Q1 Function approximation with RBFN

1. Exact interpolation method.

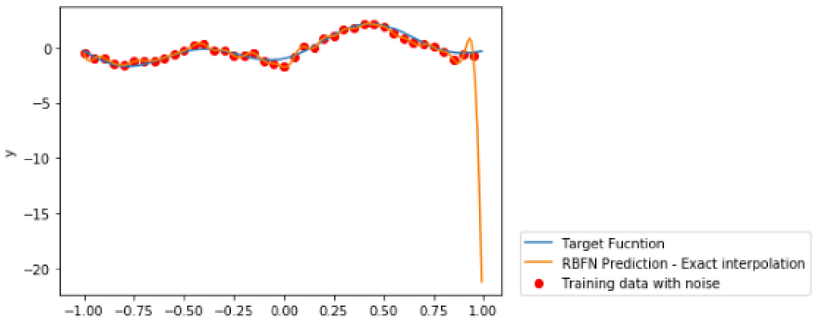
```
# RBF is a Gaussian function. Then we need to calculate phi using exact interpolation to get interpolation matrix
phi = np.zeros((len(train_x), len(train_x)))
std = 0.01
for i in range(phi.shape[0]):
    for j in range(phi.shape[1]):
        r = train_x[i] - train_x[j]
        phi[i,j] = np.exp(-r**2/(2*std))
```

```
# solve the linear equation to get unique solution of w
w = np.linalg.inv(phi).dot(train_y)
```

```
# Feed test data
phi_test = np.zeros((len(test_x), len(train_x)))
for i in range(phi_test.shape[0]):
    for j in range(phi_test.shape[1]):
        r = test_x[i] - train_x[j]
        phi_test[i,j] = np.exp(-r**2/(2*std))

# Predict
predict_y = phi_test.dot(w)

# mean of sum of squares Error
train_error = sum((train_y - (phi.dot(w)))**2)/(len(train_x))
test_error = sum((test_y - predict_y)**2)/(len(test_x))
print("Train error of exact interpolation is %r" %train_error)
print("Test error of exact interpolation is %r" %test_error)
```

```
Train error of exact interpolation is 3.3428338306829254e-19
Test error of exact interpolation is 3.4468849747928116
```

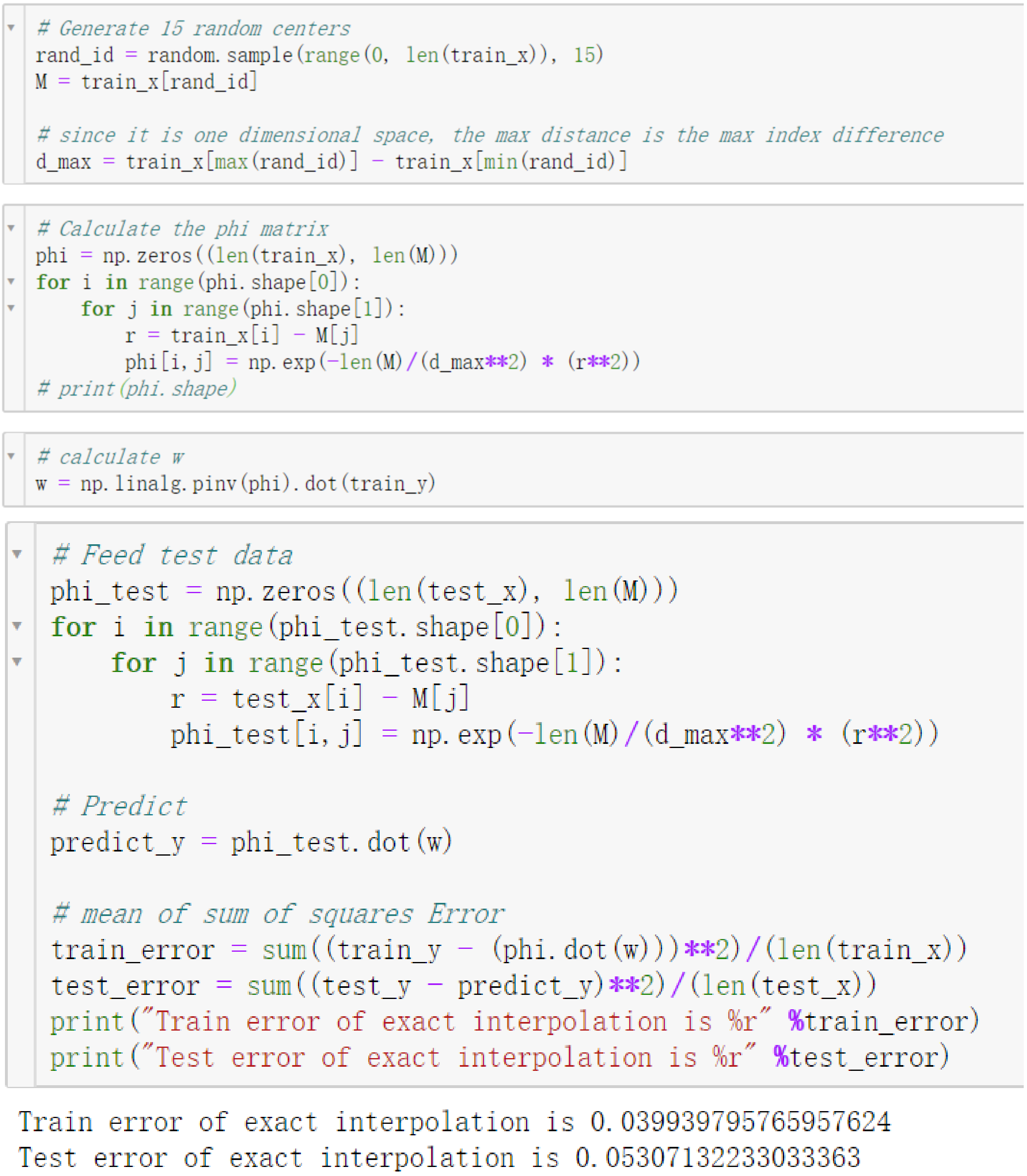
```
# Plot the approximation figure
plt.plot(test_x, test_y, label ="Target Fucntion" )
plt.plot(test_x, predict_y, label ="RBFN Prediction - Exact interpolation")
plt.scatter(train_x, train_y, label ="Training data with noise", color = "r")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc = (1.04,0))
plt.show()
```



We can found the MSE of train set is 3.34*4-19 and MSE of test is 3.44. Obviously, it is overfitting in this situation.

(b) Fixed centers selected at Random

15 centers are selected.



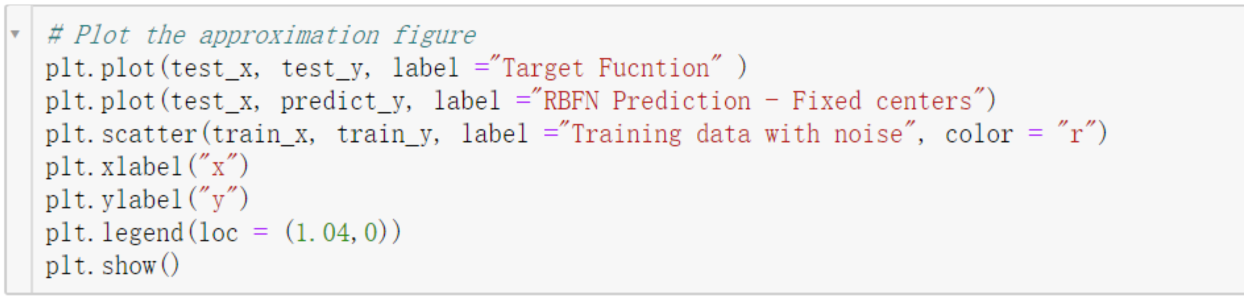
```python
# Generate 15 random centers
rand_id = random.sample(range(0, len(train_x)), 15)
M = train_x[rand_id]

# since it is one dimensional space, the max distance is the max index difference
d_max = train_x[max(rand_id)] - train_x[min(rand_id)]
```

```python
# Calculate the phi matrix
phi = np.zeros((len(train_x), len(M)))
for i in range(phi.shape[0]):
    for j in range(phi.shape[1]):
        r = train_x[i] - M[j]
        phi[i, j] = np.exp(-len(M)/(d_max**2) * (r**2))
# print(phi.shape)
```

