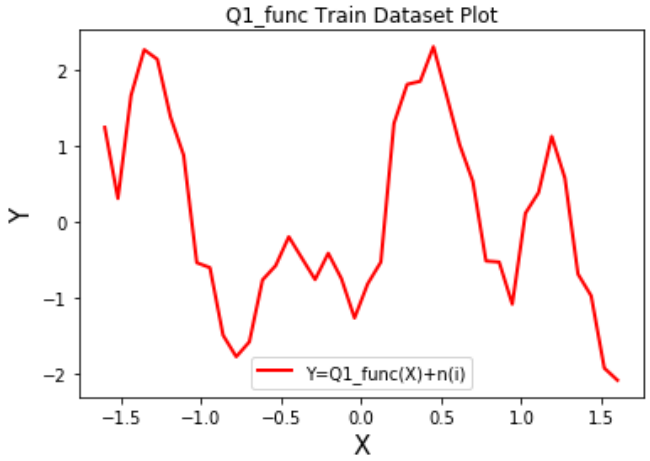
Image Classification: Radial Basis Functions & Self Organizing Maps

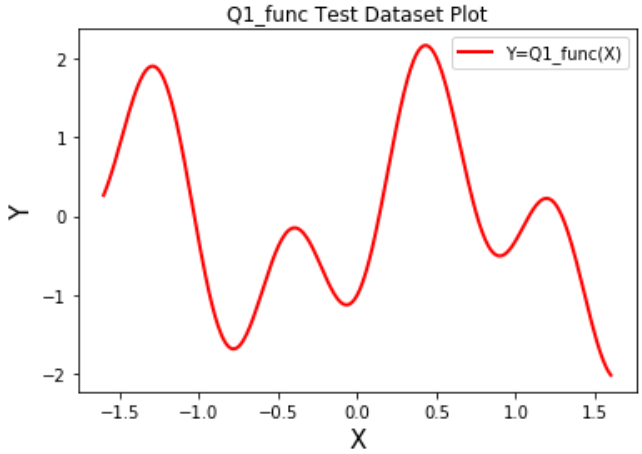
SAMYUEL DANYO, 03/2019

Introduction

Image classification on MNIST Hand-written Digits Dataset and regression via full custom Radial Basis Functions (Exact Interpolation + Fixed Centers Selected at Random Method + Regularization) and Self Organizing Maps.

Q1:





