] = datasample(x_train', num_rbfs, 'Replace', false);

% Define the standard deviation of the Gaussian activation function
sigma = 10000;

% Compute the radial basis function values for each training input and
center
% phi = zeros(size(x_train,2), num_rbfs);
for i = 1:num_rbfs

    for j = 1 : size(x_train,2)
        Eucdistance1(j,:)=pdist([x_train(:,j)';centers(i,:)]);
    end
    phi(:,i) = rbf(Eucdistance1, sigma);
end

% Solve for the network weights using the Moore-Penrose pseudoinverse
lambda=0;
w=inv(phi'*phi+lambda*eye(size(phi'*phi,1),size(phi'*phi,2)))*phi'*y_train;

Trian_pred=phi*w;

% Define the test inputs and compute the corresponding radial basis
function values
x_test = Test_Data;

for i = 1:num_rbfs

    for j = 1 : size(x_test,2)
        Eucdistance2(j,:)=pdist([x_test(:,j)';centers(i,:)]);
    end

    phi_test(:,i) = rbf(Eucdistance2, sigma);
end
```

```
% Compute the network output for the test inputs using the learned weights
and radial basis functions
Test_pred = phi_test * w;
Test_pred_round=round(Test_pred);

evaluation(Train_ClassLabel,Test_ClassLabel,Trian_pred,Test_pred);
disp("============test");
calacc_classification(Test_ClassLabel,Test_pred_round,size(Test_ClassLabel,
1));
disp("============");

function evaluation(TrLabel,TeLabel,TrPred,TePred)

    TrAcc = zeros(1,1000);
    TeAcc = zeros(1,1000);
    thr = zeros(1,1000);
    TrN = length(TrLabel);
    TeN = length(TeLabel);
    for i = 1:1000
        t = (max(TrPred)-min(TrPred)) * (i-1)/1000 + min(TrPred);
        thr(i) = t;

        TrAcc(i) = (sum(TrLabel(TrPred<t)==0) + sum(TrLabel(TrPred>=t)==1))
/ TrN;
        TeAcc(i) = (sum(TeLabel(TePred<t)==0) + sum(TeLabel(TePred>=t)==1))
/ TeN;
    end
    plot(thr,TrAcc,'.- ',thr,TeAcc,'^-');legend('tr','te');

end

function calacc_classification(Test_ClassLabel,y_test,length)

% Record the successful prediction number
k1=0;
k2=0;

% Identify the successful predictions of input
for i = 1:length
    if y_test(i)==0
        if Test_ClassLabel(i) == y_test(i)
            k1=k1+1;
        end
    end
```

```
    if y_test(i)==1
        if Test_ClassLabel(i) == y_test(i)
            k2=k2+1;
        end
    end
end

% Calculate the accuracy and Display the accuracy
accuracy = (k1+k2)/length;
Z=['Accuracy ',num2str(accuracy)];
X=['there are ',num2str(k1), ' successful predictions of 0 in
',num2str(length),' input cases'];
Y=['there are ',num2str(k2), ' successful predictions of 1 in
',num2str(length),' input cases'];
disp(Z);
disp(X);
disp(Y);
end
```

Attachment 3: Q2 c)

```
clear all
clc

load MNIST_database.mat;
% train_data = training data, 784x1000 matrix
% train_classlabel = the labels of the training data, 1x1000 vector
% test_data = test data, 784x250 matrix
% train_classlabel = the labels of the test data, 1x250 vector


trainIdx = find(train_classlabel==2 | train_classlabel==4); % find the
location of classes 0, 1, 2
Train_ClassLabel = train_classlabel(trainIdx)';
for tmp=1:length(Train_ClassLabel)
    if Train_ClassLabel(tmp)==2
        Train_ClassLabel(tmp)=0
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    else
        Train_ClassLabel(tmp)=1
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
```

```
    end
end
Train_Data = train_data(:,trainIdx);

testIdx = find(test_classlabel==2 | test_classlabel==4); % find the
location of classes 0, 1, 2
Test_ClassLabel = test_classlabel(testIdx)';
for tmp=1:length(Test_ClassLabel)
    if Test_ClassLabel(tmp)==2
        Test_ClassLabel(tmp)=0
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    else
        Test_ClassLabel(tmp)=1
%         tmpimg=reshape(train_data(:,trainIdx(tmp)),28,28);
%         imshow(double(tmpimg));
    end
end
Test_Data = test_data(:,testIdx);


% Define the training inputs and targets
x_train = Train_Data;
y_train = Train_ClassLabel;

% Define the radial basis function with a Gaussian activation function
rbf = @(x, sigma) exp(-x.^2/(2*sigma^2));