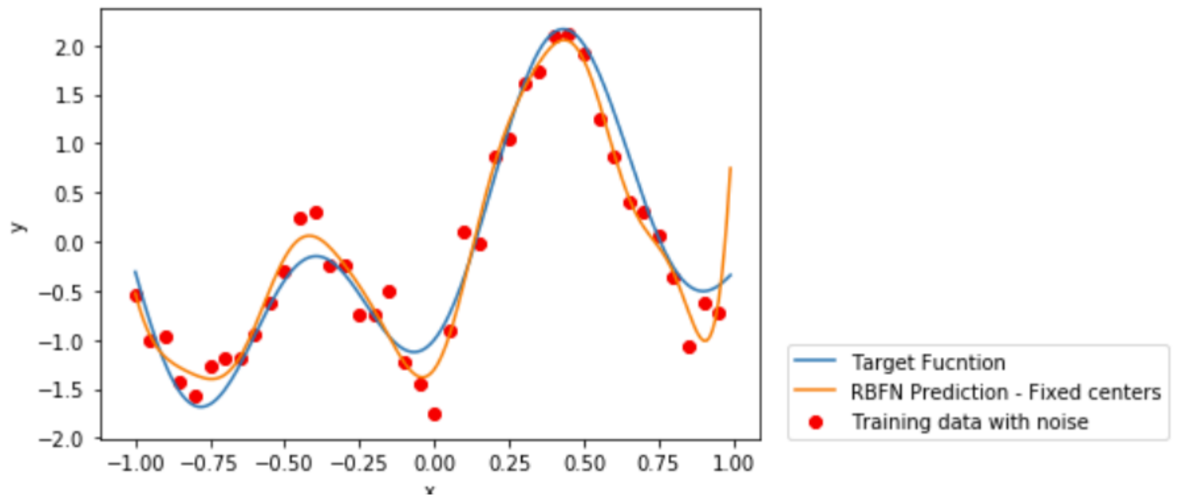
```python
# calculate w
w = np.linalg.pinv(phi).dot(train_y)
```

```python
# Feed test data
phi_test = np.zeros((len(test_x), len(M)))
for i in range(phi_test.shape[0]):
    for j in range(phi_test.shape[1]):
        r = test_x[i] - M[j]
        phi_test[i, j] = np.exp(-len(M)/(d_max**2) * (r**2))

# Predict
predict_y = phi_test.dot(w)

# mean of sum of squares Error
train_error = sum((train_y - (phi.dot(w)))**2)/(len(train_x))
test_error = sum((test_y - predict_y)**2)/(len(test_x))
print("Train error of exact interpolation is %r" %train_error)
print("Test error of exact interpolation is %r" %test_error)
```

```
Train error of exact interpolation is 0.03993979576595 7624
Test error of exact interpolation is 0.05307132233033363
```



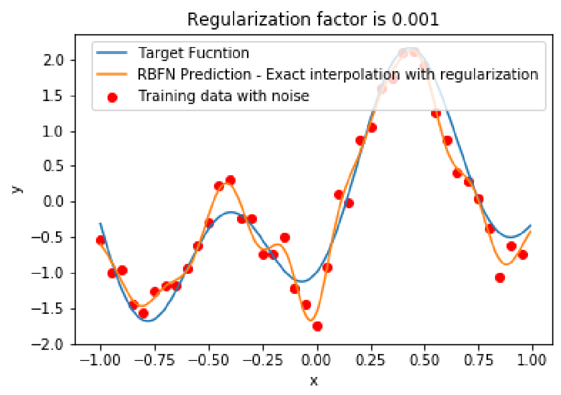
```
# Plot the approximation figure
plt.plot(test_x, test_y, label ="Target Fucntion" )
plt.plot(test_x, predict_y, label ="RBFN Prediction - Fixed centers")
plt.scatter(train_x, train_y, label ="Training data with noise", color = "r")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc = (1.04,0))
plt.show()
```

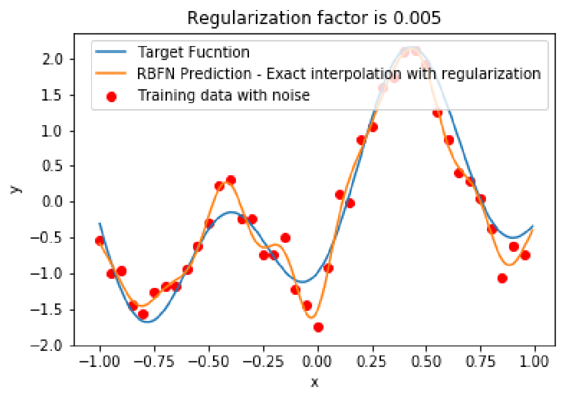


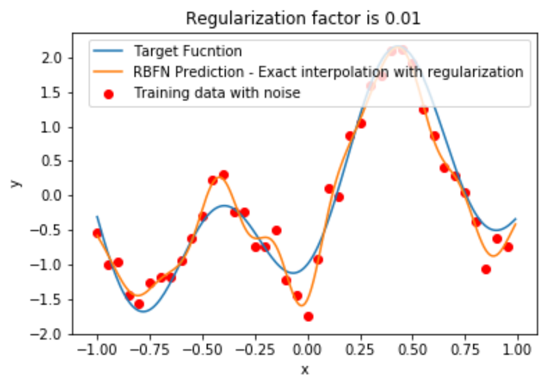
MSE for training dataset and test dataset is 0.0399 and 0.0531 respectively. The curve is much smoother than (a). The prediction curve is fluctuant around the target function. The testing error decrease a lot compared with (a).

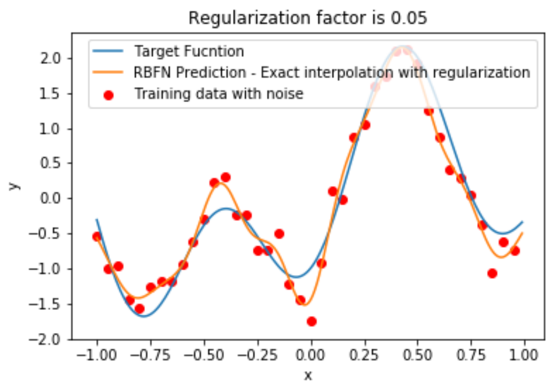
(3) Same centers and width as in part (a) and apply regularization method.

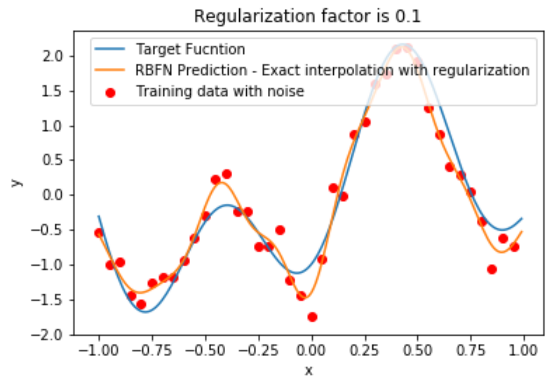
In this experiment, multiple lambda factors are used to figure out the effects on the performance of the RBFN.

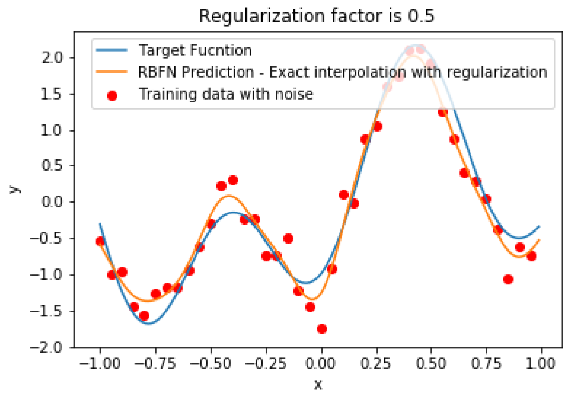


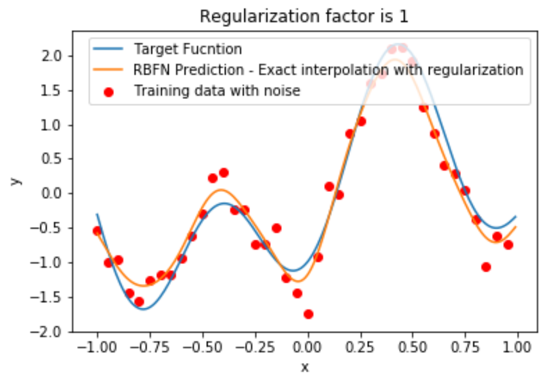


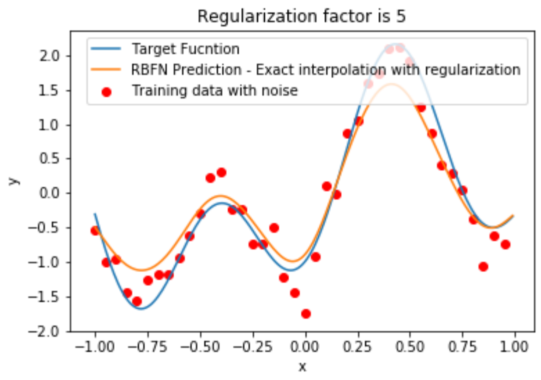

Regularization factor is 0.01
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y

