# EE5904 Neural Networks: Homework 3
National University of Singapore
Zhang Fei A0117981x

## Q1. Function Approximation with RBFN

### a). Apply exact interpolation method
### Source code

```
import numpy as np
import matplotlib.pyplot as plt

x_train = np.arange(-1,1,0.05)
y_train = 1.2*np.sin(np.pi*x_train) - np.cos(2.4*np.pi*x_train)

x_test = np.arange(-1,1,0.01)
y_test = 1.2*np.sin(np.pi*x_test) - np.cos(2.4*np.pi*x_test)

np.random.seed(6)
g_noise = np.random.normal(0,1,y_train.shape[0])
y_train_n = y_train + 0.3*g_noise

def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x):
    d = x.shape[0]
    phi = np.zeros((d,d))
    for i in range(d):
        for j in range(d):
            r = x[i]-x[j]
            phi[i][j] = guassian(r)
    return phi

def weights(x,d):
    phi = phi_matrix(x)
    return np.linalg.inv(phi).dot(d)

def predict(x_train,x_test,weights):
    y_pre = []
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(x_train.shape[0]):
            r = x_test[j] - x_train[i]
            y += weights[i]*guassian(r)
        y_pre.append(y)
    return y_pre

sigma = 0.1

w = weights(x_train,y_train_n)
pre_test = predict(x_train,x_test,w)

mean_error = np.mean(abs((pre_test - y_test)/y_test))
print(mean_error)
plt.figure()
plt.plot(x_test,y_test,x_test,pre_test)
plt.legend(('Test Dataset', 'Predicted Dataset'))
plt.ylim(-3, 3)
plt.title('result of exact interpolation method')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

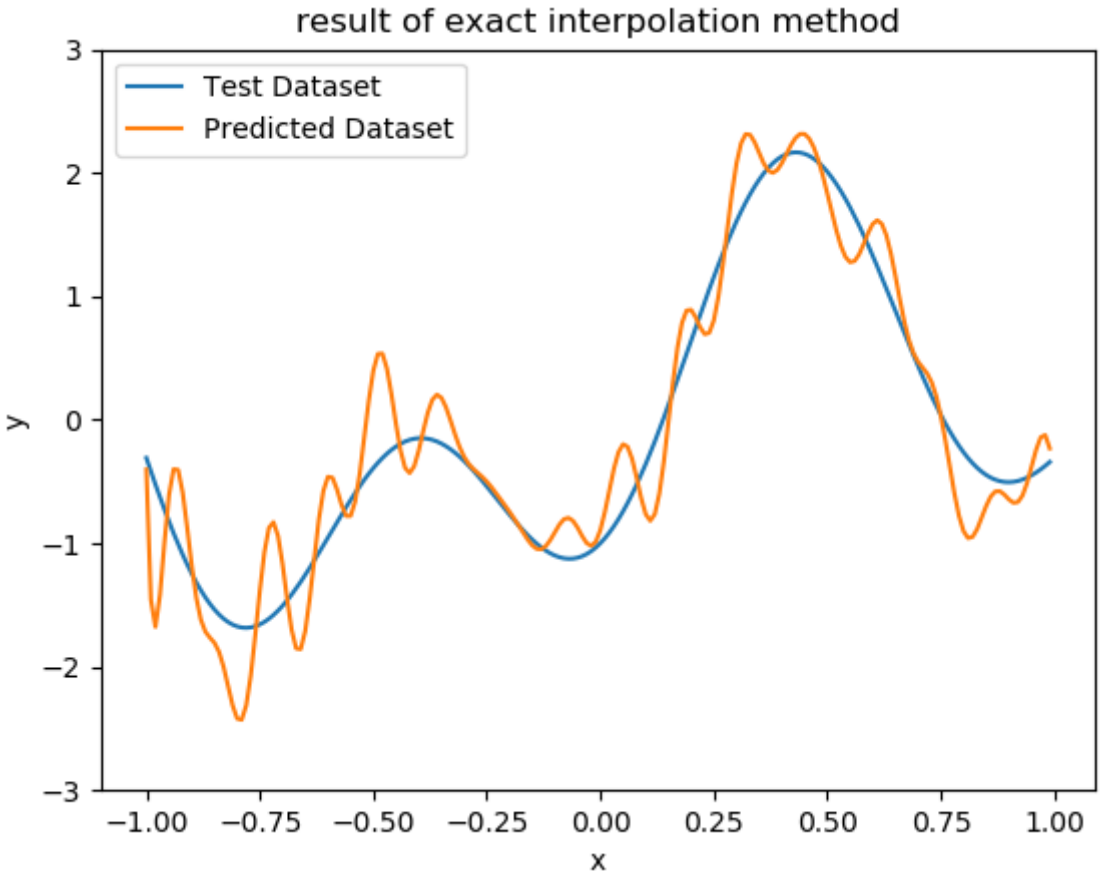



Figure 1. the result of the exact interpolation method

Figure 1 shows the test dataset output and predicted output. The predicted output does not fit well on the test dataset output. The average error rate is 0.649. The result implies that there exists overfitting along the approximation performance. This is due to the added noise is learned by the RBFN model during the training period. Therefore, this model is very sensitive to noise.

### b). Fixed Centers Selected at Random

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  import random
4.
5.  x_train = np.arange(-1,1,0.05)
6.  y_train = 1.2*np.sin(np.pi*x_train) - np.cos(2.4*np.pi*x_train)
7.
8.  x_test = np.arange(-1,1,0.01)
9.  y_test = 1.2*np.sin(np.pi*x_test) - np.cos(2.4*np.pi*x_test)
10.
11. np.random.seed(6)
12. g_noise = np.random.normal(0,1,y_train.shape[0])
13. y_train_n = y_train + 0.3*g_noise
14.
15. x_centers = []
16. y_centers = []
17. M = 15
```

```
# Select random 15 data points
idx = random.sample(range(0,len(x_train)),M)

for i in range(len(idx)):
    x_centers.append(x_train[idx[i]])
    # y_centers.append(y_train_n[idx[i]])

d_max = max(x_centers) - min(x_centers)
sigma = d_max/np.sqrt(2*M)

def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x,mu):
    N = x.shape[0]
    M = len(mu)
    phi = np.zeros((N,M))
    for i in range(N):
        for j in range(M):
            r = x[i]-mu[j]
            phi[i][j] = guassian(r)
    return phi

def weights(mu,x,d):
    phi = phi_matrix(x,mu)
    return np.linalg.inv(np.dot(phi.T,phi)).dot(phi.T).dot(d)

def predict(x_train,x_test,weights):
    y_pre = []
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(len(x_train)):
            r = x_test[j] - x_train[i]
            y += weights[i]*guassian(r)
        y_pre.append(y)
    return y_pre

w = weights(x_centers,x_train,y_train_n)
pre_test = predict(x_centers,x_test,w)
mean_error = np.mean(abs((pre_test - y_test)/y_test))
print(mean_error)

plt.figure()
plt.plot(x_test,y_test,x_test,pre_test)
plt.legend(('Test Output', 'Predicted Output'))
plt.title('result of fixed centres selected at random method')
plt.ylim(-3, 3)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

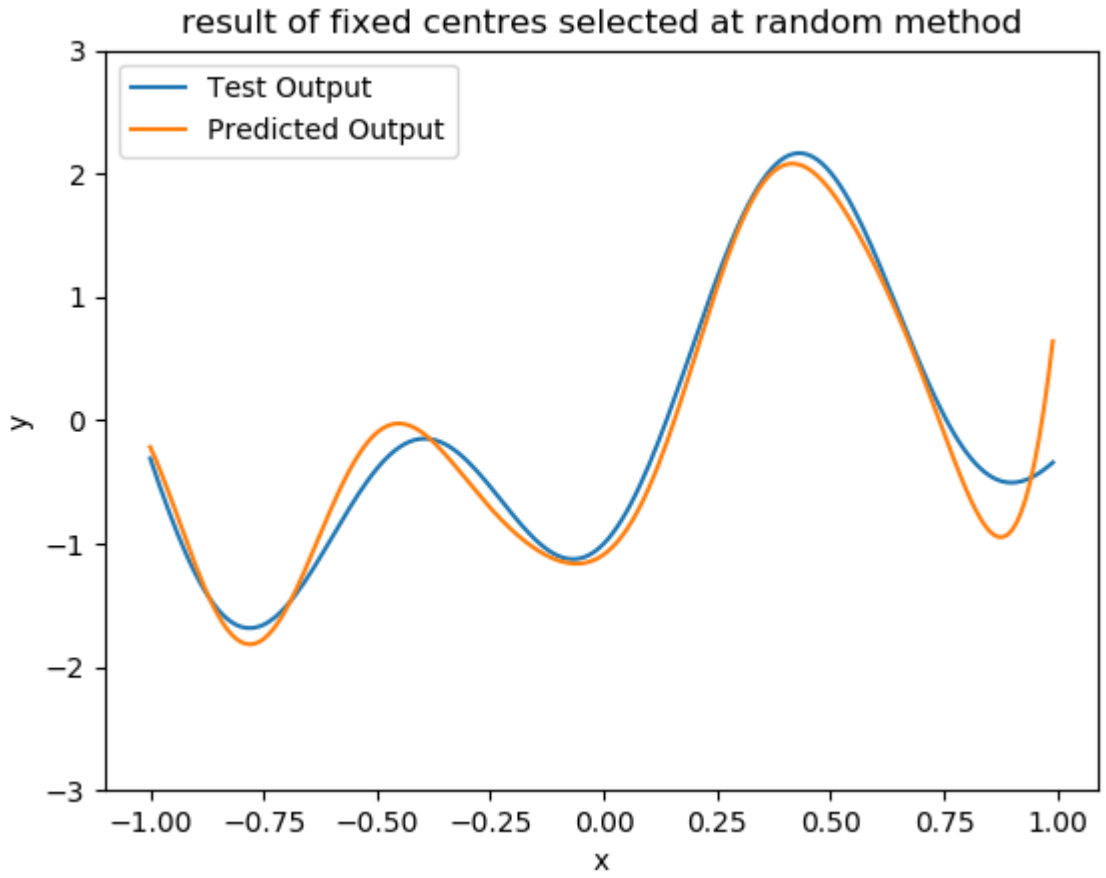



Figure 2. result of fixed centres selected at random method

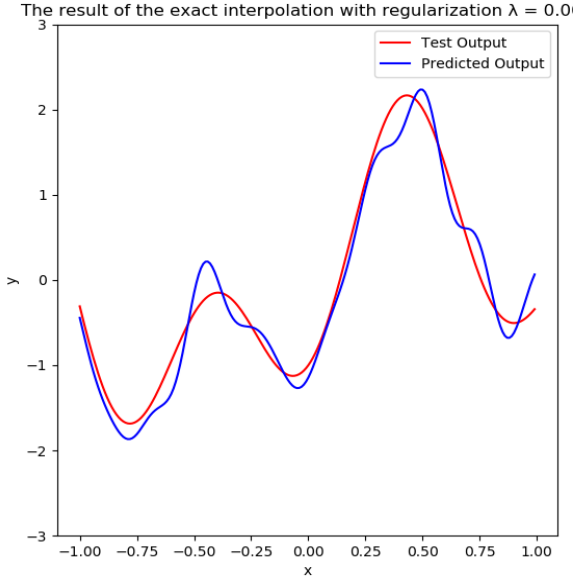
Figure 2 shows the result of the RBFN using fixed centres selected at random method, with 15 randomly selected centres. The average error rate is 0.4181. The output is quite smooth with less overfitting. The predicted approximation output of test dataset performs better than the result obtained in part a. Therefore, this method can help reduce the effect of noise.

### c) RBFN with regularization applied

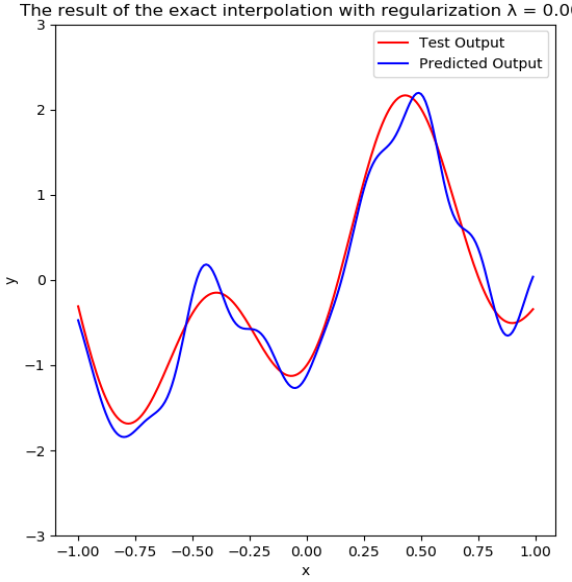
```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.  x_train = np.arange(-1,1,0.05)
5.  y_train = 1.2*np.sin(np.pi*x_train) - np.cos(2.4*np.pi*x_train)
6.
7.  x_test = np.arange(-1,1,0.01)
8.  y_test = 1.2*np.sin(np.pi*x_test) - np.cos(2.4*np.pi*x_test)
9.
10. np.random.seed(2)
11. g_noise = np.random.normal(0,1,y_train.shape[0])
12. y_train_n = y_train + 0.3*g_noise
13.
14. sigma = 0.1
15. lmd_set = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 3, 5, 10]
16.
17. def guassian(r):
18.     return np.exp(-r**2/(2*sigma**2))
19.
20. def phi_matrix(x):
21.     d = x.shape[0]
```