% Define the number of radial basis functions to use
num_rbfs = 2;

% Select the centers of the radial basis functions randomly from the
training inputs
[idx,centers] = kmeans(Train_Data',num_rbfs)

% Define the standard deviation of the Gaussian activation function
sigma = 100;

% Compute the radial basis function values for each training input and
center
% phi = zeros(size(x_train,2), num_rbfs);

for i = 1:num_rbfs
```

```
    for j = 1 : size(x_train,2)
        Eucdistance1(j,:)=pdist([x_train(:,j)';centers(i,:)]);
    end

    phi(:,i) = rbf(Eucdistance1, sigma);
end

% Solve for the network weights using the Moore-Penrose pseudoinverse
lambda=0;
w=inv(phi'*phi+lambda*eye(size(phi'*phi,1),size(phi'*phi,2)))*phi'*y_train;

Trian_pred=phi*w;

% Define the test inputs and compute the corresponding radial basis
function values
x_test = Test_Data;

for i = 1:num_rbfs

    for j = 1 : size(x_test,2)
        Eucdistance2(j,:)=pdist([x_test(:,j)';centers(i,:)]);
    end

    phi_test(:,i) = rbf(Eucdistance2, sigma);
end

% Compute the network output for the test inputs using the learned weights
and radial basis functions
Test_pred = phi_test * w;
Test_pred_round=round(Test_pred);

figure;
evaluation(Train_ClassLabel,Test_ClassLabel,Trian_pred,Test_pred);

disp("============test");
calacc_classification(Test_ClassLabel,Test_pred_round,size(Test_ClassLabel,
1));
disp("============");

function evaluation(TrLabel,TeLabel,TrPred,TePred)

    TrAcc = zeros(1,1000);
    TeAcc = zeros(1,1000);
    thr = zeros(1,1000);
```

```
    TrN = length(TrLabel);
    TeN = length(TeLabel);
    for i = 1:1000
        t = (max(TrPred)-min(TrPred)) * (i-1)/1000 + min(TrPred);
        thr(i) = t;

        TrAcc(i) = (sum(TrLabel(TrPred<t)==0) + sum(TrLabel(TrPred>=t)==1))
/ TrN;
        TeAcc(i) = (sum(TeLabel(TePred<t)==0) + sum(TeLabel(TePred>=t)==1))
/ TeN;
    end
    plot(thr,TrAcc,'.- ',thr,TeAcc,'^-');legend('tr','te');

end

function calacc_classification(Test_ClassLabel,y_test,length)

% Record the successful prediction number
k1=0;
k2=0;

% Identify the successful predictions of input
for i = 1:length
    if y_test(i)==0
        if Test_ClassLabel(i) == y_test(i)
            k1=k1+1;
        end
    end
    if y_test(i)==1
        if Test_ClassLabel(i) == y_test(i)
            k2=k2+1;
        end
    end
end

% Calculate the accuracy and Display the accuracy
accuracy = (k1+k2)/length;
Z=['Accuracy ',num2str(accuracy)];
X=['there are ',num2str(k1), ' successful predictions of 0 in
',num2str(length),' input cases'];
Y=['there are ',num2str(k2), ' successful predictions of 1 in
',num2str(length),' input cases'];
disp(Z);
disp(X);
```

```
disp(Y);
end
```

Attachment 3: Q3 a)

```
clear all
clc

% t = linspace(-pi,pi,200);
% trainX = [t.*sin(pi*sin(t)./t); 1-abs(t).*cos(pi*sin(t)./t)]; % 2x200
matrix, column-wise points
% plot(trainX(1,:),trainX(2,:),'+r');

% Set up the input data as a curve
theta = linspace(0, 2*pi, 1000);
x = 16*sin(theta).^3;
y = 13*cos(theta)-5*cos(2*theta)-2*cos(3*theta)-cos(4*theta);


% Set up the SOM parameters
num_neurons = 60;
input_dim = 2;
num_epochs = 800;
learning_rate_initial = 0.1;
sigma_initial=30;


% Initialize the weights
weights = rand(num_neurons, input_dim);

% Train the SOM
for epoch = 1:num_epochs

    % Update the SOM parameters
    learning_rate=learning_rate_initial*exp(-epoch/num_epochs);
    sigma_n = sigma_initial*exp(-epoch/(num_epochs/log(sigma_initial)));

    % Randomly select an input vector
    rand_input_idx=randi(length(x));
    input_vector = [x(rand_input_idx) y(rand_input_idx)];

    % Compute the distances between the input vector and all neurons
    distances = pdist2(input_vector, weights);

    % Find the winning neuron
```

```
    [~, winner] = min(distances);