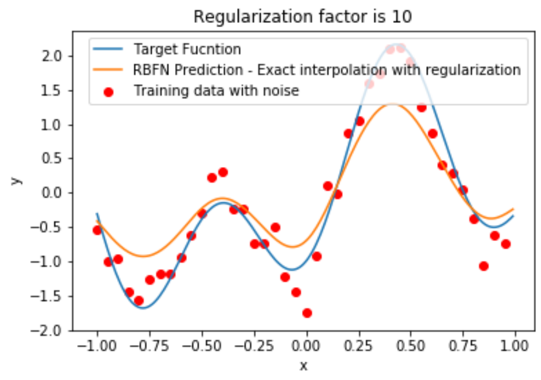

Regularization factor is 0.05
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y

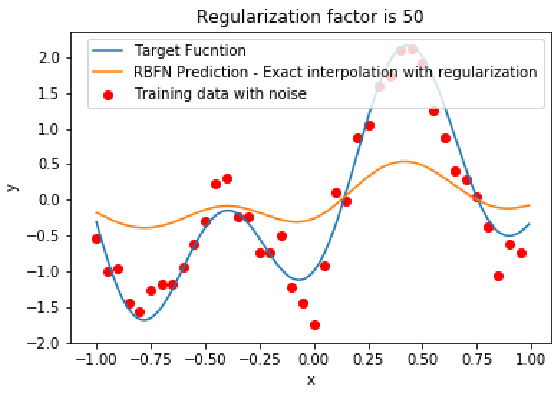

Regularization factor is 0.1
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y

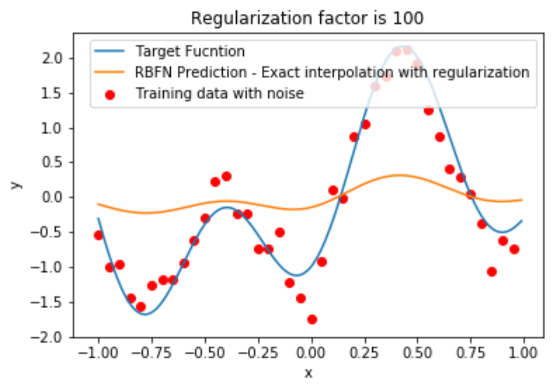

Regularization factor is 0.5
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y

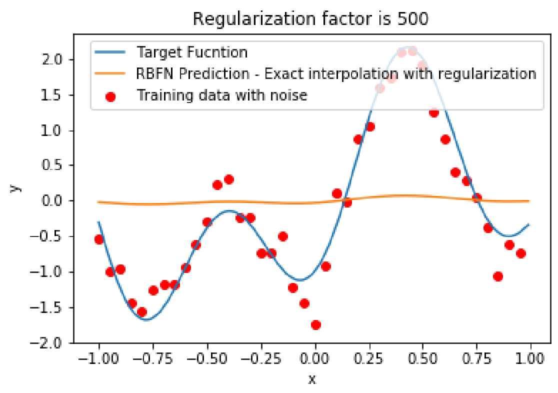

Regularization factor is 1
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y


Regularization factor is 5
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y


Regularization factor is 10
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y


Regularization factor is 50
Target Fucntion
RBFN Prediction - Exact interpolation with regularization
Training data with noise
x
y





| Regularization factor | Train error | Test error |
|---|---|---|
| 0.001 | 0.017890191333931646 | 0.06324670792822584 |
| 0.005 | 0.01940035882435805 | 0.060302591767714325 |
| 0.01 | 0.02028647023895462 | 0.05845593293992961 |
| 0.05 | 0.02341272038090462 | 0.05192064935338275 |
| 0.1 | 0.025648837188060408 | 0.0480631275407296 |
| 0.5 | 0.035192679119723985 | 0.040725120569205735 |
| 1 | 0.04214354857078361 | 0.04166250613002489 |
| 5 | 0.10265260109598322 | 0.09948269409469494 |
| 10 | 0.19602435330556786 | 0.1998828391850863 |
| 50 | 0.6506930336681187 | 0.690675388555083 |
| 100 | 0.8456289262679684 | 0.9005749947362345 |
| 500 | 1.0836933861251015 | 1.156494260489846 |

We can find that the curve would be more smooth as regularization factor increase. However, when factors are too large (e.g 50, 100, 500), it becomes flat and less fitting. In this situation, training and testing error increase a lot.

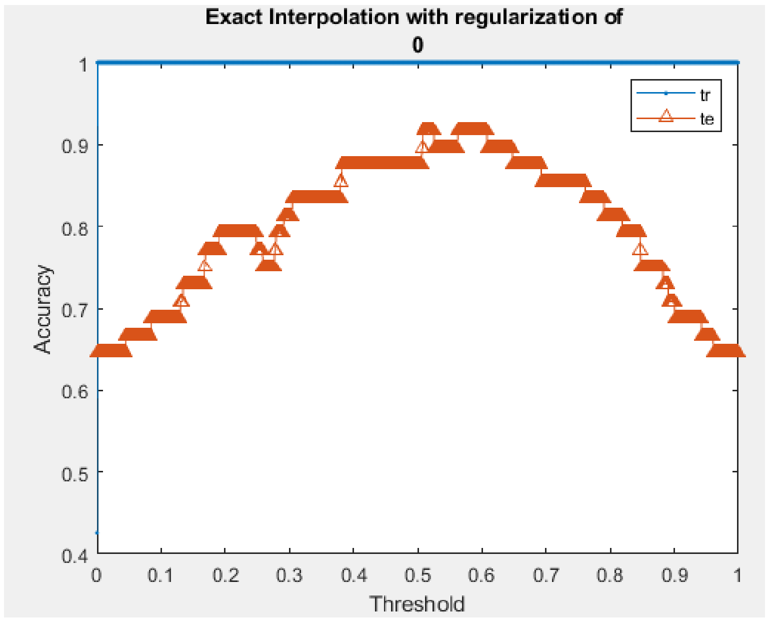
```
lambda_factor = 500
phi = np.zeros((len(train_x), len(train_x)))
std = 0.01
for i in range(phi.shape[0]):
    for j in range(phi.shape[1]):
        r = train_x[i] - train_x[j]
        phi[i, j] = np.exp(-r**2/(2*std))

# solve the linear equation to get unique solution of w
tmp =  np.linalg.inv((np.transpose(phi).dot(phi) + lambda_factor*np.identity(phi.shape[0])))
tmp2 = np.transpose(phi).dot(train_y)
w = np.dot(tmp, tmp2)

# Feed test data
phi_test = np.zeros((len(test_x), len(train_x)))
for i in range(phi_test.shape[0]):
    for j in range(phi_test.shape[1]):
        r = test_x[i] - train_x[j]
        phi_test[i, j] = np.exp(-r**2/(2*std))

# Predict
predict_y = phi_test.dot(w)
```

Q2 Handwritten digits classification using RBFN

My matriculation number is A0206944B. My class are 9 and 4

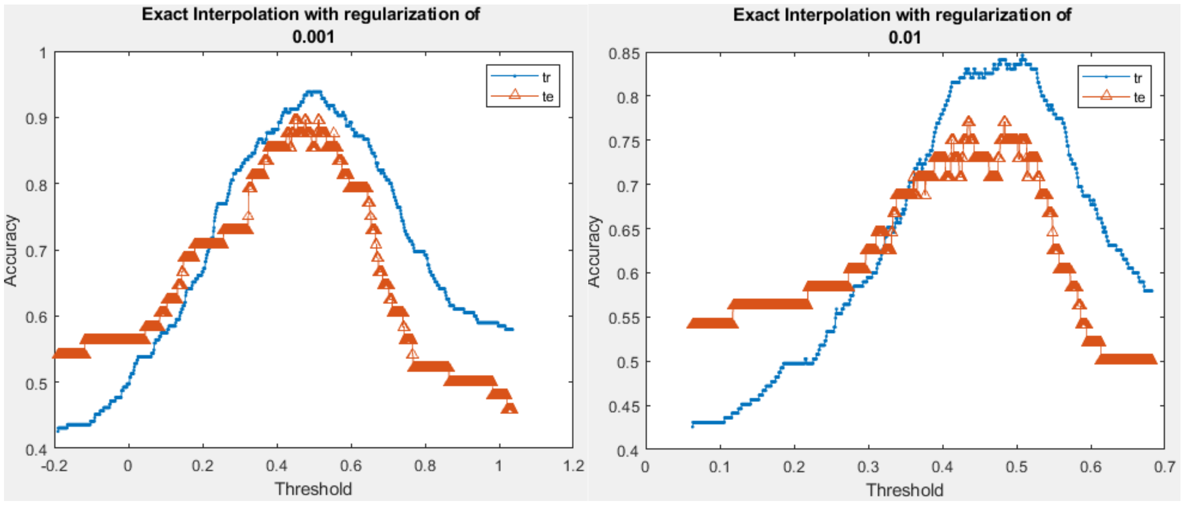
1. Exact interpolation method. RBF is Gaussian function with std of 100.