1(a): My Radial Basis Function with Exact Interpolation implementation can be seen below:

```
def gaussian(X, Xi, sd):
    return np.exp(-np.sum((X - Xi)**2, axis=1)/(2*sd**2))

class Layer_RBFN(object):
    """Base class for the different layers.
    Defines base methods and documentation of methods."""

    def get_output(self):
        """Peroforms a forward step to return the output."""
        return []

class GaussianInterpolationLayer_RBFN_BASIC(Layer_RBFN):
    """The hidden layer performs a non-linear transformation to its input."""

    def __init__(self, n_in, n_out):
        """Initialize hidden layer centers.
        n_in is the number of input variables (features per sample).
        n_out is the number of hidden neurons (number of trianing samples).
        sd is the Gaussian Standard Deviation."""
        self.C = np.full((n_out, n_in), 0)
        self.sd = 1

    def train(self, X, sd):
        """Initialize the classifier fields with the training dataset."""
        self.C = X
        self.sd = sd

    def get_params_array(self):
        """Return array of the centers & the sd."""
        return np.array(self.C), np.asscalar(self.sd)

    def get_output(self, X):
        """Perform the forward step transformation."""
        return np.array([gaussian(x.reshape(1,X.shape[1]), self.C, self.sd) for x in X])

class LinearLayer_RBFN_BASIC(Layer_RBFN):
    """The linear layer performs a linear transformation to its input."""

    def __init__(self, N):
        """Initialize hidden layer parameters.
        n_in is the number of input variables (hidden nodes)."""
        self.W = np.random.randn(N, 1) * 0.1
```

```
    def train(self, X, T, L2_rate = 1, regularization = False):
        if regularization:
            I = np.diag(np.full((self.W.shape[0]), 1))
            self.W = np.linalg.inv(X + L2_rate*I).dot(T)
        else:
            self.W = np.linalg.inv(X).dot(T)

    def get_params_array(self):
        """Return arrays of the parameters."""
        return np.array(self.W)

    def get_output(self, X):
        """Perform the forward step linear transformation."""
        if(np.isscalar(X) or X.shape == (1,)):
            return X*self.W
        else:
            return X.dot(self.W)

    def get_cost(self, Y, T):
        return (Y-T).sum()/Y.shape[0]

# Define the forward propagation step as a method.
def forward_step_RBFN(input_samples, layers):
    """
    Compute and return the forward activation of each layer in layers.
    Input:
        input_samples: A matrix of input samples (each row is an input vector)
        layers: A list of Layers
    Output:
        A list of activations where the activation at each index i+1 corresponds to
        the activation of layer i in layers. activations[0] contains the input samples.
    """
    activations = [input_samples] # List of layer activations
    # Compute the forward activations for each layer starting from the first
    X = input_samples
    for layer in layers:
        Y = layer.get_output(X)  # Get the output of the current layer
        activations.append(Y)  # Store the output for future processing
        X = activations[-1]  # Set the current input as the activations of the previous layer
    return activations  # Return the activations of each layer

def train_network_RBFN_BASIC(layers, X_Train, T_Train):
    start_time = time.time()
    print("Starting Training")
    layers[0].train(X_Train, SD)
    Y = layers[0].get_output(X_Train)
    layers[1].train(Y, T_Train, L2_rate, regularization)
    activations = forward_step_RBFN(X_Train, layers)  # Get the activations
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Train)
    end_time = time.time()
    t_time = end_time - start_time
    return layers, cost, t_time, Y

def plot_RBFN(cost, t_time):
    #Print time for training
    m, s = divmod(t_time, 60)
    h, m = divmod(m, 60)
    print("OVERALL TIME FOR TRAINING: {}h:{}m:{:.5f}s".format(h,m,s))
    print("TRAINING COST: {:.5f}".format(cost))

def test_network_RBFN(layers, X_Test, T_Test):
    # Get results of test data
    activations = forward_step_RBFN(X_Test, layers)  # Get activation of test samples
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Test)
    print("TEST COST: {:.5f}".format(cost))
    return Y
```

```
>>>>>> RBFN_Q1A Architecture <<<<<<
INPUT LAYER NODES: 1
```

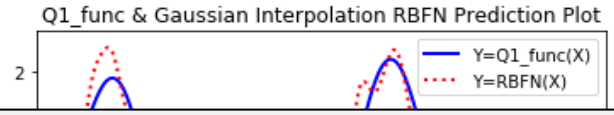
HIDDEN LAYER (Gaussian Interpolation) NODES: 160

```
# TRAIN
regularization = False
L2_rate = 1e-10
SD = 0.1  # Standard Deviation of the Gaussian Finction

RBFN_Q1A, cost, time_t, Y_Train_Q1A = train_network_RBFN_BASIC(RBFN_Q1A, X_Train_Q1, T_Train_Q1)
plot_RBFN(cost, time_t)
```

OUTPUT LAYER (Lin) NODES: 1

The performance of the RBF with exact interpolation on the noisy dataset is not very good but in line with expectations. This can be seen on the plot, displaying the test dataset predictions of the algorithm. The test cost was **0.02158.** The areas with bad performance were to be expected as exact interpolation aims to find a "perfect" fit. The approximated function needs to pass through all training points, which makes it extremely vulnerable to noise (oscillatory). The noisy centers can be easily noticed on the plot.

```
TEST COST: 0.02158
```