```
    phi = np.zeros((d,d))
    for i in range(d):
        for j in range(d):
            r = x[i]-x[j]
            phi[i][j] = guassian(r)
    return phi

def weights(x,d,lmd):
    phi = phi_matrix(x)
    I = np.identity(phi.shape[0])
    return np.linalg.inv(np.dot(phi.T,phi) + lmd*I).dot(phi.T).dot(d)

def predict(x_train,x_test,weights):
    y_pre = []
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(x_train.shape[0]):
            r = x_test[j] - x_train[i]
            y += weights[i]*guassian(r)
        y_pre.append(y)
    return y_pre

pre_test = []
for i in range(len(lmd_set)):
    w = weights(x_train,y_train_n,lmd_set[i])
    pre_test.append(predict(x_train,x_test,w))


plt.figure()
for i in list(range(0,9,2)):
    plt.subplot(1,2,1)
    plt.plot(x_test,y_test,color = 'red',label ='Test Output')
    plt.plot(x_test,pre_test[i],color = 'blue',label ='Predicted Output')
    plt.title('The result of the exact interpolation with regularization λ = '+ str(lmd_set[i]))
    plt.ylim(-3,3)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

    plt.subplot(1,2,2)
    plt.plot(x_test,y_test,color = 'red',label ='Test Output')
    plt.plot(x_test,pre_test[i+1],color = 'blue',label ='Predicted Output')
    plt.title('The result of the exact interpolation with regularization λ = '+ str(lmd_set[i+1]))
    plt.ylim(-3,3)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
    plt.clf()

err_set = []
for i in range(len(lmd_set)):
    err_set.append(np.mean(abs((pre_test[i] - y_test)/y_test)))
print(min(err_set))
plt.plot(lmd_set,err_set)
plt.xlabel('Regularization Factor')
plt.ylabel('Average Relative Error')
plt.title('Average Relative Error versus the Regularization Factor on Test Data Set')
plt.show()
```

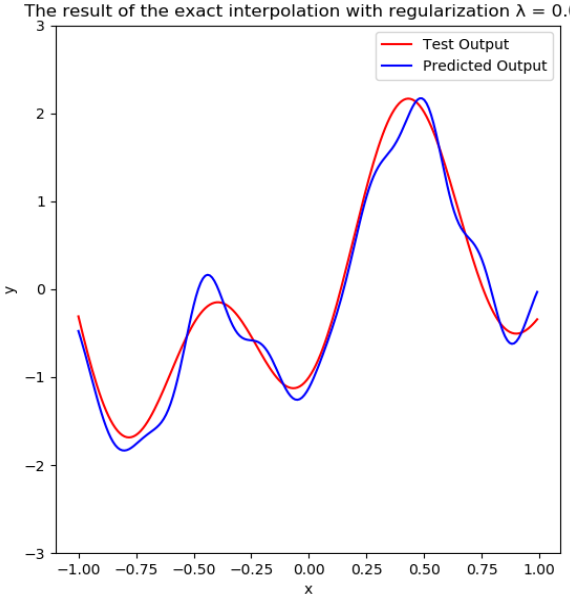
The result of the exact interpolation with regularization λ = 0.001



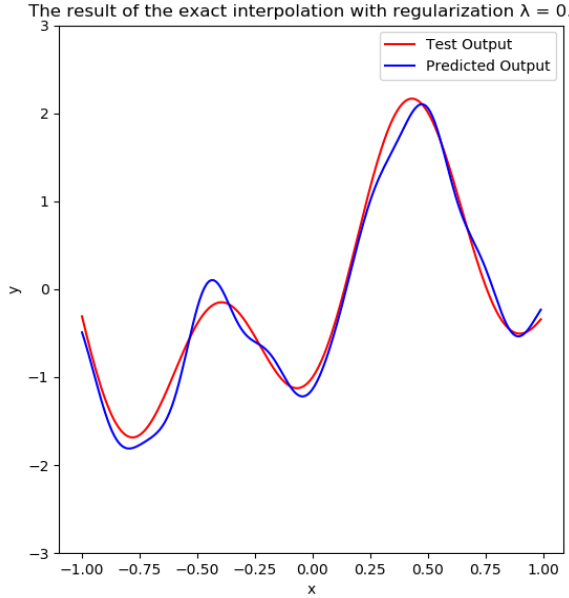

The result of the exact interpolation with regularization λ = 0.005



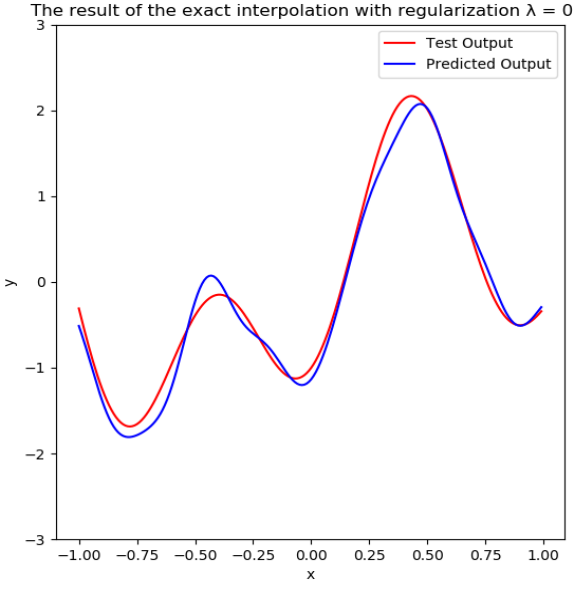

The result of the exact interpolation with regularization λ = 0.01



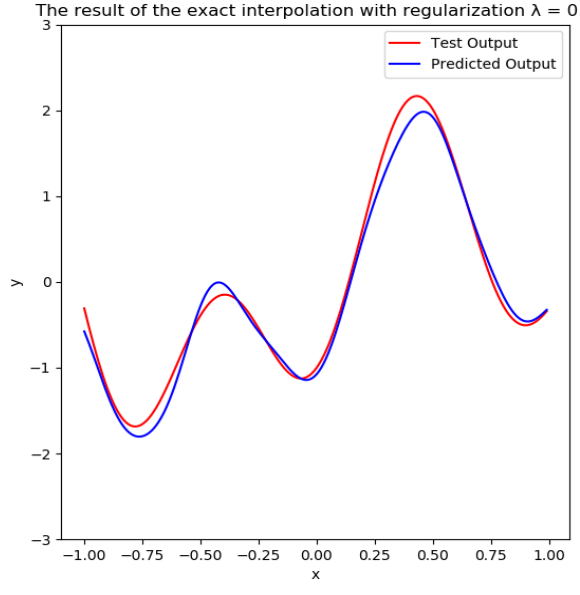

The result of the exact interpolation with regularization λ = 0.05

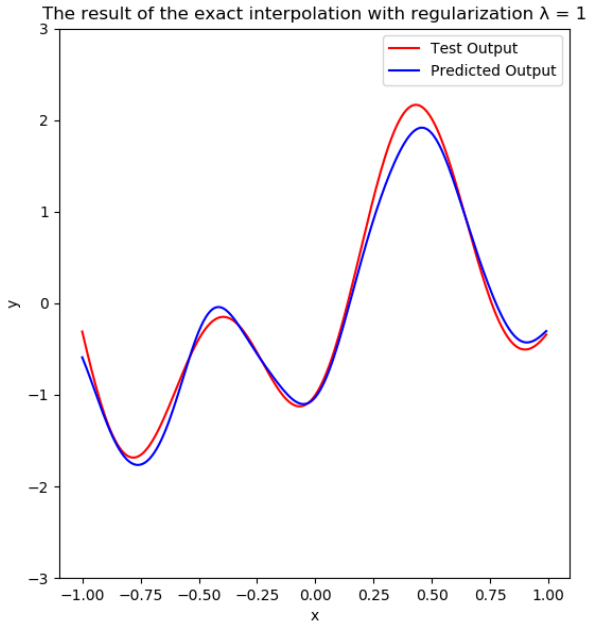


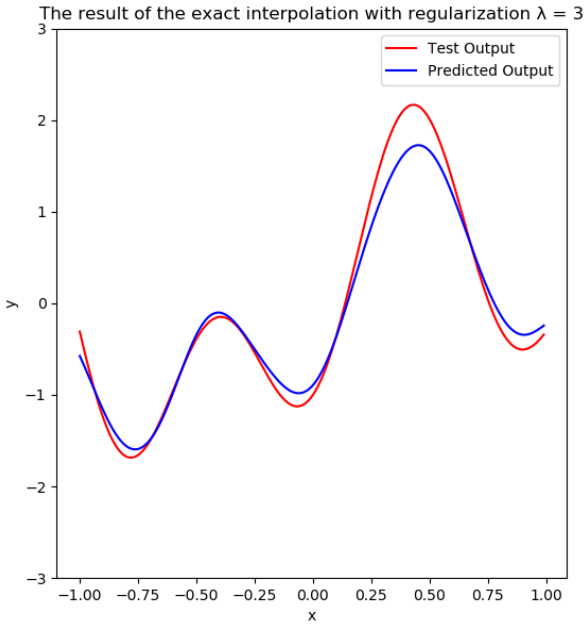

The result of the exact interpolation with regularization λ = 0.1

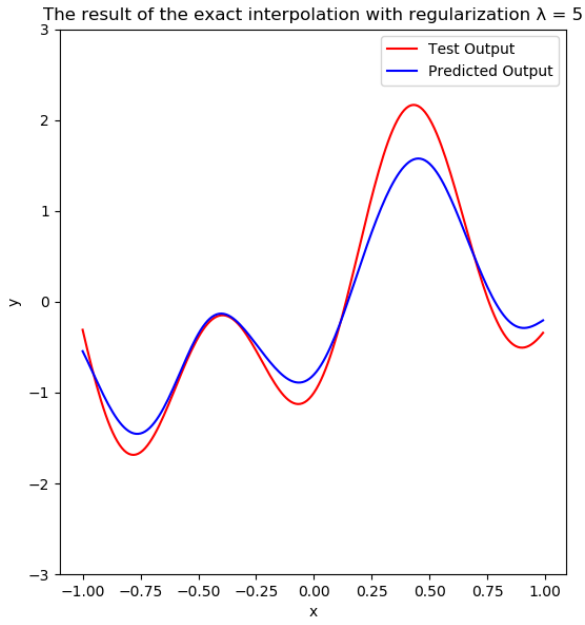


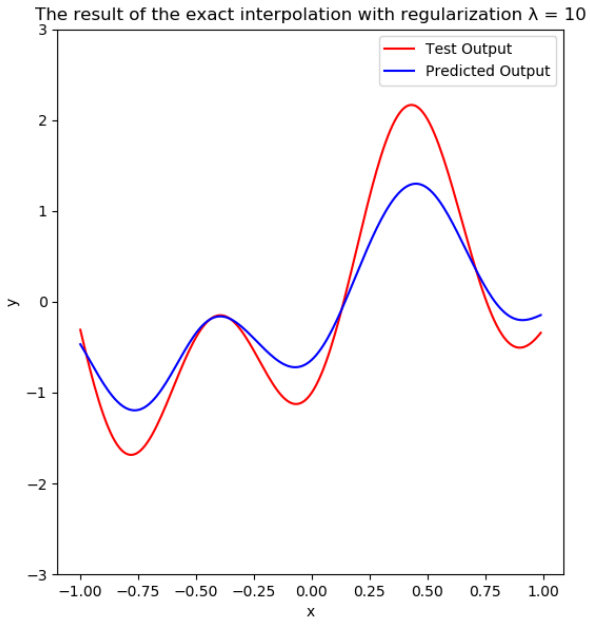

The result of the exact interpolation with regularization λ = 0.5






The figures above show the performance of RBFN in exact interpolation method with different λ values. And we can find that there is no severe overfitting along the output compare the result obtained in part a. Therefore, regularization help reduces variance. And from different values of λ, we can also find that the overfitting is reduced when λ increase from 0 to 1. And if keep increasing λ, the predicted output will be underfit. Thus, we need to tune the regularization factor λ to make the model achieve good performance.

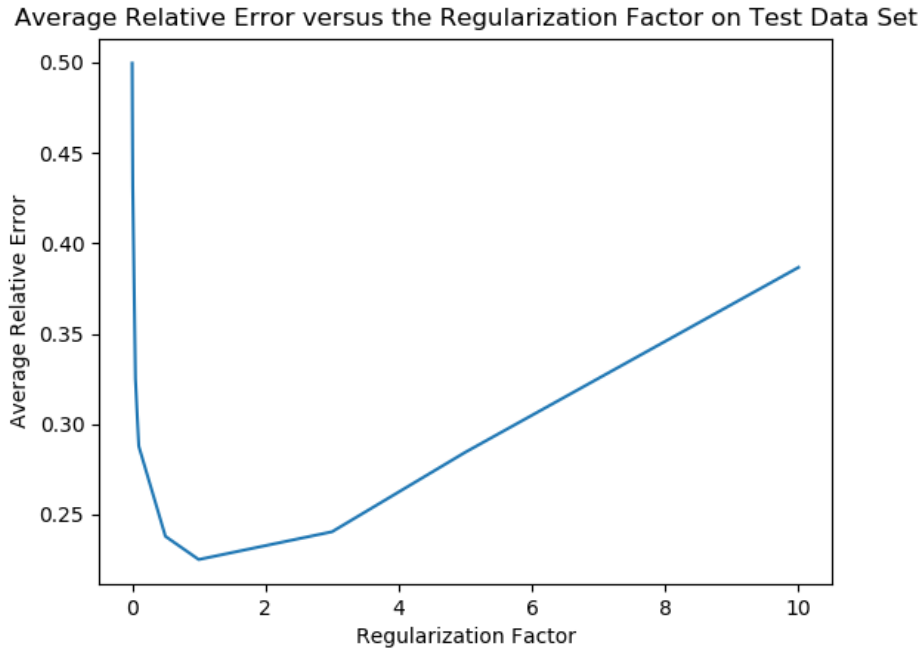


Figure 3. Average Relative Error vs the Regularization Factor on Test Data Set

The Figure 3 shows that when λ = 1, the model achieves the best result. And the average relative error is 0.225.

**Q2. Handwritten Digits Classification using RBFN**

My matric Number is A0117981X, mark digit 8 as class 1, mark digit 1 as class 0.
Source Code: in exact interpolation method without regularization