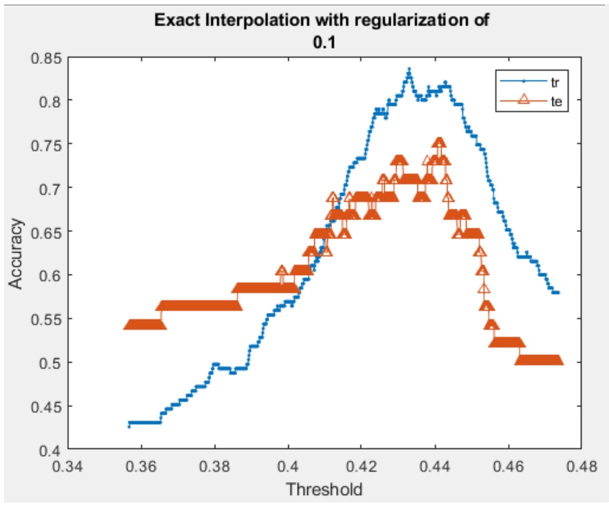
Regulation 0:

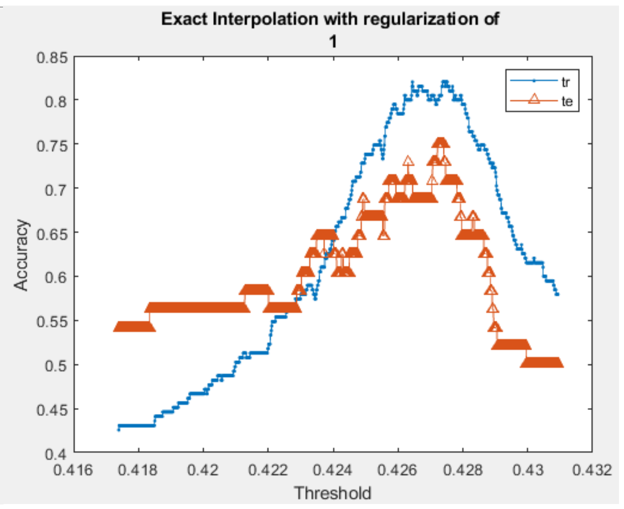


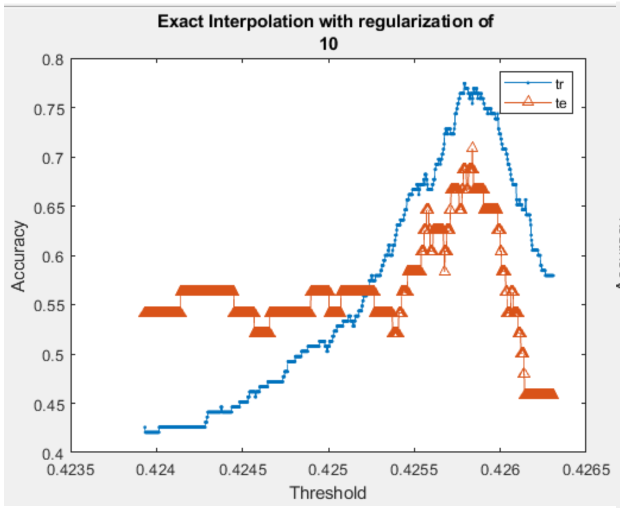
If the threshold is bigger than 0, the training accuracy would be 100%. It would pass through all training samples exactly.

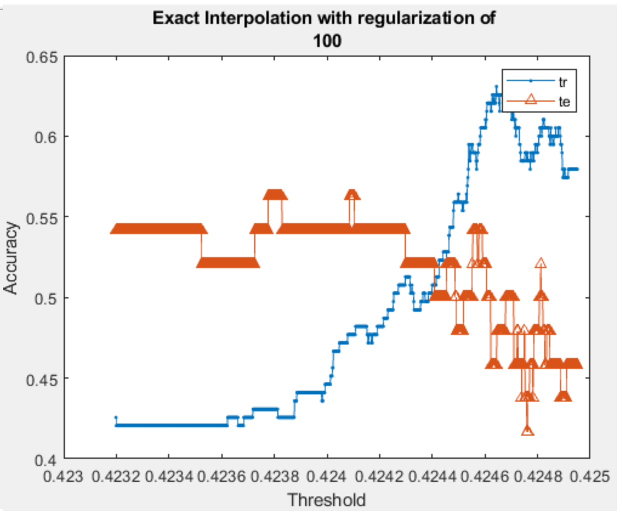
After applying the regularization,







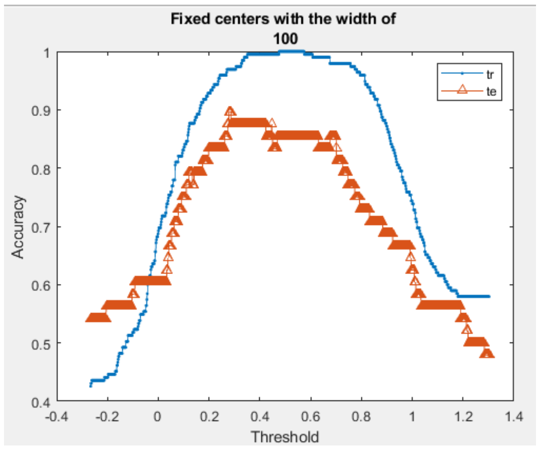




We can find that the training accuracy cannot reach 100% since the results would be more smooth than the original. When the regularization factor increase to 100, the testing accuracy drops compared with other situation. It might be resulted by the under fitting.

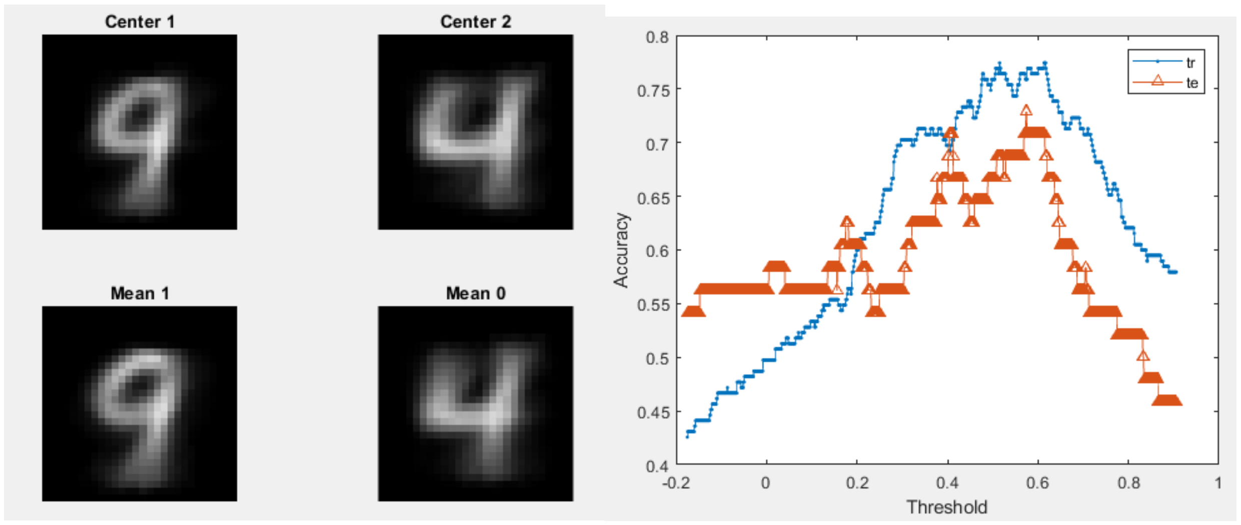
(2) Fixed centers selected at Random.

Here we select 100 centers among the training sample. Widths are firstly fixed as (1).



Width from 0.1 to 1000, the performance of figure

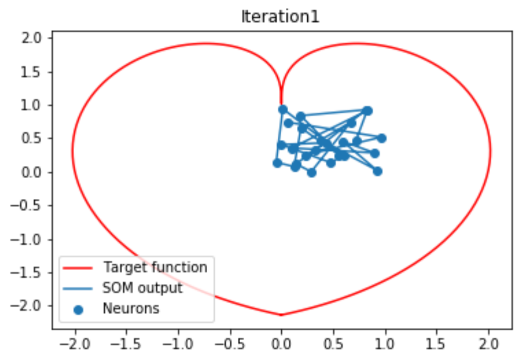
(3) K-means clustering with two centers

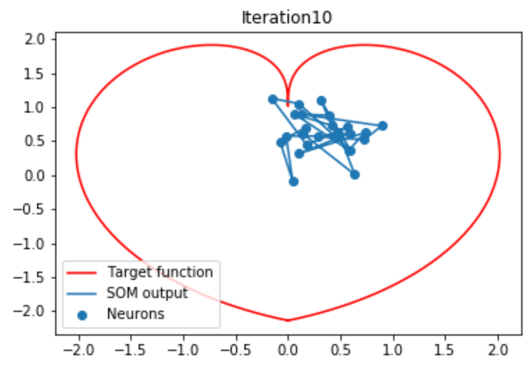


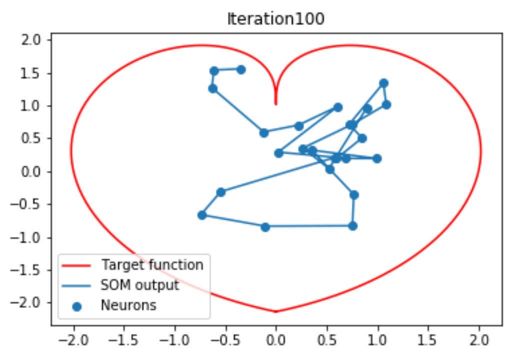
The performance wasn't better than (2). It might be the number of training and testing is limited. Thus, the performance is not enhanced a lot. Compare with (2), we can also find that the two center images are very close to two mean of training images, indicating the class 9 and 4 respectively. It's clear to find out the underlying pattern.