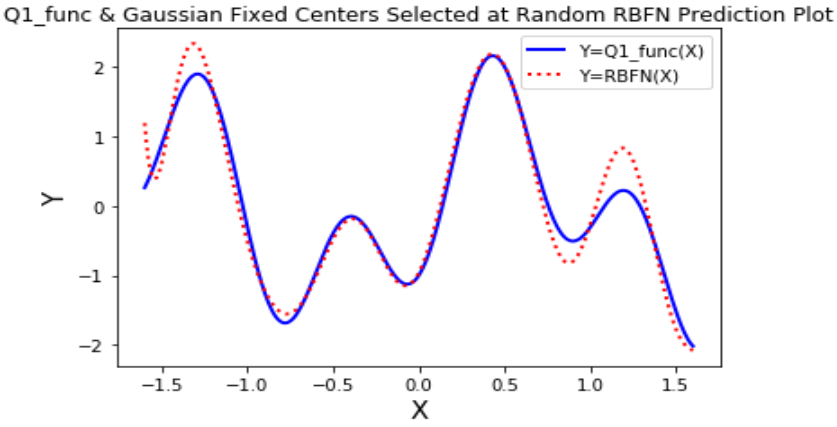
1(b): My RBF with Fixed Centers at Random implementation can be seen below :

```
def gaussian(X, Xi, sd):
    return np.exp(-np.sum((X - Xi)**2, axis=1)/(2*sd**2))

def euclidean_distance(X1, X2):
        """Calculate the Euclidean distances between X1 and X2."""
        return np.sqrt(np.absolute(-2*np.dot(X1, X2.T) + np.sum(X2**2,axis=1) + np.sum(X1**2,
axis=1)[:, np.newaxis]))

class Layer_RBFN(object):
    """Base class for the different layers.
    Defines base methods and documentation of methods."""

    def get_output(self):
        """Peroforms a forward step to return the output."""
        return []

class GaussianInterpolationLayer_RBFN_FCSR(Layer_RBFN):
    """The hidden layer performs a non-linear transformation to its input."""

    def __init__(self, n_in, n_out):
        """Initialize hidden layer centers.
        n_in is the number of input variables (features per sample).
        n_out is the number of hidden neurons (number of trianing samples).
        sd is the Gaussian Standard Deviation."""
        self.C = np.full((n_out, n_in), 0)
        self.sd = 1

    def train(self, X, M):
        """Initialize the classifier fields M randomly selected centers from the training
dataset."""
        idx = np.random.randint(X.shape[0], size=M)
        self.C = X[idx,:]
        self.sd = np.max(euclidean_distance(self.C, self.C))/np.sqrt(2*M)

    def get_params_array(self):
        """Return array of the centers & the sd."""
        return np.array(self.C), np.asscalar(self.sd)

    def get_output(self, X):
        """Perform the forward step transformation."""
        return np.array([gaussian(x.reshape(1,X.shape[1]), self.C, self.sd) for x in X])
```

```python
class LinearLayer_RBFN_FCSR(Layer_RBFN):
    """The linear layer performs a linear transformation to its input."""

    def __init__(self, N):
        """Initialize hidden layer parameters.
        n_in is the number of input variables (hidden nodes)."""
        self.W = np.random.randn(N, 1) * 0.1
        self.b = 0

    def train(self, X, T, L2_rate = 1, regularization = False):
        bias_inputs = np.full((X.shape[0]), 1)
        X = np.column_stack((bias_inputs, X))
        if regularization:
            I = np.diag(np.concatenate([np.array([1]), np.full((self.W.shape[0]), 1)]))
            self.W = np.linalg.inv(X.T.dot(X) + L2_rate*I).dot(X.T).dot(T)[1:]
            self.b = np.linalg.inv(X.T.dot(X) + L2_rate*I).dot(X.T).dot(T)[0]
        else:
            self.W = np.linalg.pinv(X).dot(T)[1:]
            self.b = np.linalg.pinv(X).dot(T)[0]

    def get_params_array(self):
        """Return arrays of the parameters."""
        return np.array(self.W), np.array(self.b)

    def get_output(self, X):
        """Perform the forward step linear transformation."""
        if(np.isscalar(X) or X.shape == (1,)):
            return X*self.W + self.b
        else:
            return X.dot(self.W) + self.b

    def get_cost(self, Y, T):
        return (Y-T).sum()/Y.shape[0]

# Define the forward propagation step as a method.
def forward_step_FCSR(input_samples, layers):
    """
    Compute and return the forward activation of each layer in layers."""
    activations = [input_samples] # List of layer activations
    X = input_samples
    for layer in layers:
        Y = layer.get_output(X)  # Get the output of the current layer
        activations.append(Y)  # Store the output for future processing
        X = activations[-1]  # Set the current input as the activations of the previous layer
    return activations  # Return the activations of each layer

def train_network_RBFN_FCSR(layers, X_Train, T_Train):
    start_time = time.time()
    print("Starting Training")
    layers[0].train(X_Train, HIDDEN_M)
    Y = layers[0].get_output(X_Train)
    layers[1].train(Y, T_Train, L2_rate, regularization)
    activations = forward_step_RBFN(X_Train, layers)  # Get the activations
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Train)
    end_time = time.time()
    t_time = end_time - start_time
    return layers, cost, t_time, Y

def plot_RBFN(cost, t_time):
    #Print time for training
    m, s = divmod(t_time, 60)
    h, m = divmod(m, 60)
    print("OVERALL TIME FOR TRAINING: {}h:{}m:{:.5f}s".format(h,m,s))
    print("TRAINING COST: {:.5f}".format(cost))

def test_network_RBFN(layers, X_Test, T_Test):
    # Get results of test data
    activations = forward_step_RBFN(X_Test, layers)  # Get activation of test samples
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Test)
    print("TEST COST: {:.5f}".format(cost))
    return Y
```

```
>>>>>> RBFN_Q1B Architecture <<<<<<
INPUT LAYER NODES: 1
HIDDEN LAYER (Gaussian Fixed Centers) NODES: 20
OUTPUT LAYER (Lin) NODES: 1
```