    % Update the weights of the winning neuron and its neighbors
    for neuron = 1:num_neurons
        distance_to_winner = abs(neuron - winner);

        neighborhood_function = exp(-distance_to_winner^2/(2*sigma_n^2));

        weights(neuron, :) = weights(neuron, :) +
learning_rate*neighborhood_function*(input_vector - weights(neuron, :));

    end
end

% Plot the trained weights
scatter(weights(:, 1), weights(:, 2), 'filled');
hold on;

% Plot lines to connect every topologically adjacent neurons
for i = 1:num_neurons
    if i < num_neurons
        line([weights(i, 1), weights(i+1, 1)], [weights(i, 2), weights(i+1,
2)], 'Color', 'red');
    else
        line([weights(i, 1), weights(1, 1)], [weights(i, 2), weights(1, 2)],
'Color', 'red');
    end
end

% Plot the heart curve
plot(x, y, 'Color', 'blue', 'LineWidth', 2);
axis equal;
title('Self-organizing map for Heart curve');
```

Attachment 3: Q3 b)

```
clear all
clc

% Set up the input data as a curve
trainX = rands(2,500);
x = trainX(1,:);
y = trainX(2,:);
```

```matlab
% Set up the SOM parameters
output_dim = [5 10];
input_dim = 2;
num_epochs = 10000;
learning_rate_initial = 0.6;
sigma_initial=sqrt(output_dim(1)^2+output_dim(2)^2)/2;

% Initialize the weights
weights = rand(output_dim(1)*output_dim(2), input_dim);

% Train the SOM
for epoch = 1:num_epochs

    % Update the SOM parameters
    learning_rate=learning_rate_initial*exp(-epoch/num_epochs);
    sigma_n = sigma_initial*exp(-epoch/(num_epochs/log(sigma_initial)));

    % Randomly select an input vector
    rand_input_idx=randi(length(x));
    input_vector = [x(rand_input_idx) y(rand_input_idx)];

    % Compute the distances between the input vector and all neurons
    distances = pdist2(input_vector, weights);

    % Find the winning neuron
    [~, winner] = min(distances);
    winner_col=mod(winner,output_dim(2));
    winner_row=(winner-winner_col)/output_dim(2);

    % Update the weights of the winning neuron and its neighbors
    for neuron = 1:output_dim(1)*output_dim(2)

        neuron_col= mod(neuron,output_dim(2));
        neuron_row=(neuron-neuron_col)/output_dim(2);

        row_dist = neuron_row-winner_row;
        col_dist = neuron_col-winner_col;
        distance_to_winner = sqrt(row_dist^2 + col_dist^2);

        neighborhood_function = exp(-distance_to_winner^2/(2*sigma_n^2));
```

```
        weights(neuron, :) = weights(neuron, :) +
learning_rate*neighborhood_function*(input_vector - weights(neuron, :));

    end
end


% Plot the trained weights as points in a 2D plane
figure;
scatter(weights(:, 1), weights(:, 2), 'filled');
hold on;

% Plot lines to connect every topologically adjacent neurons
for i = 1:output_dim(1)*output_dim(2)

    [row, col] = ind2sub([output_dim(1) output_dim(2)], i);

        if i-1>=(row-1)*output_dim(2)+1
        left = i-1;
        line([weights(i, 1), weights(left, 1),], [weights(i, 2),
weights(left, 2)], 'Color', 'blue','LineWidth', 5);
        end
        if i+1<=row*output_dim(2)
        right=i+1;
        line([weights(i, 1), weights(right, 1),], [weights(i, 2),
weights(right, 2)], 'Color', 'blue','LineWidth', 5);
        end
        if i-output_dim(2)>=1
        up=i-output_dim(2);
        line([weights(i, 1), weights(up, 1),], [weights(i, 2), weights(up,
2)], 'Color', 'blue','LineWidth', 5);
        end
        if i+output_dim(2)<=output_dim(1)*output_dim(2)
        down=i+output_dim(2);
        line([weights(i, 1), weights(down, 1),], [weights(i, 2),
weights(down, 2)], 'Color', 'blue','LineWidth', 5);
        end

end

% Plot the heart curve
plot(x, y, 'Color', 'red', 'LineWidth', 0.1);
axis equal;
title('Self-organizing map for square');
```

Attachment 3: Q3 c)