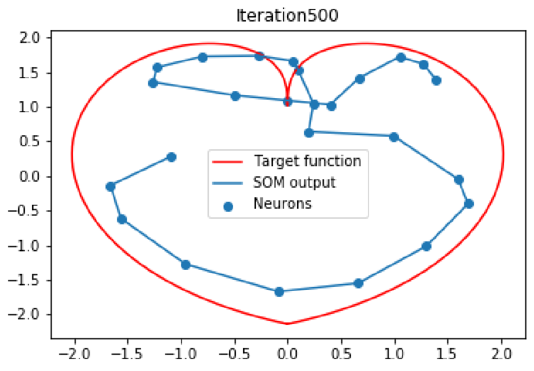
## Q3 Self-Organizing Map

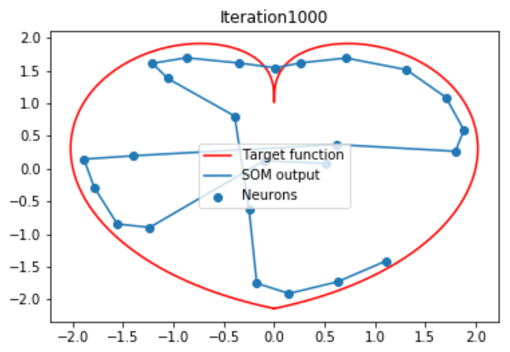
1. SOM with output layer of 25 neurons. Various iterations were tested in this experiment.

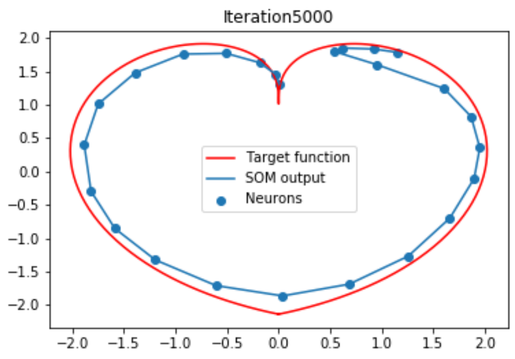


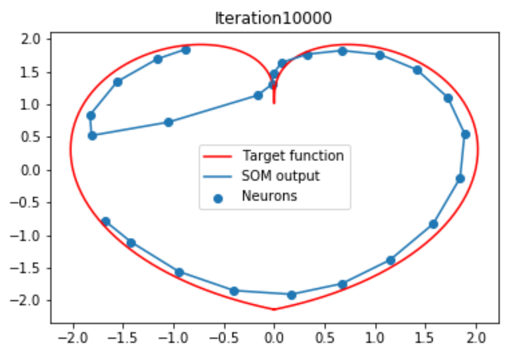


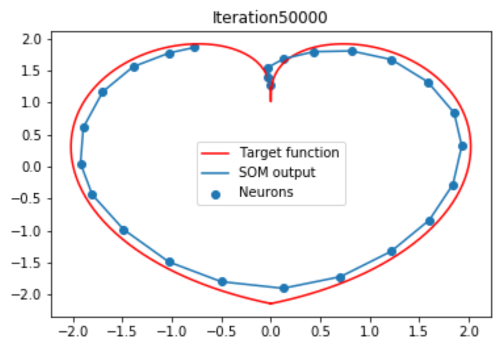




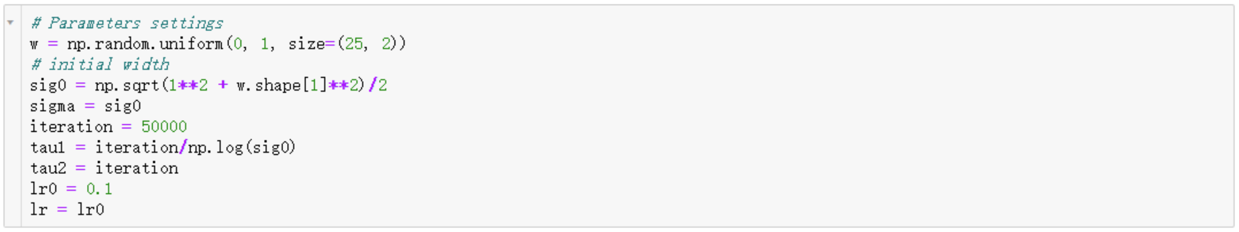




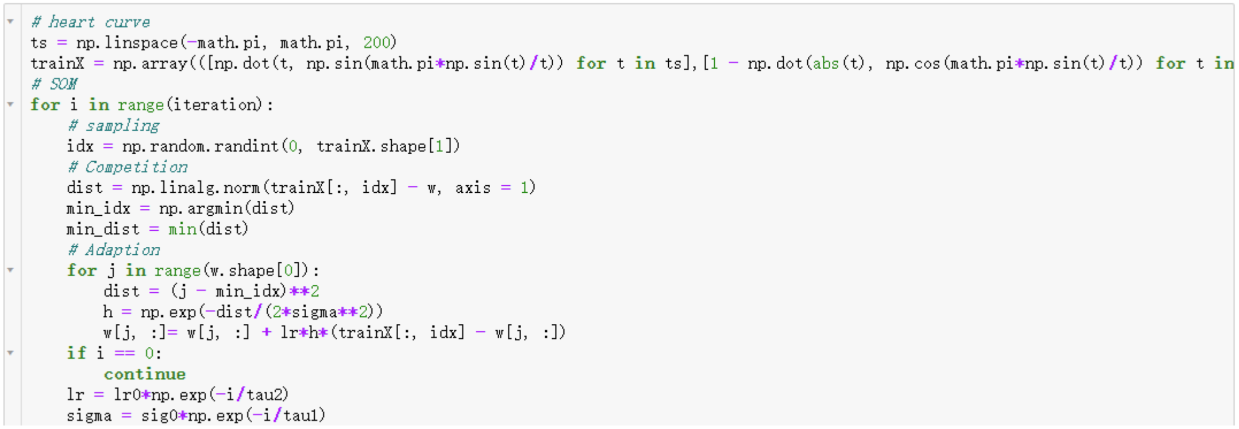




We can find that the fitting is better when running more iterations.