TEST COST: 0.02297



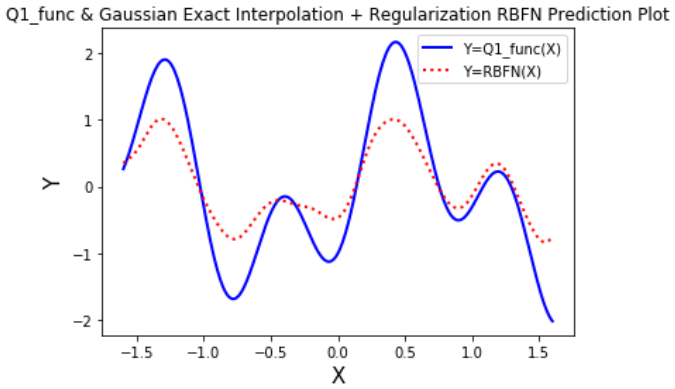
As can be seen by the test dataset prediction plot to the left and confirmed by the test cost of **0.02297**, the fixed centers selected at random perform extremely well. This is due to the reduced vulnerability towards noise, since there are only 20 centers, which can fit the function much looser and so map a smoother function. Besides saving computing power, the smaller amount of weights is also less prone any outliers or spikes in noise contamination.

1(c):

```
>>>>>> RBFN_Q1C Architecture <<<<<<
INPUT LAYER NODES: 1
HIDDEN LAYER (Exact Gaussian Interpolation) NODES: 160
OUTPUT LAYER (Lin) NODES: 1
```

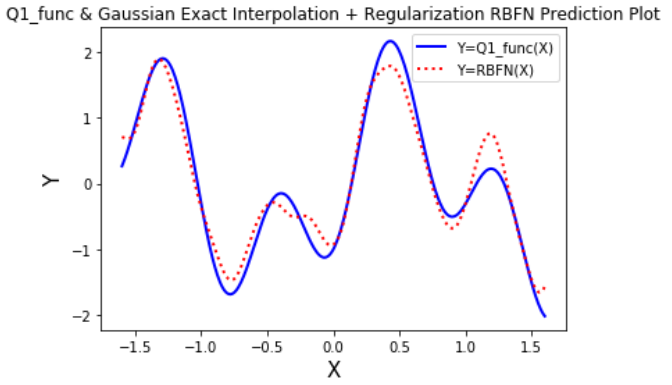
```
# TRAIN
regularization = True
L2_rate = 3e-0
```

TEST COST: -0.00150



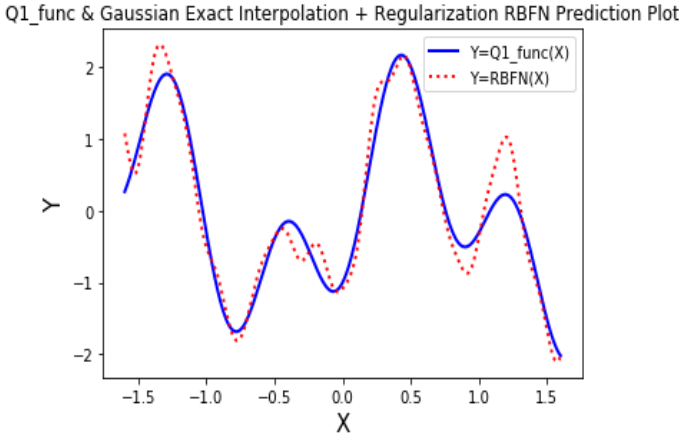

```
# TRAIN
regularization = True
L2_rate = 5e-1
```