```
import numpy as np
from scipy import io
import matplotlib.pyplot as plt

# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train_raw = mat['train_data'].T           # shape (1000,784)
y_train_raw = mat['train_classlabel'].T     # shape (1000,1)
x_test_raw = mat['test_data'].T             # shape (250,784)
y_test_raw = mat['test_classlabel'].T       # shape (250,1)
# my matrix No. A0117981X, mark digit 8 class 1 and digit 1 as class 0
x_train = x_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
y_train = y_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
x_test = x_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_test = y_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_train = np.where(y_train == 8, 1,0)
y_test = np.where(y_test == 8, 1,0)
np.random.seed(8)

def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x):
    d = x.shape[0]
    phi = np.zeros((d,d))
    for i in range(d):
        for j in range(d):
            r = x[i]-x[j]
            phi[i][j] = guassian(np.linalg.norm(r))
    return phi

```

```
def weights(x,d):
    phi = phi_matrix(x)
    return np.linalg.inv(phi).dot(d)

def predict(x_train,x_test,W):
    y_pre = np.zeros((x_test.shape[0], 1))
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(x_train.shape[0]):
            r = x_test[j] - x_train[i]
            y += W[i]*guassian(np.linalg.norm(r))
        y_pre[j][0] = y
    return y_pre

sigma = 100
w = weights(x_train,y_train)
pre_test = predict(x_train,x_test,w)
pre_train = predict(x_train,x_train,w)

TrAcc = []
TeAcc = []
TrN = y_train.shape[0]
TeN = y_test.shape[0]
Threshold = []
for i in range(1000):
    t = (np.amax(pre_train) - np.amin(pre_train))*i/1000 + np.amin(pre_train)
    Threshold.append(t)
    TrAcc.append((np.sum(y_train[pre_train<t]==0) + np.sum(y_train[pre_train>=t]==1
)) / TrN)
    TeAcc.append((np.sum(y_test[pre_test<t]==0) + np.sum(y_test[pre_test>=t]==1)) /
 TeN)

plt.figure()
plt.plot(Threshold,TrAcc,Threshold,TeAcc)
plt.title('Accuracy in Exact Interpolation Method without regularization')
plt.legend(('Train ', 'Test'))
plt.xlabel('Threshold')
plt.ylabel('Accuracy')
plt.show()
```

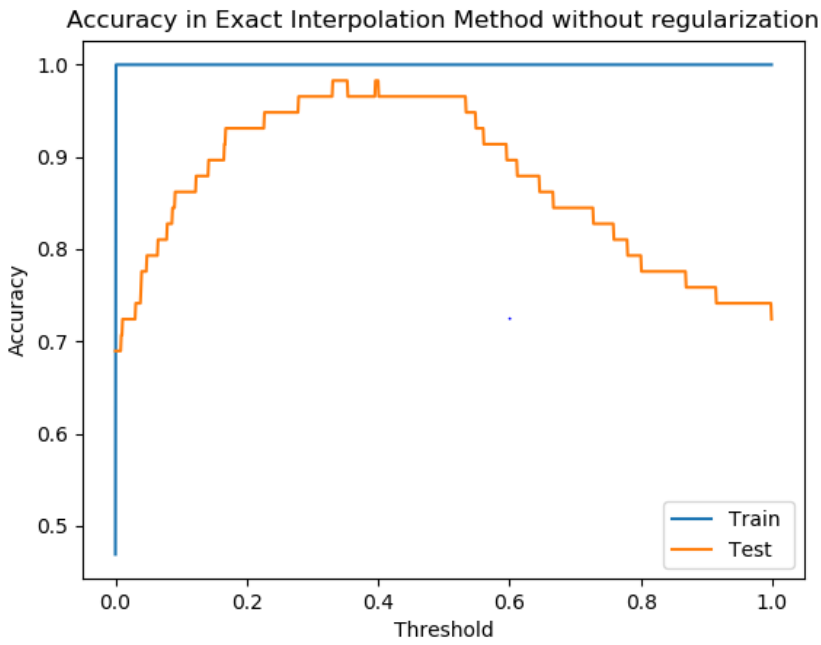


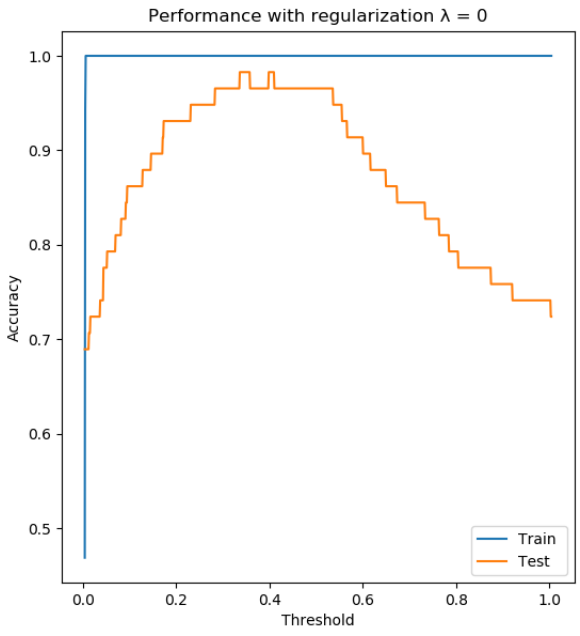

Figure 4. Accuracy vs Threshold in Exact Interpolation Method without regularization

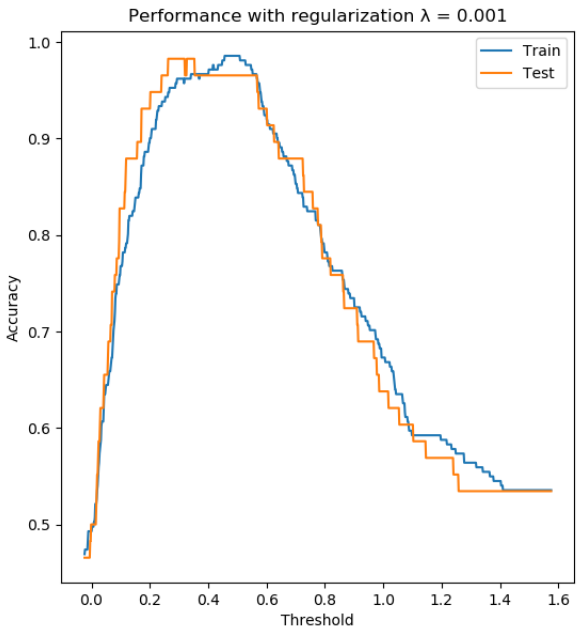
The figure shows that the accuracy of training dataset jumps from 0 to 1 when threshold is above 0. And the accuracy of testing dataset starts from 0.68 when Threshold is 0, it keeps increasing to 0.97 when threshold is 0.3. Then it starts to decrease when threshold keeps increasing from 0.5.

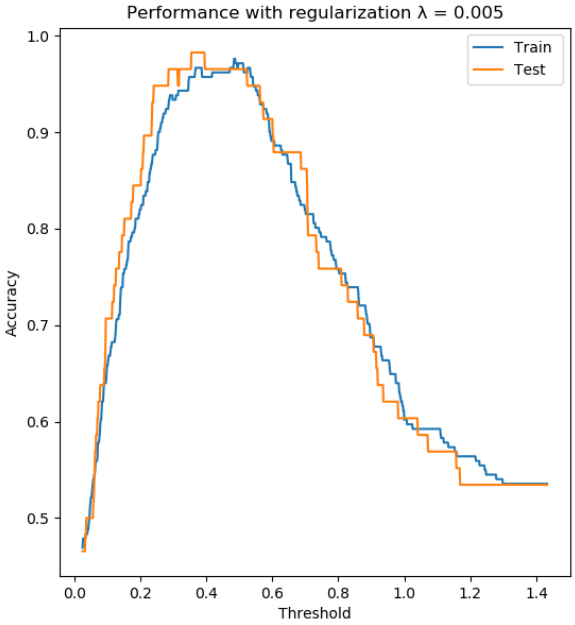
Source Code: in exact interpolation method with regularization

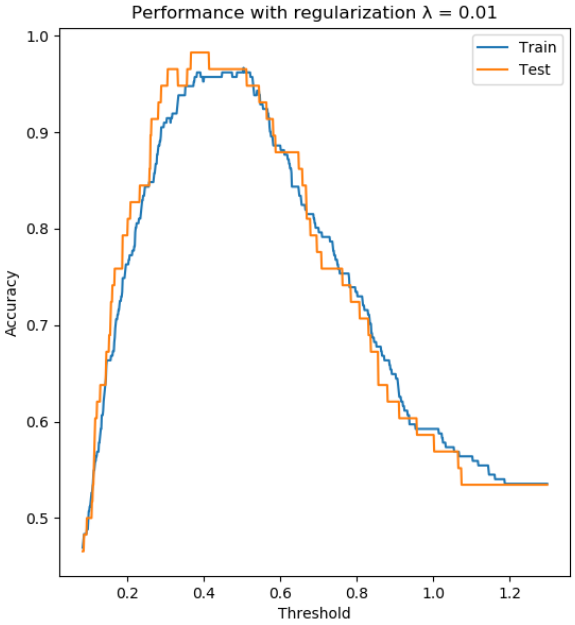
```
import numpy as np
from scipy import io
import matplotlib.pyplot as plt

# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train_raw = mat['train_data'].T           # shape (1000,784)
y_train_raw = mat['train_classlabel'].T     # shape (1000,1)
x_test_raw = mat['test_data'].T             # shape (250,784)
y_test_raw = mat['test_classlabel'].T       # shape (250,1)

# my matrix No. A0117981X, mark digit 8 class 1 and digit 1 as class 0
x_train = x_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
y_train = y_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
x_test = x_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_test = y_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]

y_train = np.where(y_train == 8, 1,0)
y_test = np.where(y_test == 8, 1,0)

np.random.seed(8)
sigma = 100
def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x):
    d = x.shape[0]
    phi = np.zeros((d,d))
    for i in range(d):
        for j in range(d):
            r = x[i]-x[j]
            phi[i][j] = guassian(np.linalg.norm(r))
    return phi

def weights(x,d,lmd):
    phi = phi_matrix(x)
    I = np.identity(phi.shape[0])
    return np.linalg.inv(np.dot(phi.T,phi) + lmd*I).dot(phi.T).dot(d)

def predict(x_train,x_test,W):
    y_pre = np.zeros((x_test.shape[0], 1))
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(x_train.shape[0]):
            r = x_test[j] - x_train[i]
            y += W[i]*guassian(np.linalg.norm(r))
        y_pre[j][0] = y
    return y_pre

lmd_set = [0,0.001,0.005,0.01,0.05,0.1,0.5,1,5,10]
pre_test = []
pre_train = []
for i in range(len(lmd_set)):
    w = weights(x_train,y_train,lmd_set[i])
    pre_train.append(predict(x_train,x_train,w))
    pre_test.append(predict(x_train,x_test,w))

```

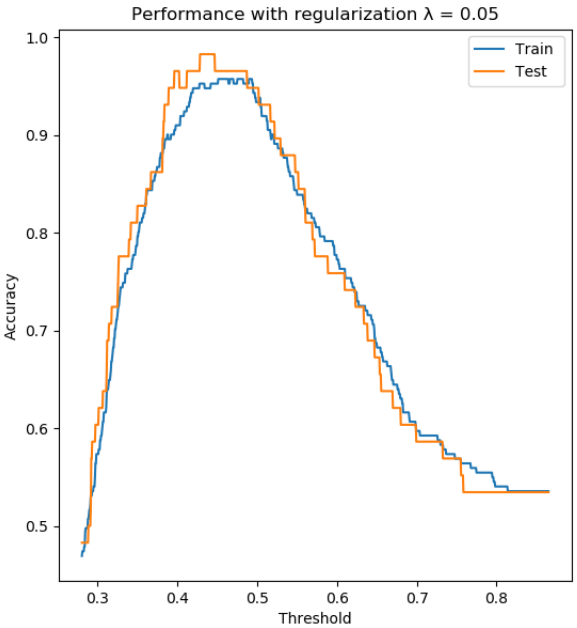
```
TrN = y_train.shape[0]
TeN = y_test.shape[0]
TrAccuracy = []
TeAccuracy = []
thr = []
for j in range(len(lmd_set)):
    TrAcc = []
    TeAcc = []
    Threshold = []
    for i in range(1000):
        t = (np.amax(pre_train[j]) - np.amin(pre_train[j]))*i/1000 + np.amin(pre_train[j])
        Threshold.append(t)