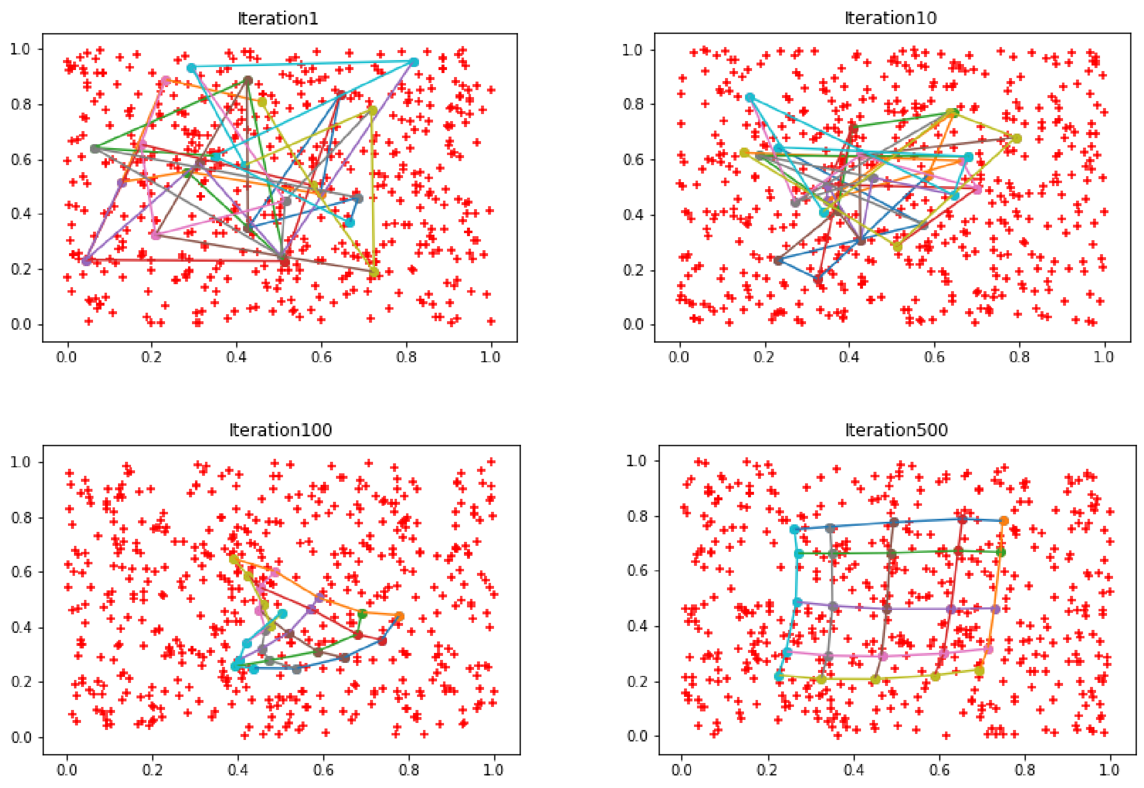


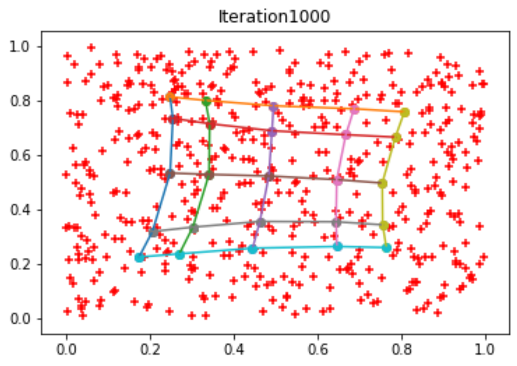
```
# Parameters settings
w = np.random.uniform(0, 1, size=(25, 2))
# initial width
sig0 = np.sqrt(1**2 + w.shape[1]**2)/2
sigma = sig0
iteration = 50000
tau1 = iteration/np.log(sig0)
tau2 = iteration
lr0 = 0.1
lr = lr0
```

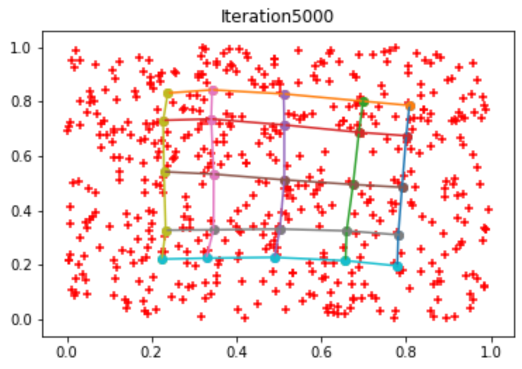


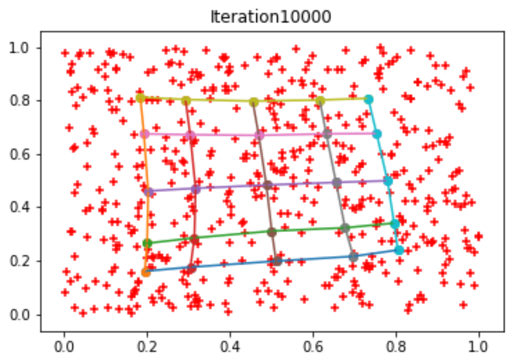
```
# heart curve
ts = np.linspace(-math.pi, math.pi, 200)
trainX = np.array(([np.dot(t, np.sin(math.pi*np.sin(t)/t)) for t in ts],[1 - np.dot(abs(t), np.cos(math.pi*np.sin(t)/t)) for t in
# SOM
for i in range(iteration):
    # sampling
    idx = np.random.randint(0, trainX.shape[1])
    # Competition
    dist = np.linalg.norm(trainX[:, idx] - w, axis = 1)
    min_idx = np.argmin(dist)
    min_dist = min(dist)
    # Adaption
    for j in range(w.shape[0]):
        dist = (j - min_idx)**2
        h = np.exp(-dist/(2*sigma**2))
        w[j, :]= w[j, :] + lr*h*(trainX[:, idx] - w[j, :])
    if i == 0:
        continue
    lr = lr0*np.exp(-i/tau2)
    sigma = sig0*np.exp(-i/tau1)
```

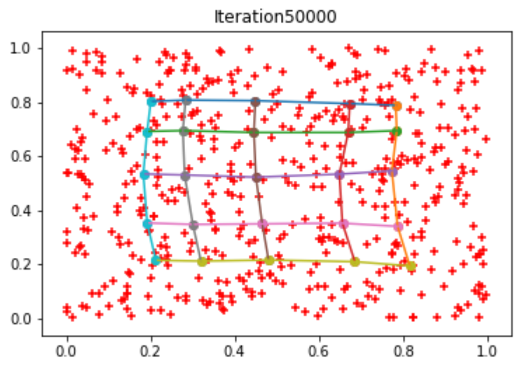
(2) 2-D output later of 25 neurons to a "square"











With more iteration, the fitting is more likely to be a square shape.

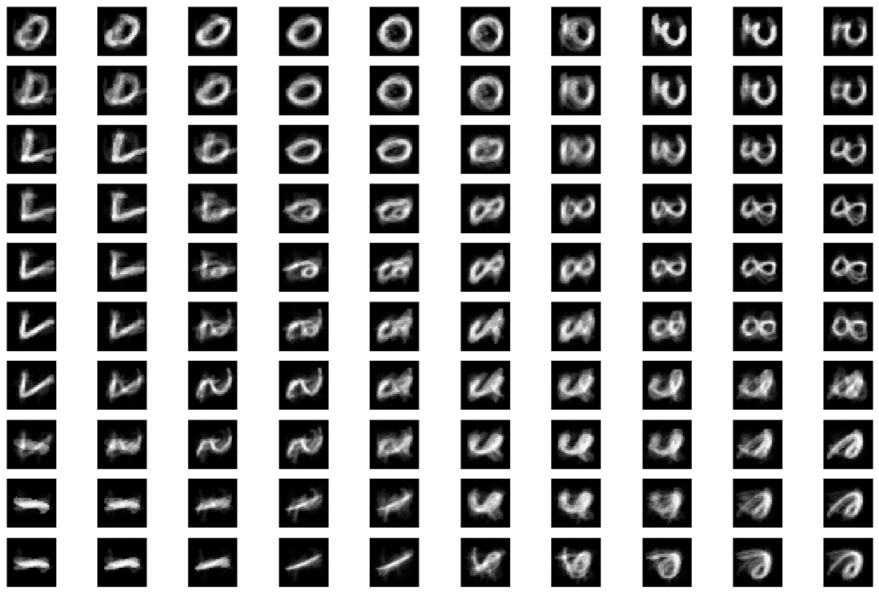
```
# Sqaure
trainX = np.random.uniform(0,1, size=(2, 500))
# SOM
for i in range(iteration):
    # sampling
    idx = np.random.randint(0, trainX.shape[1])
    # Competition
    dist = np.zeros((5,5))
    for r in range(5):
        for c in range(5):
            dist[r,c] = np.sqrt((trainX[0, idx] - w[0, r,c])**2 +(trainX[1, idx] - w[1, r,c])**2)
    min_r_idx, min_c_idx = np.where(dist == np.amin(dist))
    # Adaption
    for r in range(5):
        for c in range(5):
            dist = (r - min_r_idx)**2 + (c - min_c_idx)**2
            h = np.exp(-dist/(2*sigma**2))
            w[:, r, c]= w[:, r, c] + lr*h*(trainX[:, idx] - w[:, r, c])
    if i ==0:
        continue
    lr = lr0*np.exp(-i/tau2)
    sigma = sig0*np.exp(-i/tau1)
```

(3) MNIST_database.mat

My matriculation number is A0206944B. I would omit class 9 and 4.

a. The procedure of generating the semantic map is 1) calculate weights, 2) select one neuron, 3) search all training set and calculate the distance with this neuron, 4) find the minimal distance and return the label of this training data as neuron label, 5) repeat for all neurons.

```
[[0.  3.  0.  0.  0.  0.  3.  3.  3.  5.]
 [7.  7.  0.  0.  0.  0.  3.  3.  3.  5.]
 [7.  7.  7.  0.  0.  0.  5.  3.  3.  3.]
 [7.  7.  7.  6.  0.  8.  3.  3.  8.  8.]
 [7.  7.  7.  6.  6.  8.  8.  3.  8.  8.]
 [7.  7.  7.  6.  8.  2.  8.  8.  8.  8.]
 [7.  7.  5.  5.  8.  2.  2.  2.  2.  6.]
 [1.  5.  5.  5.  2.  2.  2.  2.  2.  6.]
 [1.  1.  1.  1.  1.  7.  2.  2.  6.  6.]
 [1.  1.  1.  1.  1.  7.  2.  6.  6.  6.]]
```