TEST COST: 0.02056



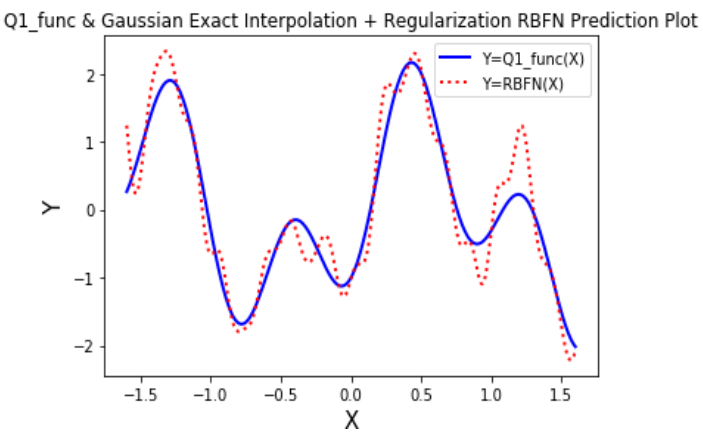

```
# TRAIN
regularization = True
L2_rate = 5e-2
```

TEST COST: 0.02483




```
# TRAIN
regularization = True
L2_rate = 5e-5
```

TEST COST: 0.02159



From the plots above, it can be seen that with declining regularization rate, the algorithm is let to overfit to the noisy training data. The effect of noise distortions and oscillations start appearing to the point where at regularization rate of `5e-5`, the model completely overfits. This show how important it is to fine-tune the regularization rate for best trade-off between bias and variance, in order to get the best generalization.

Q2:

```
>>>>>> MNIST DATASET FEATURE SPACE <<<<<<
>>> FULL <<<      >>> TEST <<<
(1250, 785)       (250, 784)
>>> TRAIN <<<     (250,)
(1000, 784)       >>>>>> BINARY MNIST DATASET LABELS <<<<<<
(1000,)
```

The class 0 samples (digits 0-4,6,8,9) are 4 times more than class 1 (digits 5,7). I expect this imbalance to distort and mute the overall performance of the models, due to a bias towards class 0.

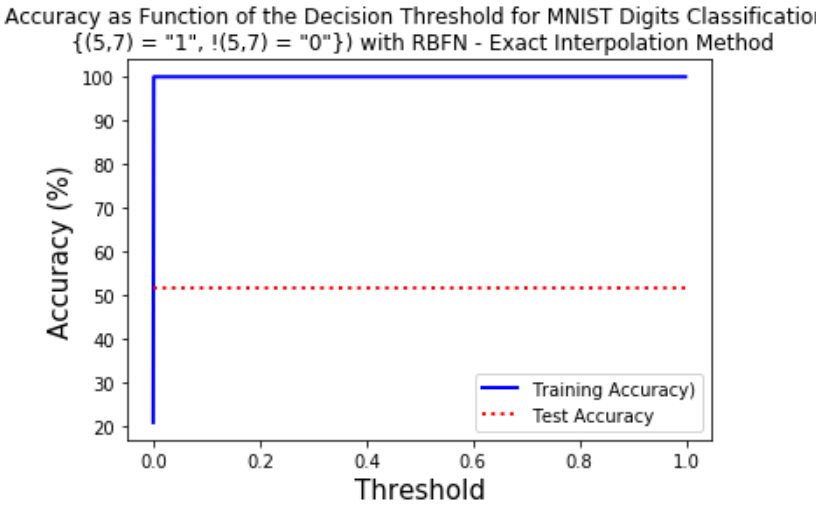
I am using the RBF implementations, presented in Q1.