        TrAcc.append((np.sum(y_train[pre_train[j]<t]==0) + np.sum(y_train[pre_train[j]>=t]==1)) / TrN)
        TeAcc.append((np.sum(y_test[pre_test[j]<t]==0) + np.sum(y_test[pre_test[j]>=t]==1)) / TeN)
    thr.append(Threshold)
    TrAccuracy.append(TrAcc)
    TeAccuracy.append(TeAcc)

plt.figure()
for i in list(range(0,9,2)):
    plt.subplot(1,2,1)
    plt.plot(thr[i],TrAccuracy[i],thr[i],TeAccuracy[i])
    plt.title('Performance with regularization λ = '+str(lmd_set[i]))
    plt.legend(('Train ', 'Test'))
    plt.xlabel('Threshold')
    plt.ylabel('Accuracy')

    plt.subplot(1,2,2)
    plt.plot(thr[i+1],TrAccuracy[i+1],thr[i+1],TeAccuracy[i+1])
    plt.title('Performance with regularization λ = '+str(lmd_set[i+1]))
    plt.legend(('Train', 'Test'))
    plt.xlabel('Threshold')
    plt.ylabel('Accuracy')
    plt.show()
    plt.clf()
```

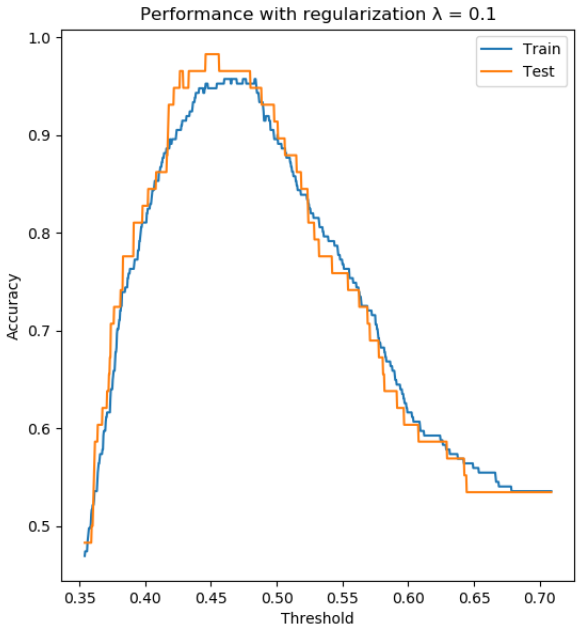


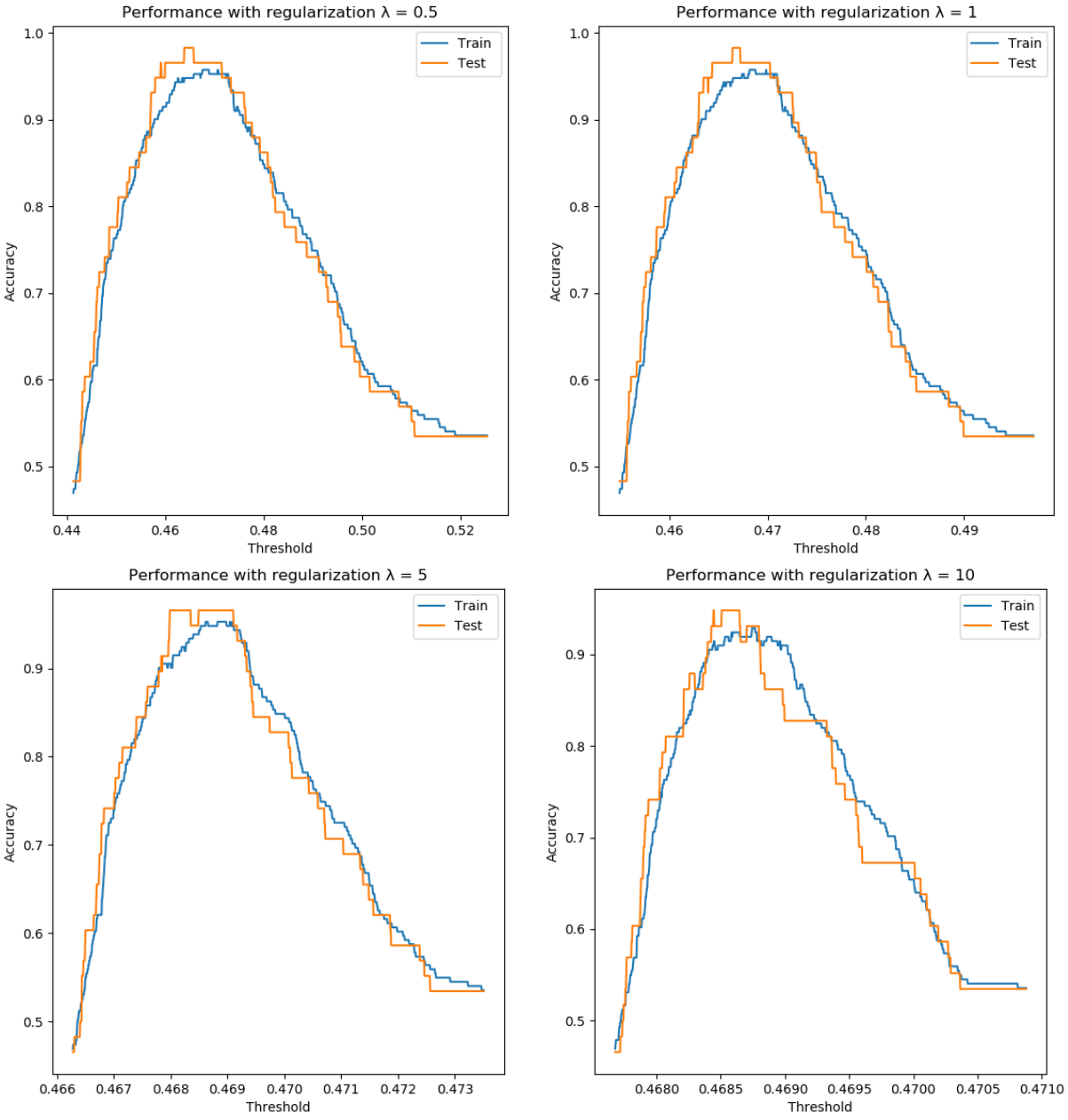




Performance with regularization λ = 0.005
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


Performance with regularization λ = 0.01
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
0.2
0.4
0.6
0.8
1.0
1.2


Performance with regularization λ = 0.05
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
0.3
0.4
0.5
0.6
0.7
0.8


Performance with regularization λ = 0.1
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
0.35
0.40
0.45
0.50
0.55
0.60
0.65
0.70




The Figures above show the performance of RBFN model with different regularization factors λ. When λ is 0, which means there is no regularization factor. And the result is almost the same as first part. As λ increases, the accuracies of training dataset and testing dataset increase similarly. The accuracy of testing set can reach maximum of 97% when the threshold value is about 0.45. And as λ keeps increase to a large value, both accuracies will degrade.

2b). **Apply Fixed Centers Selected at Random Method**
Source Code:

```
1.  import numpy as np
2.  from scipy import io
```

```
import matplotlib.pyplot as plt
import random
from scipy.spatial.distance import cdist

# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train_raw = mat['train_data'].T           # shape (1000,784)
y_train_raw = mat['train_classlabel'].T     # shape (1000,1)
x_test_raw = mat['test_data'].T             # shape (250,784)
y_test_raw = mat['test_classlabel'].T       # shape (250,1)
# my matrix No. A0117981X, mark digit 8 class 1 and digit 1 as class 0
x_train = x_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
y_train = y_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
x_test = x_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_test = y_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_train = np.where(y_train == 8, 1,0)
y_test = np.where(y_test == 8, 1,0)


# np.random.seed(8)
def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x,centre):
    N = x.shape[0]
    M = centre.shape[0]
    phi = np.zeros((N,M))
    for i in range(N):
        for j in range(M):
            r = x[i]-centre[j]
            phi[i][j] = guassian(np.linalg.norm(r))
    return phi

def weights(x,d,centre):
    phi = phi_matrix(x,centre)
    return np.linalg.inv(np.dot(phi.T,phi)).dot(phi.T).dot(d)

def predict(centre,x_test,W):
    y_pre = np.zeros((x_test.shape[0], 1))
    for j in range(x_test.shape[0]):
        y = 0
        for i in range(centre.shape[0]):
            r = x_test[j] - centre[i]
            y += W[i]*guassian(np.linalg.norm(r))
        y_pre[j][0] = y
    return y_pre

M = 100
# Select random 100 data points
# np.random.seed(12)
idx = random.sample(range(0,len(x_train)),M)
centres = x_train[idx,:]

d = cdist(centres,centres,metric='euclidean')
d_max = np.amax(d)
sigma = d_max/np.sqrt(2*M)
print('The width is '+str(sigma))

w = weights(x_train,y_train,centres)
pre_test = predict(centres,x_test,w)
pre_train = predict(centres,x_train,w)


TrAcc = []
TeAcc = []
thr = []
```

```
TrN = y_train.shape[0]
TeN = y_test.shape[0]
Thr = []
for i in range(1000):
    t = (np.amax(pre_train) - np.amin(pre_train))*i/1000 + np.amin(pre_train)
    Thr.append(t)

    TrAcc.append((np.sum(y_train[pre_train<t]==0) + np.sum(y_train[pre_train>=t]==1
)) / TrN)
    TeAcc.append((np.sum(y_test[pre_test<t]==0) + np.sum(y_test[pre_test>=t]==1)) /
 TeN)