```
clear all
clc

load MNIST_database.mat;
% train_data = training data, 784x1000 matrix
% train_classlabel = the labels of the training data, 1x1000 vector
% test_data = test data, 784x250 matrix
% train_classlabel = the labels of the test data, 1x250 vector


trainIdx =
find(train_classlabel==1|train_classlabel==3|train_classlabel==5|train_clas
slabel==6|train_classlabel==7|train_classlabel==8|train_classlabel==9); %
find the location of classes 0, 1, 2
Train_ClassLabel = train_classlabel(trainIdx)';
Train_Data = train_data(:,trainIdx);


testIndx = find(test_classlabel==1 | test_classlabel==3|
test_classlabel==5| test_classlabel==6| test_classlabel==7|
test_classlabel==8| test_classlabel==9); % find the location of classes 0,
1, 2
Test_ClassLabel = test_classlabel(testIndx)';
Test_Data = test_data(:,testIndx);


% Set up the SOM parameters
nenum=10;
output_dim = [nenum nenum];
num_neurons = output_dim(1)*output_dim(1);
num_epochs = 20;
learning_rate_initial = 1;
sigma_initial=sqrt(output_dim(1)^2+output_dim(2)^2)/2;
num_images=size(Train_Data,2);

% Initialize the weights
weights = rand(size(Train_Data,1), num_neurons);


% Step 4: Train the SOM network using the image data
for epoch = 1:num_epochs
```

```
    % Shuffle the data for each epoch
    shuffled_data = Train_Data(:, randperm(num_images));

    % Update the SOM parameters
    lr=learning_rate_initial*exp(-epoch/num_epochs);
    sigma_n = sigma_initial*exp(-epoch/(num_epochs/log(sigma_initial)));

    % Train the SOM network on the shuffled data
    for i = 1:num_images
        x = shuffled_data(:, i);
        [~, bmu] = min(sum((weights - x).^2)); % Find the best matching unit
(BMU)
        bmu_row = mod(bmu-1, output_dim(2)) + 1;
        bmu_col = ceil(bmu/output_dim(2));
        for j = 1:num_neurons
            dist = sqrt((bmu_row - mod(j-1, 10) - 1)^2 + (bmu_col -
ceil(j/10))^2); % Calculate the distance between the BMU and the current
neuron
            neighbor_function = exp(-dist^2/(2*sigma_n^2));
            % Update the weights of the current neuron
            weights(:, j) = weights(:, j) + lr *neighbor_function* (x -
weights(:, j));

        end
    end
end

% Step 5: Generate a semantic map for each neuron
figure;
for i = 1:num_neurons
    % Find the input patterns that activate the current neuron the most

    [~, index] = min(sum((weights(:,i) - Train_Data).^2)); % Find the best
matching image and return the index

    weights_label(i)=Train_ClassLabel(index);

    % Visualize the top input patterns
    subplot(10, 10, i);
    imshow(reshape(Train_Data(:,index), 28, 28), []);
    sgtitle(sprintf('Neuron %d', i));
end

hold off;
```

```
for i=1:size(Train_Data,2)
    [~, bmu] = min(sum((weights - Train_Data(:,i)).^2)); % Find the best
matching unit (BMU)
    Trian_pred(i)=weights_label(bmu);
end


for i=1:size(Test_Data,2)
    [~, bmu] = min(sum((weights - Test_Data(:,i)).^2)); % Find the best
matching unit (BMU)
    Test_pred(i)=weights_label(bmu);
end

accuracy(Test_pred,Trian_pred,Test_ClassLabel,Train_ClassLabel);

function accuracy(Test_pred,Trian_pred,Test_ClassLabel,Train_ClassLabel)
    test_record_array=zeros(1,9);
    for i=1:length(Test_pred)
        if Test_pred(i)==Test_ClassLabel(i)
            test_record_array(Test_pred(i))=
test_record_array(Test_pred(i))+1;
        end
    end

    train_record_array=zeros(1,9);
    for i=1:length(Test_pred)
        if Trian_pred(i)==Train_ClassLabel(i)
            train_record_array(Test_pred(i))=
train_record_array(Test_pred(i))+1;
        end
    end

% Calculate the accuracy and Display the accuracy
disp("============test");
accuracy = sum(test_record_array)/length(Test_pred);
Z=['Accuracy ',num2str(accuracy)];
disp(Z);
 for i=1:length(test_record_array)
    X=['there are ',num2str(test_record_array(i)), ' successful predictions
of ' ,num2str(i), ' in ',num2str(length(Test_pred)),' input cases'];
    disp(X);
 end
```

```
 disp("============train");
 accuracy = sum(train_record_array)/length(Trian_pred);
 Z=['Accuracy ',num2str(accuracy)];
disp(Z);
 for i=1:length(train_record_array)
    X=['there are ',num2str(train_record_array(i)), ' successful
predictions of ' ,num2str(i), ' in ',num2str(length(Trian_pred)),' input
cases'];
    disp(X);
 end
end
```