2(a):

```
>>>>>> RBFN_Q2A Exact Interpolation Architecture <<<<<<
INPUT LAYER NODES: 784
HIDDEN LAYER (Gaussian Interpolation) NODES: 1000
OUTPUT LAYER (Lin) NODES: 1
```

```
# TRAIN
regularization = False
```

```
TEST COST: 19.19506
```

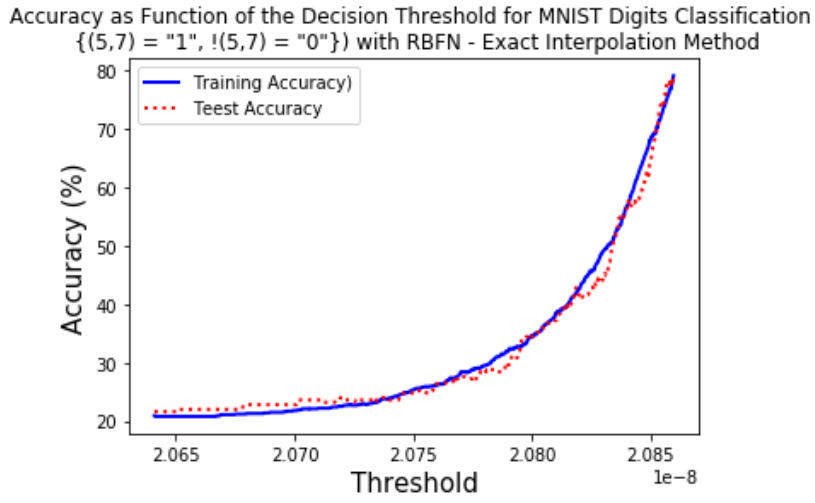



```
OPTIMAL THRESHOLD: ~ 0.0
TRAIN ACCURACY: 100.0%
TEST ACCURACY: 50.00%
```

With no regularization, the exact interpolation technique fits all training samples, resulting in 100% training accuracy. The test accuracy is also flat at 50.0% as result of the overfitting.

```
# TRAIN
regularization = True
L2_rate = 1e+10
```

```
TEST COST: -0.21600
```



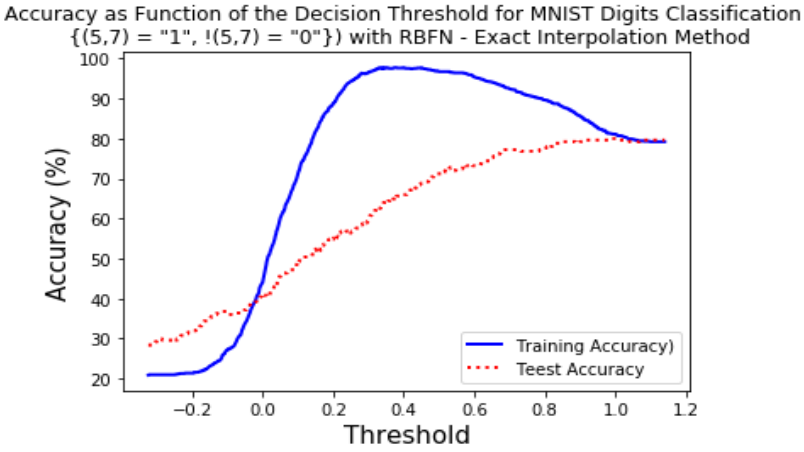

```
OPTIMAL THRESHOLD: ~ 0.572
TRAIN ACCURACY: 82.20%
TEST ACCURACY: 80.40%
TEST ERROR RATE: 21.60%
TEST FALSE NEGATIVES ERROR: 21.6%
TEST FALSE POSITIVES ERROR: 0.00%
```

Very high regularization rate, forces the model to extremely underfit and to maintain the weights quite small. The effect of that, as seen by the error rates above, is that the model cannot distinguish between classes 1/0 and classifies all of them the same way – **1** at the bottom of the envelope and **0** at the top.

```
# TRAIN
regularization = True
L2_rate = 1e-5
```

```
OPTIMAL THRESHOLD: ~ 0.987
TRAIN ACCURACY: 81.20%
TEST ACCURACY: 80.00%
```

```
TEST COST: -0.03721
```