plt.figure()
plt.plot(Thr,TrAcc,Thr,TeAcc)
plt.title('Fixed Centres Selected at Random with width at '+str(round(sigma,2)))
plt.legend(('Train ', 'Test'))
plt.xlabel('Threshold')
plt.ylabel('Accuracy')
plt.show()
```

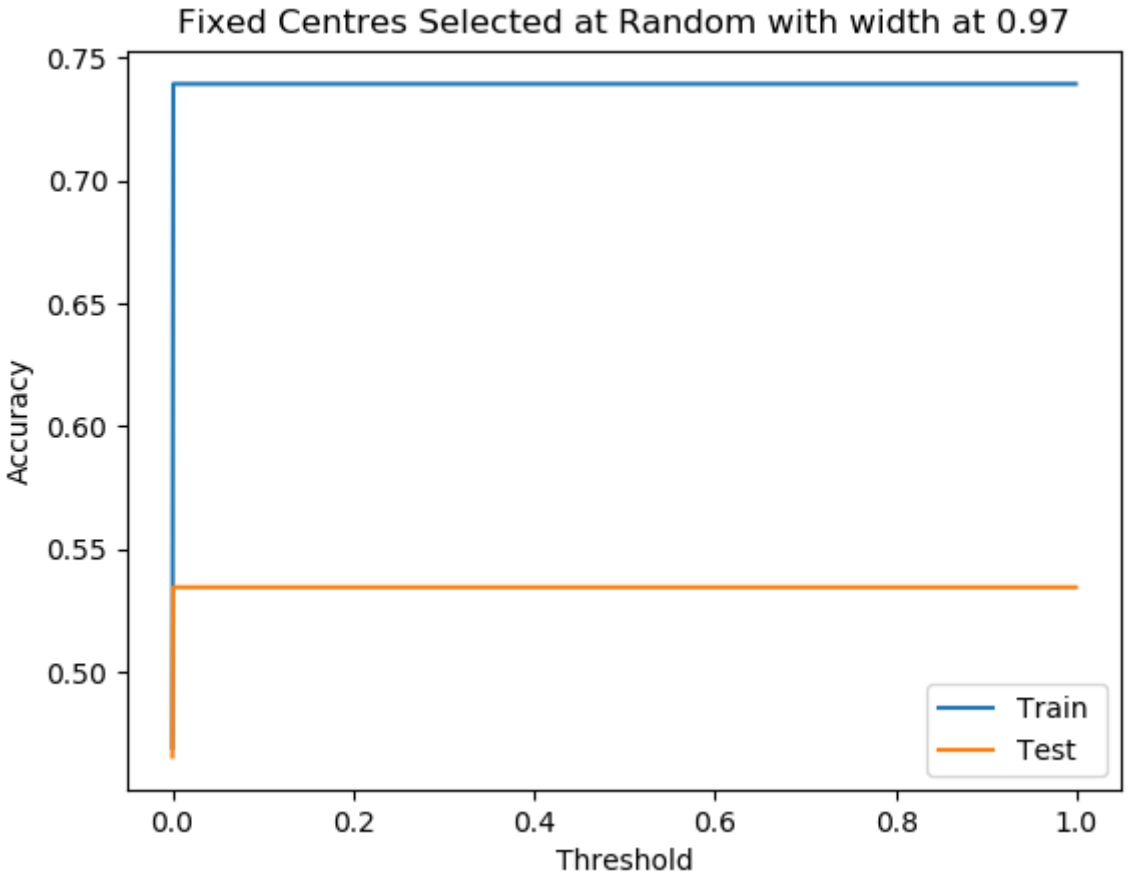


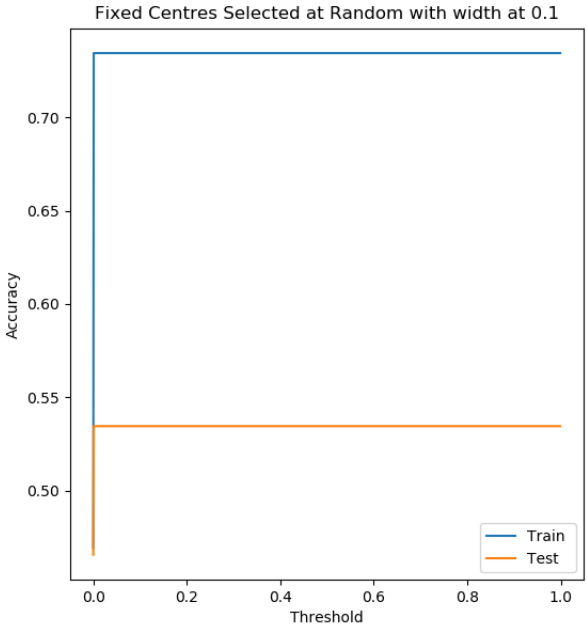

Figure 5. Performance of Fixed Centres Selected at Random with width

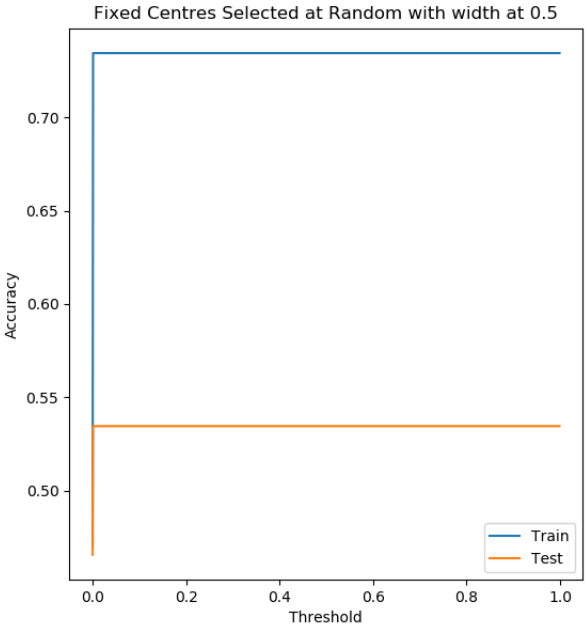
Figure 5. shows the Performance of Fixed Centres Selected with width. And the width is calculated as:

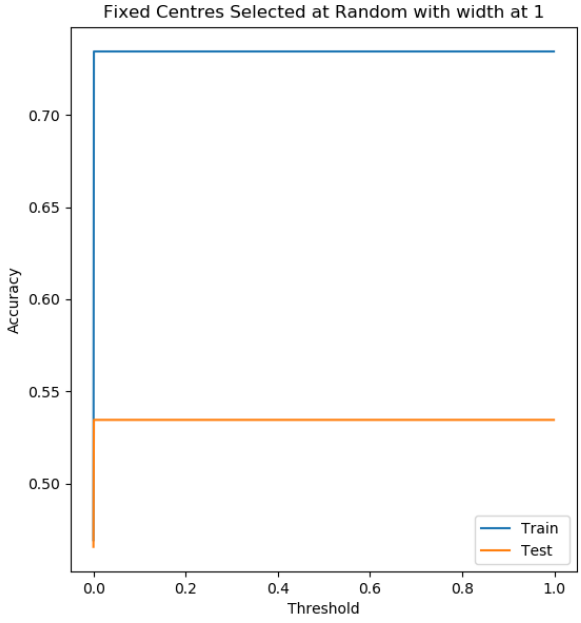
$$\text{width} = \frac{d_{max}}{\sqrt{2M}}$$

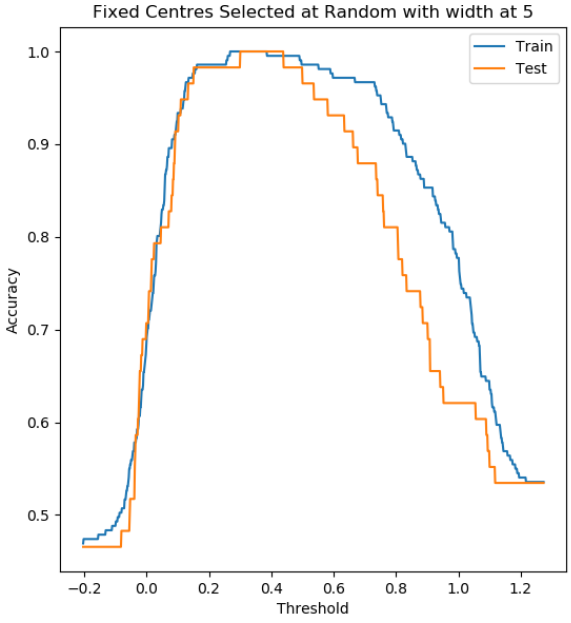
we can find that both training and testing set accuracies are not performing well and decision boundary is narrow. This is because the RBF is too peaked, this will

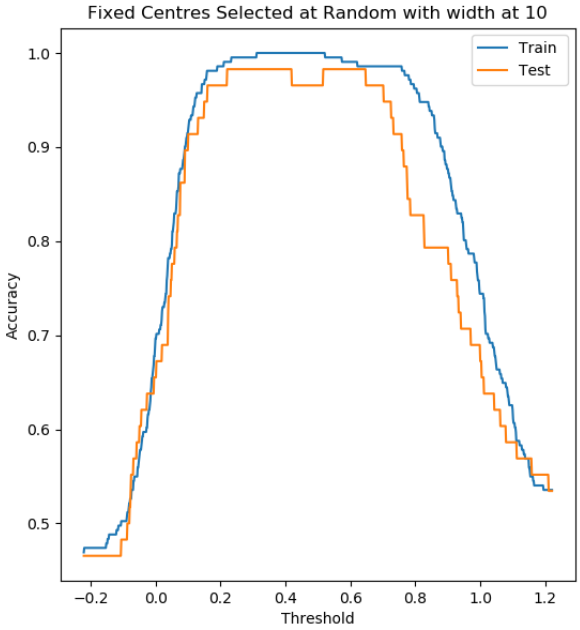
squeeze the distribution of data points. Therefore, this leads to a hard decision boundary and degrade performance.

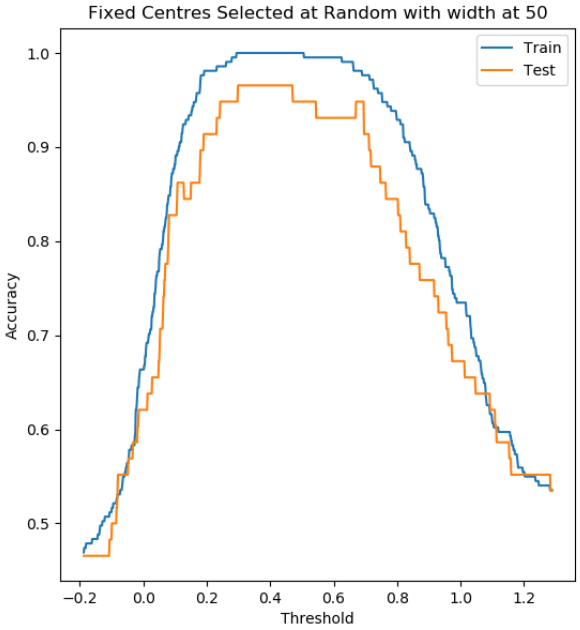


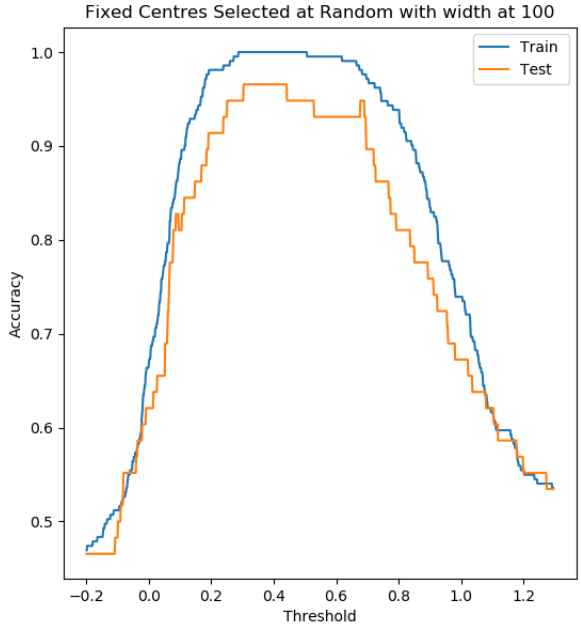



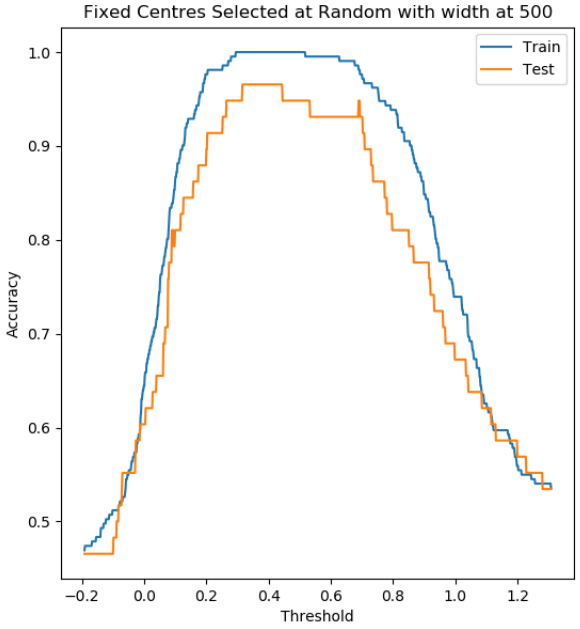



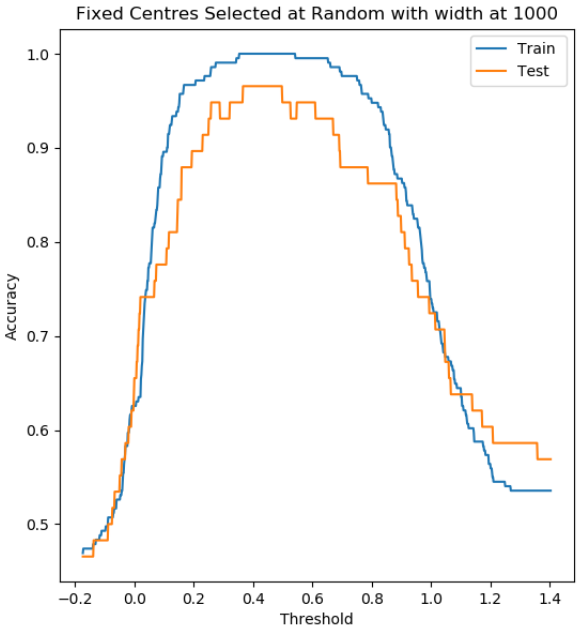



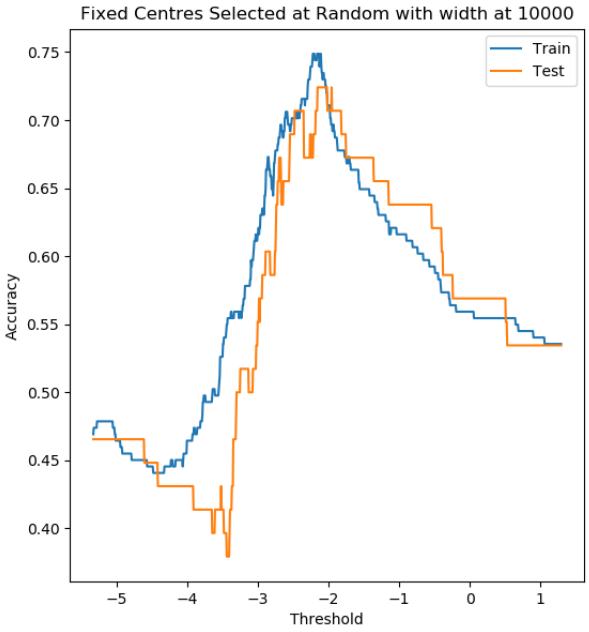

Fixed Centres Selected at Random with width at 10
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Fixed Centres Selected at Random with width at 50
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Fixed Centres Selected at Random with width at 100
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Fixed Centres Selected at Random with width at 500
Train
Test
Accuracy
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2




Source code: for different width set, other functions are same as above