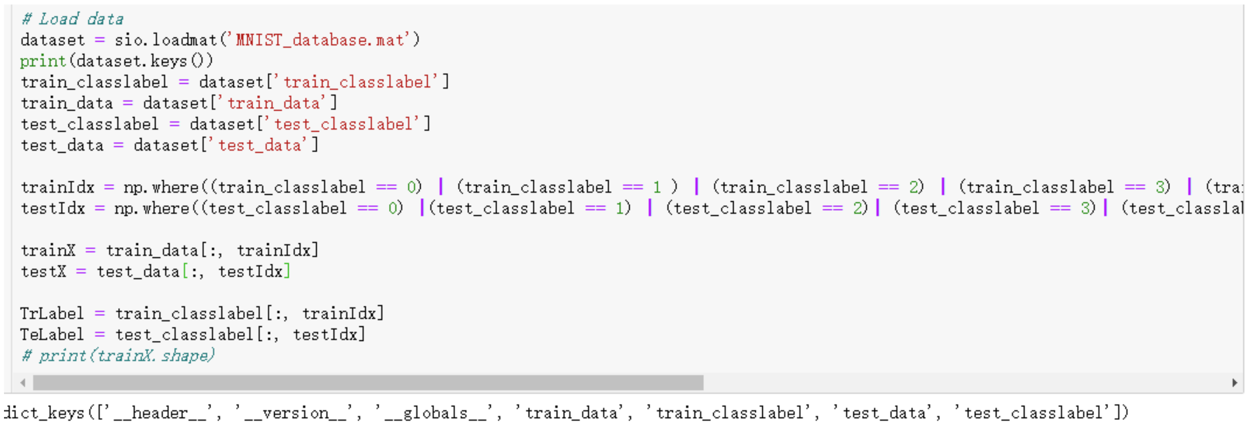
We can find that same label images are stored closely. They are compactly clustered. The label and weights information are connected correspondingly.

b. Test the SOM.

Here we used 10000 iteration and initial learning rate is 1.

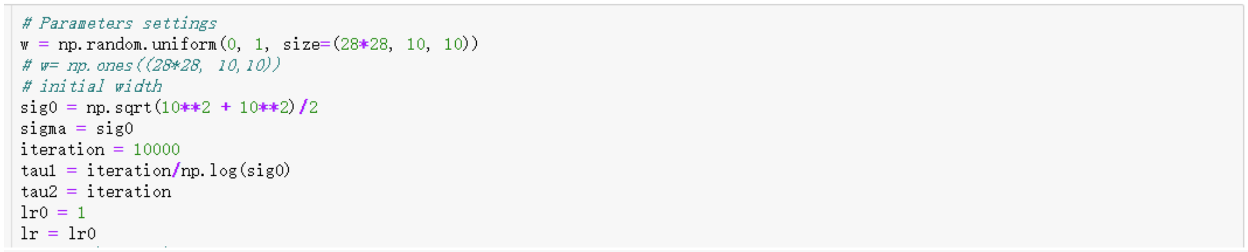
Accuracy on training dataset is 0.7726708074534161.

Accuracy on testing dataset is 0.7623762376237624.

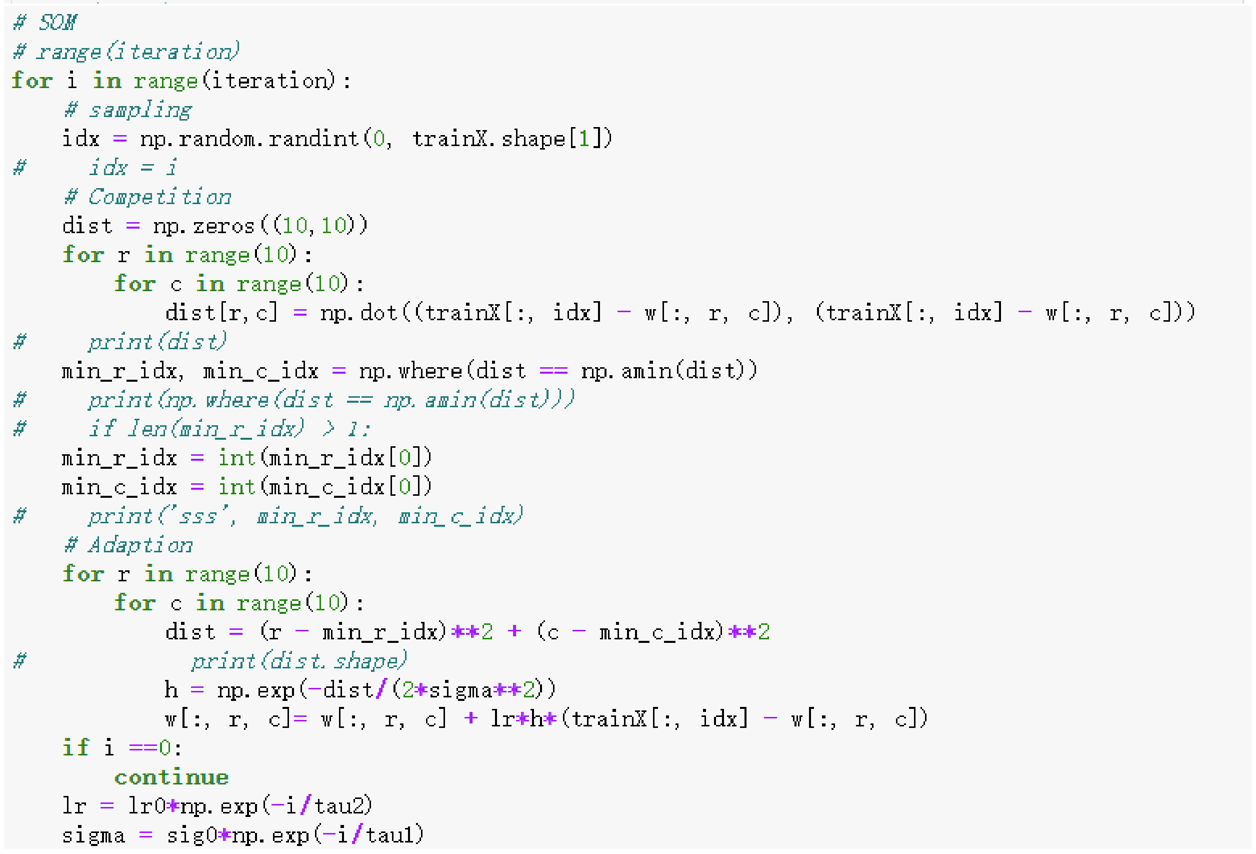


```
# Load data
dataset = sio.loadmat('MNIST_database.mat')
print(dataset.keys())
train_classlabel = dataset['train_classlabel']
train_data = dataset['train_data']
test_classlabel = dataset['test_classlabel']
test_data = dataset['test_data']

trainIdx = np.where((train_classlabel == 0) | (train_classlabel == 1 ) | (train_classlabel == 2) | (train_classlabel == 3) | (tra
testIdx = np.where((test_classlabel == 0) |(test_classlabel == 1) | (test_classlabel == 2)| (test_classlabel == 3)| (test_classla

trainX = train_data[:, trainIdx]
testX = test_data[:, testIdx]

TrLabel = train_classlabel[:, trainIdx]
TeLabel = test_classlabel[:, testIdx]
# print(trainX.shape)
```

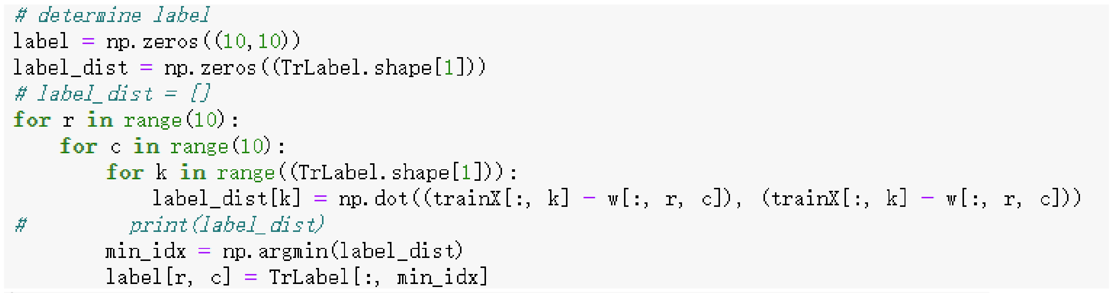
```
dict_keys(['__header__', '__version__', '__globals__', 'train_data', 'train_classlabel', 'test_data', 'test_classlabel'])
```