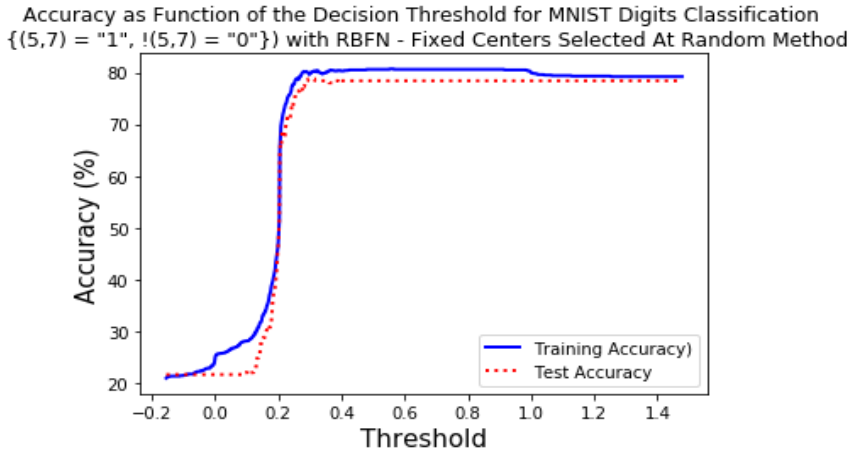
With a sensible regularization rate, the model is able to learn better. The results are still weak though as the model fits every training sample, resulting in overfitting and poor generalization.

2(b):

```
>>>>>> RBFN_Q2B Fixed Centers Selected at Random Architecture <<<<<<
INPUT LAYER NODES: 784
HIDDEN LAYER (Gaussian Interpolation) NODES: 100
OUTPUT LAYER (Lin) NODES: 1
```

```
# TRAIN
self.sd = np.max(euclidean_distance(self.C, self.C))/np.sqrt(2*M)
```

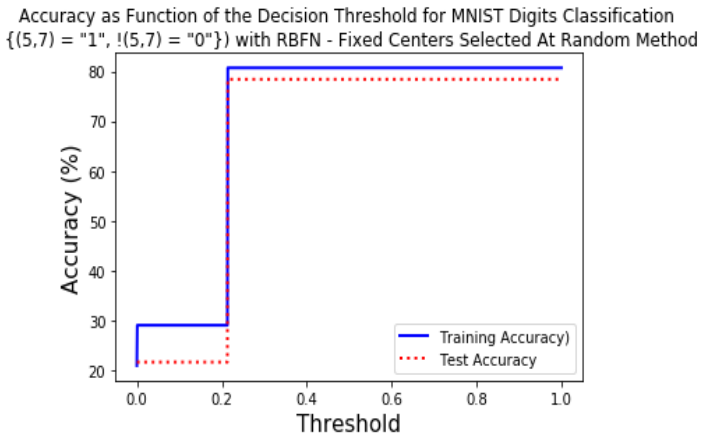
```
TEST COST: -0.00882
```



With the standard deviation being a function of the maximum Euclidean distances between the fixed centers & their number, the model fit looks like a regularized exact interpolation. The model does not overfit to the training data but the optimal performance still is not great.

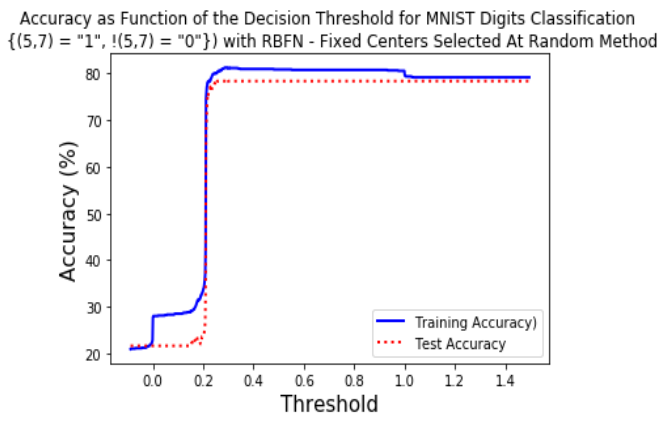
```
# TRAIN
self.sd = 0.1
```

```
TEST COST: -0.00227
```