```
M = 100
# Select random 100 data points
np.random.seed(8)
idx = random.sample(range(0,len(x_train)),M)
centres = x_train[idx,:]

d = cdist(centres,centres,metric='euclidean')
d_max = np.amax(d)
# sigma = d_max/np.sqrt(2*M)

sigma_set = [0.1,0.5,1,5,10,50,100,500,1000,10000]
pre_test = []
pre_train = []
for j in range(len(sigma_set)):
    sigma = sigma_set[j]
    w = weights(x_train,y_train,centres)
    pre_test.append(predict(centres,x_test,w))
    pre_train.append(predict(centres,x_train,w))


TrN = y_train.shape[0]
TeN = y_test.shape[0]
TrAccuracy = []
TeAccuracy = []
thr = []

for j in range(len(sigma_set)):
    TrAcc = []
    TeAcc = []
    Threshold = []
    for i in range(1000):
        t = (np.amax(pre_train[j]) - np.amin(pre_train[j]))*i/1000 + np.amin(pre_tr
ain[j])
        Threshold.append(t)

        TrAcc.append((np.sum(y_train[pre_train[j]<t]==0) + np.sum(y_train[pre_train
[j]>=t]==1)) / TrN)
        TeAcc.append((np.sum(y_test[pre_test[j]<t]==0) + np.sum(y_test[pre_test[j]>
=t]==1)) / TeN)
    thr.append(Threshold)
```

```
    TrAccuracy.append(TrAcc)
    TeAccuracy.append(TeAcc)


plt.figure()
for i in list(range(0,9,2)):
    plt.subplot(1,2,1)
    plt.plot(thr[i],TrAccuracy[i],thr[i],TeAccuracy[i])
    plt.title('Fixed Centres Selected at Random with width at '+str(round(sigma_set[i],2)))
    plt.legend(('Train ', 'Test'))
    plt.xlabel('Threshold')
    plt.ylabel('Accuracy')

    plt.subplot(1,2,2)
    plt.plot(thr[i+1],TrAccuracy[i+1],thr[i+1],TeAccuracy[i+1])
    plt.title('Fixed Centres Selected at Random with width at '+str(round(sigma_set[i+1],2)))
    plt.legend(('Train', 'Test'))
    plt.xlabel('Threshold')
    plt.ylabel('Accuracy')
    plt.show()
    plt.clf()
```

The figures above show the performance RBFN model using Fixed Centres Selected at Random Method with different widths from 0.1 to 10000. We can see that both training and testing set accuracy increase and then decrease as width increases. The accuracy can achieve above 95% when width is from 5 to 1000. However, if width is too large, both training and testing accuracies are getting worse. Therefore, we need to tune the width to achieve good performance, which means we cannot select the width that is too small and too large.

### 2c) Apply K- Mean Clustering with 2 centres

Source code:

```
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy import io
from scipy.spatial.distance import cdist
from sklearn.decomposition import PCA

# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train_raw = mat['train_data'].T           # shape (1000,784)
y_train_raw = mat['train_classlabel'].T     # shape (1000,1)
x_test_raw = mat['test_data'].T             # shape (250,784)
y_test_raw = mat['test_classlabel'].T       # shape (250,1)
# my matrix No. A0117981X, mark digit 8 class 1 and digit 1 as class 0
x_train = x_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
y_train = y_train_raw[np.where((y_train_raw ==8) | (y_train_raw ==1))[0]]
x_test = x_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_test = y_test_raw[np.where((y_test_raw ==8) | (y_test_raw ==1))[0]]
y_train = np.where(y_train == 8, 1,0)
y_test = np.where(y_test == 8, 1,0)
# np.random.seed(12)
def euclDistance(vector1,vector2):
    return np.sqrt(np.sum(np.power(vector2 - vector1,2)))

def initCentroids(dataSet,k):
```

```
    nuSamples, dim = dataSet.shape
    index = random.sample(range(nuSamples),k) #randomly generate k non repeated num , return in list
    # centroids = np.array([dataSet[i,:] for i in index]) # create k centroids from the array
    centroids = x_train[np.random.randint(dataSet.shape[0], size=2), :] #random centres
    return centroids

def k_means(dataSet,k):
    nuSamples = dataSet.shape[0]
    clusterAssment = np.zeros((nuSamples,2))
    centroids = initCentroids(dataSet,k)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(nuSamples):
            minDistance = float("inf")
            ownGroup = 0
            for j in range(k):
                distance = euclDistance(centroids[j,:],dataSet[i,:])
                if distance < minDistance:
                    minDistance = distance
                    ownGroup = j
            if clusterAssment[i,1] != ownGroup:
                clusterAssment[i,1] = ownGroup
                clusterChanged = True
        for j in range(k):
            pointsInCluster = dataSet[np.nonzero(clusterAssment[:,1]==j)[0]]

            centroids[j,:] = np.mean(pointsInCluster,axis =0)
        return centroids

k = 2
centroids = k_means(x_train, k)

d = cdist(centroids,centroids,metric='euclidean')
d_max = np.amax(d)
sigma = d_max/np.sqrt(2*k)

def guassian(r):
    return np.exp(-r**2/(2*sigma**2))

def phi_matrix(x,centre):
    N = x.shape[0]
    M = centre.shape[0]
    phi = np.zeros((N,M))
    for i in range(N):
        for j in range(M):
            r = x[i]-centre[j]
            phi[i][j] = guassian(np.linalg.norm(r))
    return phi

def weights(x,d,centre):
    phi = phi_matrix(x,centre)
    return np.linalg.inv(np.dot(phi.T,phi)).dot(phi.T).dot(d)

def predict(centre,x,W):
    y_pre = np.zeros((x.shape[0], 1))
    for j in range(x.shape[0]):
        y = 0
        for i in range(centre.shape[0]):
            r = x[j] - centre[i]
            y += W[i]*guassian(np.linalg.norm(r))
        y_pre[j][0] = y
    return y_pre
```

```
88.
89.
90. w = weights(x_train,y_train,centroids)
91. pre_test = predict(centroids,x_test,w)
92. pre_train = predict(centroids,x_train,w)
93.
94.
95. TrAcc = []
96. TeAcc = []
97. thr = []
98. TrN = y_train.shape[0]
99. TeN = y_test.shape[0]
100.        Thr = []
101.        for i in range(1000):
102.            t = (np.amax(pre_train) - np.amin(pre_train))*i/1000 + np.amin(pre_train
    )
103.            Thr.append(t)
104.
105.            TrAcc.append((np.sum(y_train[pre_train<t]==0) + np.sum(y_train[pre_train
    >=t]==1)) / TrN)
106.            TeAcc.append((np.sum(y_test[pre_test<t]==0) + np.sum(y_test[pre_test>=t]
    ==1)) / TeN)
107.
108.        # accuracy
109.        plt.figure()
110.        plt.plot(Thr,TrAcc,Thr,TeAcc)
111.        plt.title('Accuracy in K-Mean Clustering with 2 centres')
112.        plt.legend(('Train ', 'Test'))
113.        plt.xlabel('Threshold')
114.        plt.ylabel('Accuracy')
115.        plt.show()
116.        plt.clf()
117.
118.
119.        mean_value = np.zeros((2,784))
120.        for i in range(k):
121.            mean_value[i,:] = np.mean(x_train[np.nonzero(y_train[:,0]==i)[0]],axis =
    0)
122.
123.
124.        # dimension reduction
125.        pca = PCA(2)
126.        projected = pca.fit_transform(centroids)
127.        classmean = pca.fit_transform(mean_value)
128.
129.        plt.scatter(projected[0:,0],projected[0:,1],s=400,c='red',marker='o',alpha=0
    .5,label='Centre A')
130.        plt.scatter(classmean[1:,0],classmean[1:,1],s=400,c='black',marker='o',alpha
    =0.5,label='Class 0 mean')
131.
132.        plt.scatter(projected[1:,0],projected[1:,1],s=400,c='blue',marker='o',alpha=
    0.5,label='Centre B')
133.
134.        plt.scatter(classmean[0:,0],classmean[0:,1],s=400,c='green',marker='o',alpha
    =0.5,label='Class 1 mean')
135.
136.
137.        plt.xlabel('component 1')
138.        plt.ylabel('component 2')
139.        plt.legend(prop = {'size':16})
140.        plt.show()
141.        plt.clf()
142.
143.
144.        plt.subplot(2,2,1)
145.        plt.imshow(centroids[0].reshape(28,28).T,cmap='gray')
```