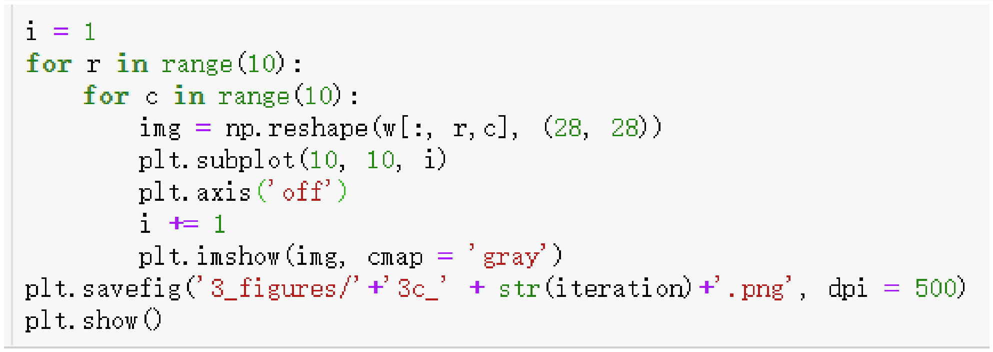
```
# Parameters settings
w = np.random.uniform(0, 1, size=(28*28, 10, 10))
# w= np.ones((28*28, 10,10))
# initial width
sig0 = np.sqrt(10**2 + 10**2)/2
sigma = sig0
iteration = 10000
tau1 = iteration/np.log(sig0)
tau2 = iteration
lr0 = 1
lr = lr0
```



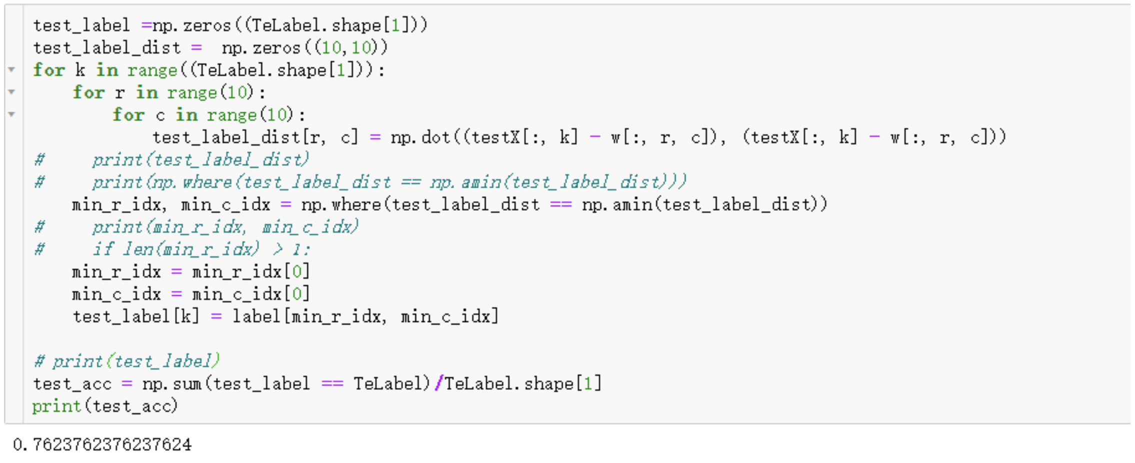
```
# SOM
# range(iteration)
for i in range(iteration):
    # sampling
    idx = np.random.randint(0, trainX.shape[1])
#     idx = i
    # Competition
    dist = np.zeros((10,10))
    for r in range(10):
        for c in range(10):
            dist[r,c] = np.dot((trainX[:, idx] - w[:, r, c]), (trainX[:, idx] - w[:, r, c]))
#     print(dist)
    min_r_idx, min_c_idx = np.where(dist == np.amin(dist))
#     print(np.where(dist == np.amin(dist)))
#     if len(min_r_idx) > 1:
    min_r_idx = int(min_r_idx[0])
    min_c_idx = int(min_c_idx[0])
#     print('sss', min_r_idx, min_c_idx)
    # Adaption
    for r in range(10):
        for c in range(10):
            dist = (r - min_r_idx)**2 + (c - min_c_idx)**2
#             print(dist.shape)
            h = np.exp(-dist/(2*sigma**2))
            w[:, r, c]= w[:, r, c] + lr*h*(trainX[:, idx] - w[:, r, c])
    if i ==0:
        continue
    lr = lr0*np.exp(-i/tau2)
    sigma = sig0*np.exp(-i/tau1)
```



```
# determine label
label = np.zeros((10,10))
label_dist = np.zeros((TrLabel.shape[1]))
# label_dist = []
for r in range(10):
    for c in range(10):
        for k in range((TrLabel.shape[1])):
            label_dist[k] = np.dot((trainX[:, k] - w[:, r, c]), (trainX[:, k] - w[:, r, c]))
#          print(label_dist)
        min_idx = np.argmin(label_dist)
        label[r, c] = TrLabel[:, min_idx]
```

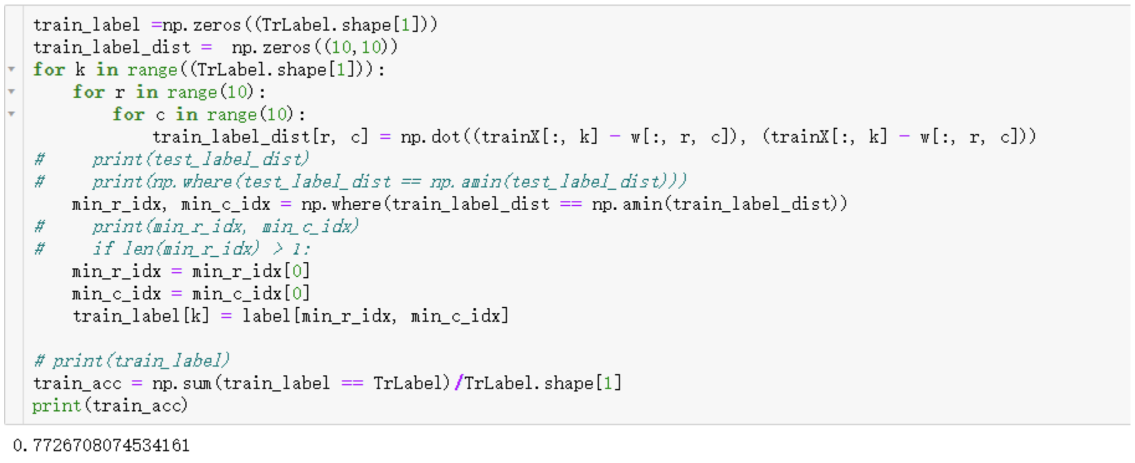


```
i = 1
for r in range(10):
    for c in range(10):
        img = np.reshape(w[:, r,c], (28, 28))
        plt.subplot(10, 10, i)
        plt.axis('off')
        i += 1
        plt.imshow(img, cmap = 'gray')
plt.savefig('3_figures/'+'3c_' + str(iteration)+'.png', dpi = 500)
plt.show()
```



```
test_label =np.zeros((TeLabel.shape[1]))
test_label_dist =  np.zeros((10,10))
for k in range((TeLabel.shape[1])):
    for r in range(10):
        for c in range(10):
            test_label_dist[r, c] = np.dot((testX[:, k] - w[:, r, c]), (testX[:, k] - w[:, r, c]))
#     print(test_label_dist)
#     print(np.where(test_label_dist == np.amin(test_label_dist)))
    min_r_idx, min_c_idx = np.where(test_label_dist == np.amin(test_label_dist))
#     print(min_r_idx, min_c_idx)
#     if len(min_r_idx) > 1:
    min_r_idx = min_r_idx[0]
    min_c_idx = min_c_idx[0]
    test_label[k] = label[min_r_idx, min_c_idx]

# print(test_label)
test_acc = np.sum(test_label == TeLabel)/TeLabel.shape[1]
print(test_acc)
```

```
0.7623762376237624
```



```
train_label =np.zeros((TrLabel.shape[1]))
train_label_dist =  np.zeros((10,10))
for k in range((TrLabel.shape[1])):
    for r in range(10):
        for c in range(10):
            train_label_dist[r, c] = np.dot((trainX[:, k] - w[:, r, c]), (trainX[:, k] - w[:, r, c]))
#     print(test_label_dist)
#     print(np.where(test_label_dist == np.amin(test_label_dist)))
    min_r_idx, min_c_idx = np.where(train_label_dist == np.amin(train_label_dist))
#     print(min_r_idx, min_c_idx)
#     if len(min_r_idx) > 1:
    min_r_idx = min_r_idx[0]
    min_c_idx = min_c_idx[0]
    train_label[k] = label[min_r_idx, min_c_idx]

# print(train_label)
train_acc = np.sum(train_label == TrLabel)/TrLabel.shape[1]
print(train_acc)
```

```
0.7726708074534161
```

After the experiment, I found that by increasing the running epochs and tuning parameters, the accuracy can be enhanced. Here, I used 10000 iterations and one as learning rate value to conduct the test. It generated not bad accuracy results.