```
146.        plt.title('Centre A')
147.        plt.axis('off')
148.
149.        plt.subplot(2,2,2)
150.        plt.imshow(centroids[1].reshape(28,28).T,cmap='gray')
151.        plt.title('Centre B')
152.        plt.axis('off')
153.
154.        plt.subplot(2,2,3)
155.        plt.imshow(mean_value[1].reshape(28,28).T,cmap='gray')
156.        plt.title('Class 1 mean')
157.        plt.axis('off')
158.
159.        plt.subplot(2,2,4)
160.        plt.imshow(mean_value[0].reshape(28,28).T,cmap='gray')
161.        plt.title('Class 0 mean')
162.        plt.axis('off')
163.        plt.show()
```

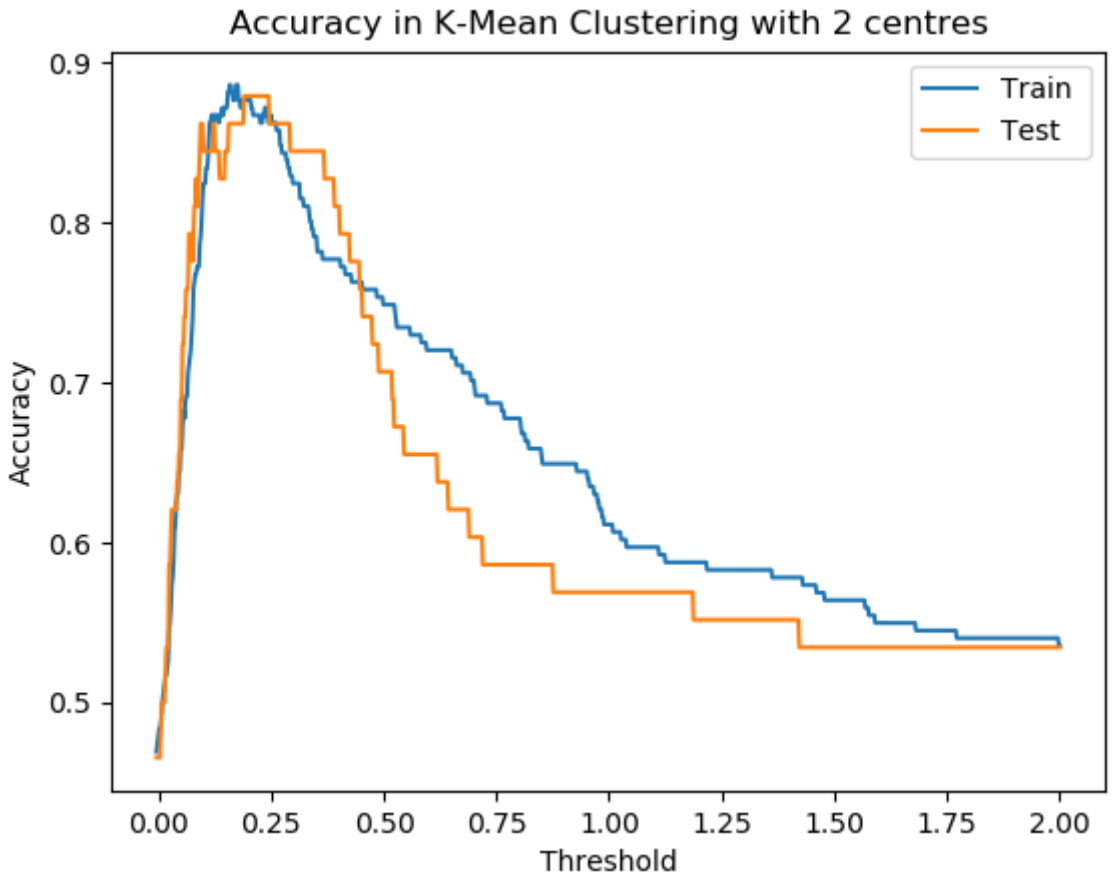



Figure 6. Performance of RBFN in K-Mean Clustering with 2 centres

Figure 6 shows the accuracy on both training set and testing set, and both accuracies can reach around 88%. Moreover, K-Mean Clustering compute faster as only two centres are selected.

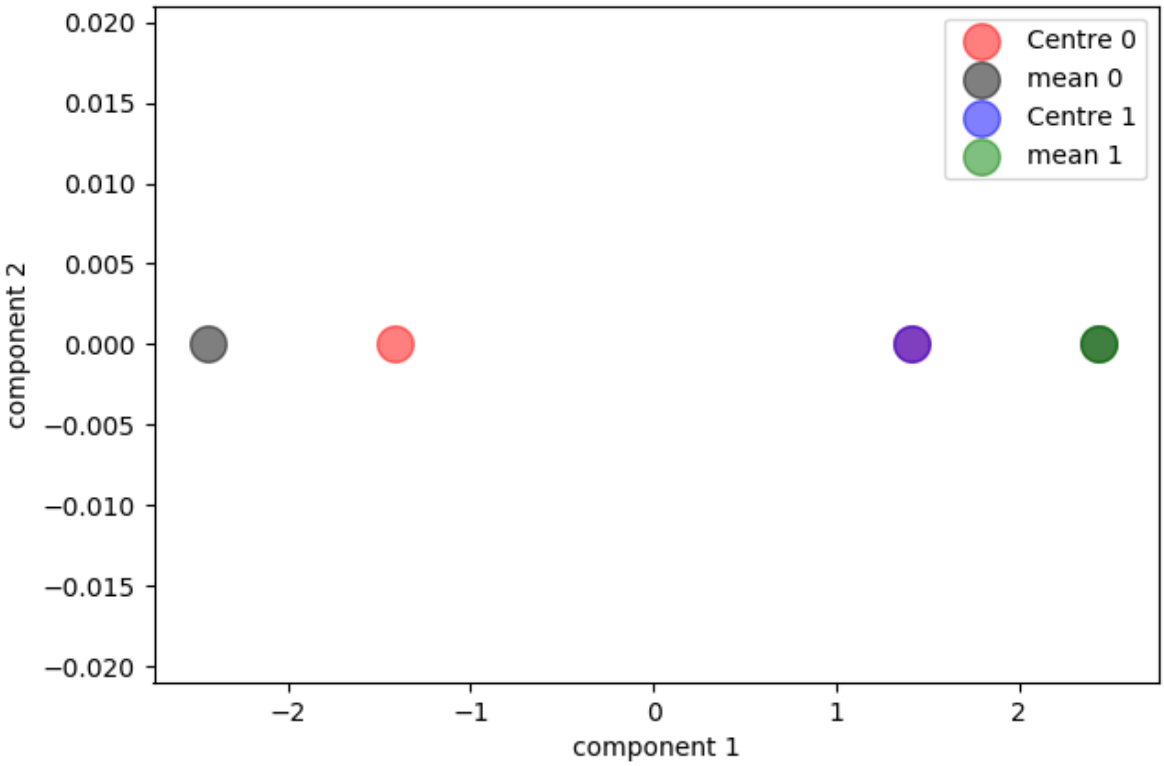



Figure 7. Clustering Centres and Class Mean Centres

Figure 7 shows the Clustering Centres and Class Mean Centres through PCA dimension reduction method in 2 Dimension.

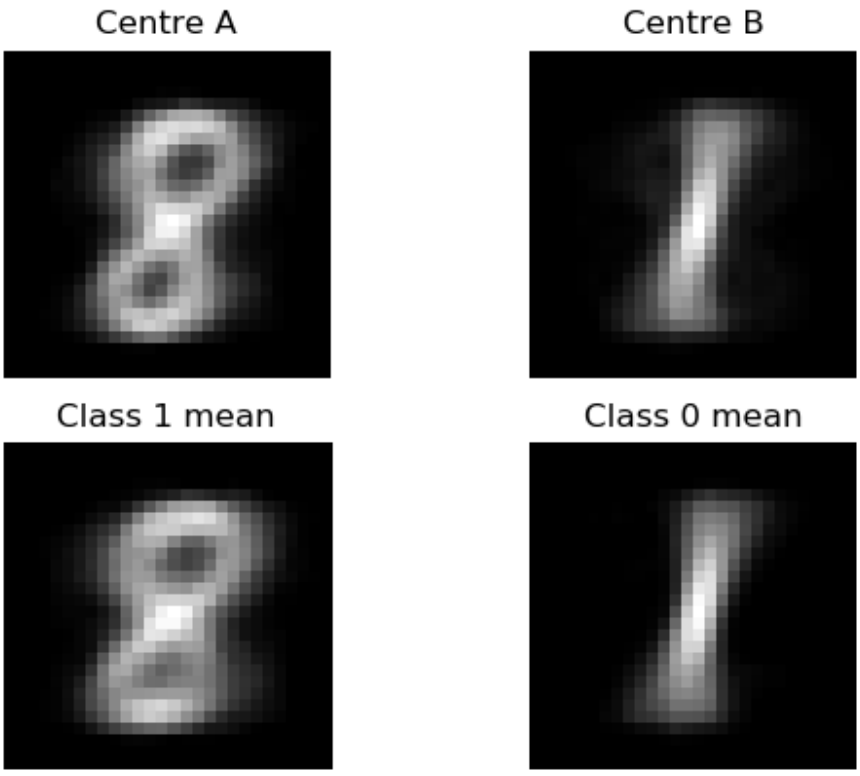



Figure 8. Visualization of obtained centres and mean centres

Figure 8 shows the images of obtained centres and mean centres. Centre A is similar to Class 1 mean, Centre B is similar to Class 0 mean. We can find that the visualization of the obtained centres and the mean of training images of each class are almost the same. In the k-mean clustering, the centres keep updated as the mean centre of each class. The iteration stops when members of the two classes are not changing anymore.  That's why the projected centre images and mean centres images are almost the same.

**Q3 Self-Organizing Map (SOM)**

a.) maps a 1-dimensional output layer of 25 neurons to a "heart curve"

```
import numpy as np
import random
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt

np.random.seed(6)
def initOutputLayer(m,n):  # m-number of neurons, n-dimension of feature
    layer = np.random.rand(m,n)
    return layer

# find min dist and return winer
def winnerIndex(x,y):
    a = np.array(x).reshape(1,y.shape[1])
    d = cdist(a,y,metric='euclidean')
    idx = np.argmin(d)
    return idx

def adjustWeight(layer,x,iter):
    time_ew = iter /np.log(sigma_init)
    w = layer
    for i in range(iteration):
        for j in range(x.shape[0]):
            #winner index
            idx = winnerIndex(x[j],w)
            # update learning rate
            LR = learn_rate*np.exp(-(i+1)/iter)
            # update effective width
            sigma = sigma_init*np.exp(-(i+1)/time_ew)

            # time varying neibourhood func
            N_index = np.arange(m).reshape(m,1)
            h = np.exp(-(N_index - idx)**2/(2*sigma**2))
            alpha = LR*h
            # update weights
            w = (1-alpha)*w + alpha*x[j]
    return w


m = 25
n = 2
learn_rate = 0.1
iteration = 1000
sigma_init = 1 * m
t = np.linspace(-np.pi,np.pi,200)

trainX = np.array((t*np.sin(np.pi*np.sin(t)/t),1-
abs(t)*np.cos(np.pi*np.sin(t)/t))).T

weights = adjustWeight(initOutputLayer(25,2),trainX,iteration)

```

```
50.
51. plt.plot(trainX[:,0],trainX[:,1],)
52. plt.plot(weights[:,0],weights[:,1])
53. plt.title('Performance of SOM')
54. plt.legend(('Samples', 'Weights'))
55. plt.show()
```

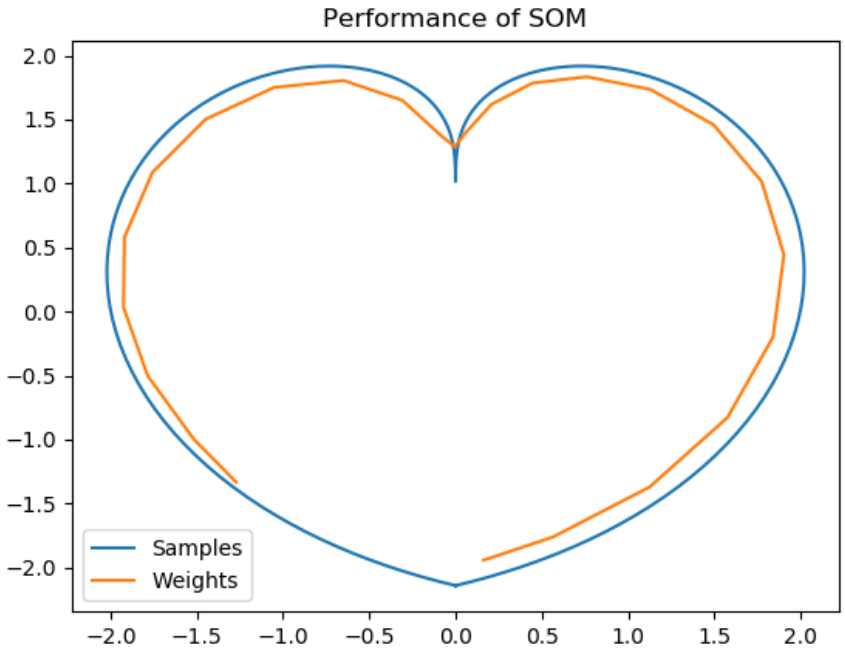



Figure 3-a-1 SOM Mapping for Heart Curve function

Figure 3-a-1 shows the SOM Mapping for Heart Curve function, and we can see that most of the weights fit to the desired output.

b) **a SOM that maps a 2-dimensional output layer of 25 neurons to a "square"** Source code in Python:

```
1.  import numpy as np
2.  import random
3.  from scipy.spatial.distance import cdist
4.  import matplotlib.pyplot as plt
5.  np.random.seed(12)
6.
7.  def init_grid(h,w):
8.      k = 0
9.      grid = np.zeros((h*w,2))
10.     for i in range(h):
11.         for j in range(w):
12.             grid[k,:] = [i,j]
13.             k += 1
14.     return grid
15.
16. def init_weights(h,w,n):  # m-number of neurons, n-dimension of feature
17.     weights = np.random.rand(h*w,n)
18.     return weights
19.
20. # find min dist and return winer
21. def winnerIndex(x,y):
```

```
    a = np.array(x).reshape(1,y.shape[1])
    d = cdist(a,y,metric='euclidean')
    idx = np.argmin(d)
    return idx

def distEclud(x, weights):
    dist = []
    for w in weights:
        d = np.linalg.norm(x-w)
        dist.append(d)
    return np.array(dist)

def adjustWeight(w,x,iter):
    time_ew = iter /np.log(sigma_init)

    for i in range(iteration):
        for j in range(x.shape[0]):
            # winner index
            idx = winnerIndex(x[j],w)
            a = np.array(grid[idx]).reshape(1,grid.shape[1])
            d = cdist(a,grid,metric='euclidean')

            # index2 = (dist2 ; 1).nonzero()[0]
            # update learning rate
            LR = learn_rate*np.exp(-(i+1)/iter)
            # update effective width
            sigma = sigma_init*np.exp(-(i+1)/time_ew)

            # time varying neibourhood func
            h1 = np.exp(-(d**2)/(2*sigma**2)).reshape(25,1)
            alpha = LR*h1
            # update weights
            w = (1-alpha)*w + alpha*x[j]
    return w

w = 5
h = 5
m = w*h
n = 2
learn_rate = 0.1
iteration = 1000
sigma_init = np.sqrt(w**2+h**2)/2
grid = init_grid(h,w)
weights = init_weights(h,w,n)

trainX = -1 + 2*np.random.random((500,2))
w = adjustWeight(weights,trainX,iteration)


def plot_weights(weights, h=5, w=5, size=(6,6)):
    x_axis = weights[:,0].reshape(h, w)
    y_axis = weights[:,1].reshape(h, w)
    plt.figure(figsize=size)
    plt.title('Training Points and Weights')
    plt.scatter(trainX[:,0],trainX[:,1],c='blue',label = 'Samples')
    plt.scatter(weights[:,0],weights[:,1],marker = 'x',c='red',label = 'Weights')
    plt.legend(loc ='upper right')
    for i in range(h):
        plt.plot(x_axis[i], y_axis[i],c='red')
        plt.plot(x_axis.T[i], y_axis.T[i],c='red')
    plt.show()

plot_weights(w,5,5,size=(6,6))
```

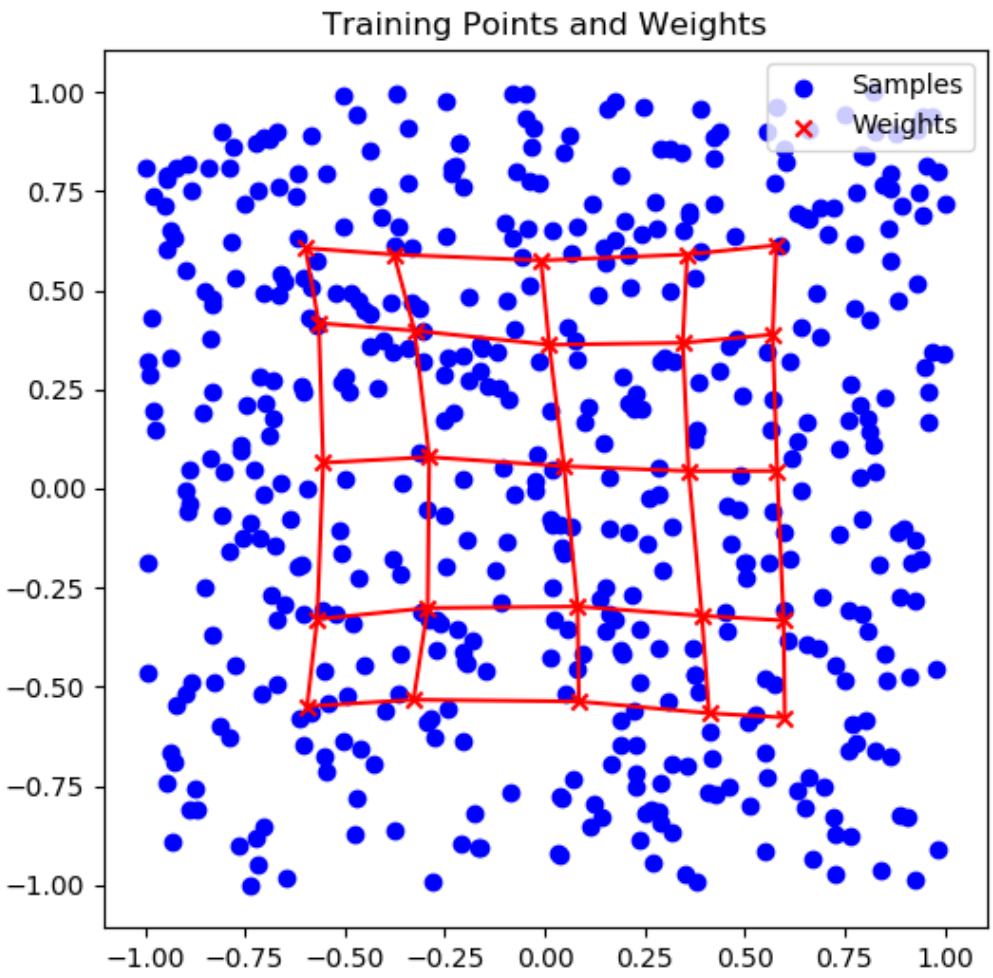



Figure 3-b-1 SOM square mapping

c-1) **Implement Cluster and Classify Handwritten Characters**
My Metrix Number is A0117981X. I ignore class 8 and 1.
Source Code in Python:

```
import numpy as np
import random
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from scipy import io

# np.random.seed(12)
# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train = mat['train_data'].T           # shape (1000,784)
y_train = mat['train_classlabel'].T     # shape (1000,1)
x_test = mat['test_data'].T             # shape (250,784)
y_test = mat['test_classlabel'].T       # shape (250,1)

# mark labels accordingly, my matrix No. A0117981X, mask digits 8 and 1
x_train = x_train[np.where((y_train !=8) & (y_train !=1))[0]]

def init_grid(h,w):
    k = 0
    grid = np.zeros((h*w,2))
    for i in range(h):
        for j in range(w):
```

```
            grid[k,:] = [i,j]
            k += 1
    return grid

def init_weights(h,w,n):  # m-number of neurons, n-dimension of feature
    weights = np.random.rand(h*w,n)
    return weights

# find min dist and return winer
def winnerIndex(x,y):
    a = np.array(x).reshape(1,y.shape[1])
    d = cdist(a,y,metric='euclidean')
    idx = np.argmin(d)
    return idx

def adjustWeight(w,x,iter):
    time_ew = iter /np.log(sigma_init)

    for i in range(iteration):
        for j in range(x.shape[0]):
            # winner index
            idx = winnerIndex(x[j],w)
            a = np.array(grid[idx]).reshape(1,grid.shape[1])
            d = cdist(a,grid,metric='euclidean')

            # update learning rate
            LR = learn_rate*np.exp(-(i+1)/iter)
            # update effective width
            sigma = sigma_init*np.exp(-(i+1)/time_ew)

            # time varying neibourhood func

            h1 = np.exp(-(d**2)/(2*sigma**2)).reshape(100,1)
            alpha = LR*h1
            # update weights
            w = (1-alpha)*w + alpha*x[j]
    return w


w = 10
h = 10
m = w*h
n = 28*28
learn_rate = 0.1
iteration = 10
sigma_init = np.sqrt(w**2+h**2)/2
grid = init_grid(h,w)
weights = init_weights(h,w,n)

updated_w = adjustWeight(weights,x_train,iteration)

#conceptual
plt.figure()
for i in range(100):
    index = winnerIndex(updated_w[i],x_train)
    plt.subplot(10,10,i+1)
    plt.imshow(x_train[index].reshape(28,28).T,cmap='gray')
    plt.axis('off')
plt.show()

##Visualization of Weight
plt.figure()
for j in range(100):

    plt.subplot(10,10,j+1)
    plt.imshow(updated_w[j].reshape(28,28).T,cmap='gray')
```

```
89.     plt.axis('off')
90. plt.show()
```

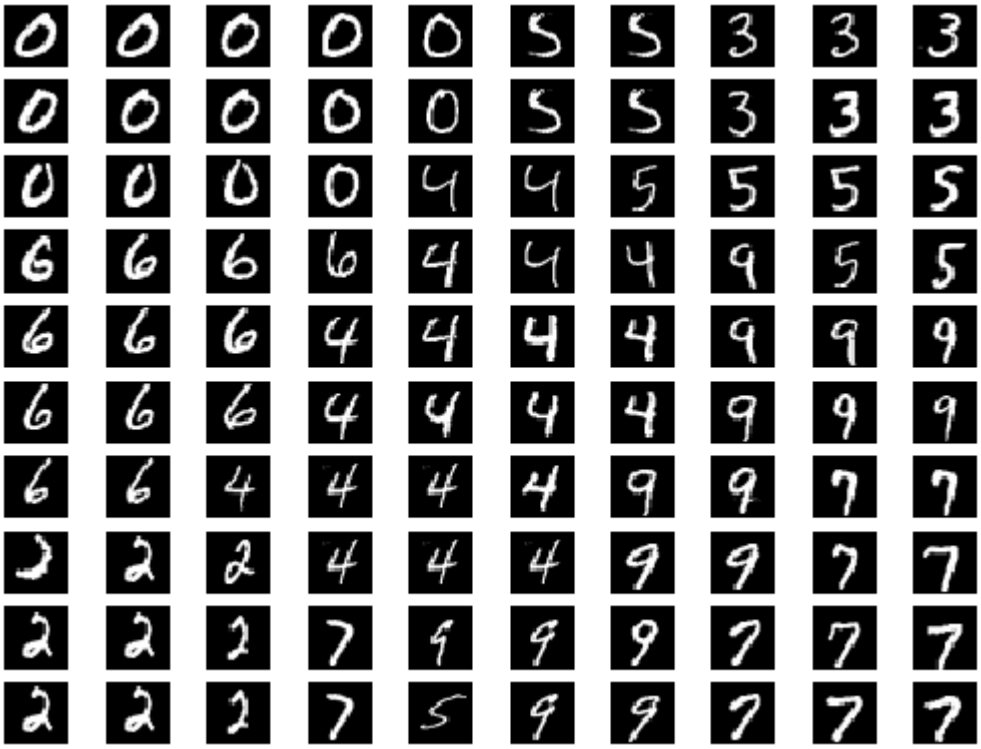


Figure 3-c-1 Conceptual/Semantic Map

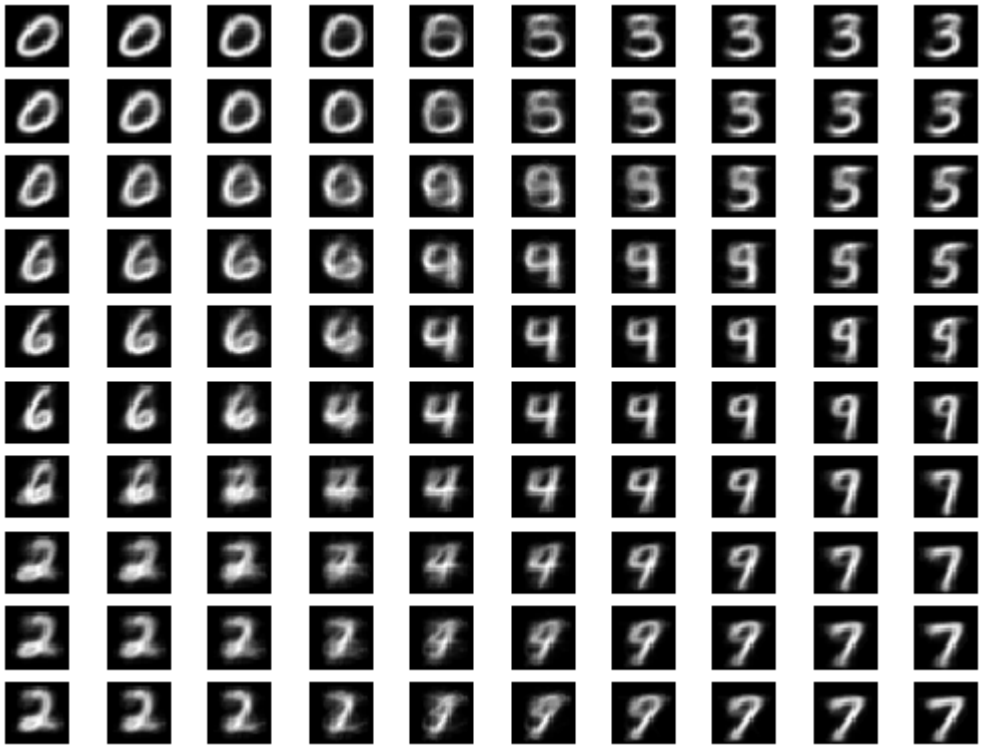


Figure 3-c-2 Visualization of weights

Figure 3-c-1 shows corresponding conceptual/semantic map of the trained SOM. and Figure 3-c-2 shows the trained weights of each output neuron on a 10×10 map. Compare these two figures, we can find that the images are matched on the same on the same position. This is because the conceptual/semantic map consists of winner input images corresponding to each weight.

### c-2) Classification accuracy

Source Code in Python:

```
import numpy as np
import random
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from scipy import io


# Load Data
mat = io.loadmat('../MNIST_database.mat')
x_train = mat['train_data'].T           # shape (1000,784)
y_train = mat['train_classlabel'].T     # shape (1000,1)
x_test = mat['test_data'].T             # shape (250,784)
y_test = mat['test_classlabel'].T       # shape (250,1)

# mark labels accordingly, my matrix No. A0117981X, mask digits 8 and 1
idx_train = np.where((y_train != 8) & (y_train != 1))[0]
idx_test = np.where((y_test !=8) & (y_test !=1))[0]
y_train = y_train[idx_train]
x_train = x_train[idx_train]

x_test = x_test[idx_test]
y_test = y_test[idx_test]

x_test = x_test[np.where((y_test !=8) & (y_test !=1))[0]]

def init_grid(h,w):
    k = 0
    grid = np.zeros((h*w,2))
    for i in range(h):
        for j in range(w):
            grid[k,:] = [i,j]
            k += 1
    return grid

np.random.seed(6)
def init_weights(h,w,n):  # m-number of neurons, n-dimension of feature
    weights = np.random.rand(h*w,n)
    return weights

# find min dist and return winer
def winnerIndex(x,y):
    a = np.array(x).reshape(1,y.shape[1])
    d = cdist(a,y,metric='euclidean')
    idx = np.argmin(d)
    return idx

def adjustWeight(w,x,iter):
    time_ew = iter /np.log(sigma_init)
    for i in range(iteration):
        for j in range(x.shape[0]):
            # winner index
            idx = winnerIndex(x[j],w)
            a = np.array(grid[idx]).reshape(1,grid.shape[1])
            d = cdist(a,grid,metric='euclidean')
```

```
55.
56.             # update learning rate
57.             LR = learn_rate*np.exp(-(i+1)/iter)
58.             # update effective width
59.             sigma = sigma_init*np.exp(-(i+1)/time_ew)
60.
61.             # time varying neibourhood func
62.             h1 = np.exp(-(d**2)/(2*sigma**2)).reshape(100,1)
63.             alpha = LR*h1
64.             # update weights
65.             w = (1-alpha)*w + alpha*x[j]
66.     return w
67.
68. w = 10
69. h = 10
70. m = w*h
71. n = 28*28
72. learn_rate = 0.1
73. iteration = 10
74. sigma_init = np.sqrt(w**2+h**2)/2
75. grid = init_grid(h,w)
76. weights = init_weights(h,w,n)
77. updated_w = adjustWeight(weights,x_train,iteration)
78.
79.
80. W_labels = []
81. for i in range(100):
82.     index = winnerIndex(updated_w[i],x_train)
83.     W_labels.append(y_train[index][0])
84.
85. test_Labels =[]
86. for j in range(x_test.shape[0]):
87.     index = winnerIndex(x_test[j],updated_w)
88.     test_Labels.append(W_labels[index])
89.
90. error =0
91. for k in range(x_test.shape[0]):
92.     if test_Labels[k] != y_test[k][0]:
93.         error +=1
94. # print(error)
95. acc = 1-error/x_test.shape[0]
96. print('The classification accuracy of testing set is %.2f' %(acc*100) +'%')
```

The final classification accuracy is 75% after 100 epochs of training which is slightly lower than the results obtained in RBFN model. This is because that some images may be assigned to their adjacent class groups since they have similar features.