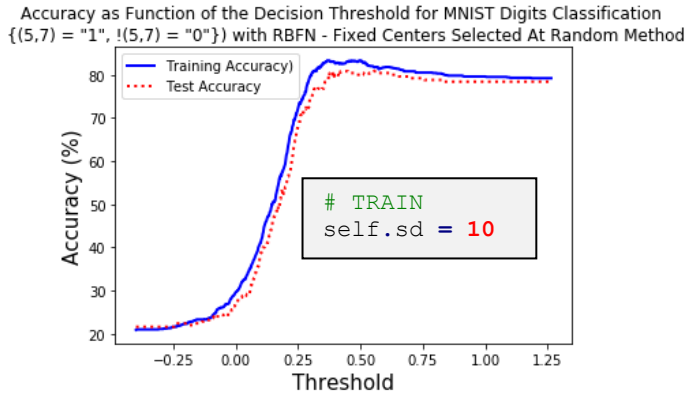



```
# TRAIN
self.sd = 1
```

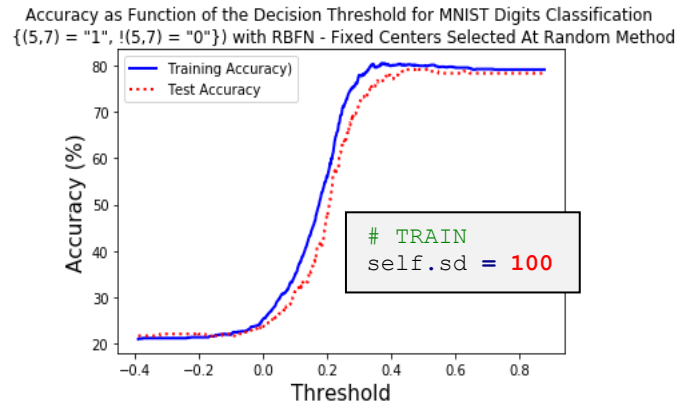
```
TEST COST: -0.00546
```



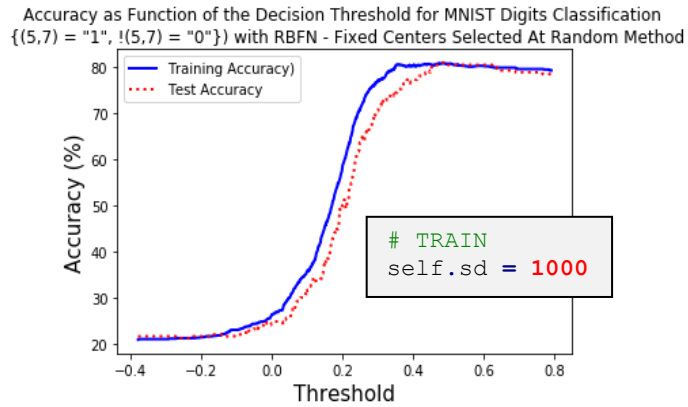
TEST COST: -0.00990



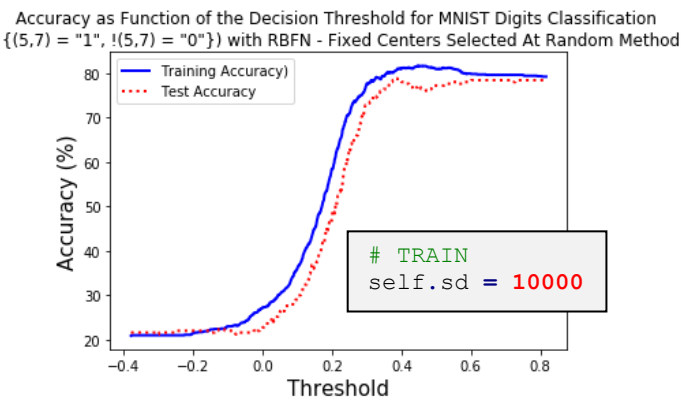
TEST COST: -0.01712



TEST COST: -0.00883



TEST COST: -0.00329



The standard deviation controls the distance of surrounding samples, which have effect on the prediction. With big SD the solution will be influenced by many samples and so might be too general (underfit) – we will have a flat RBF. Small SD means only the closest samples will influence the prediction, which might result in overfitting – peaked RBF. A balanced solution would usually be preferred, hence why using the maximum centers distance and their number usually works well. It can be seen that it falls somewhere between SD = 1 and SD = 10. It can also be seen that SD = 0.1, 10000 result in worse performance.

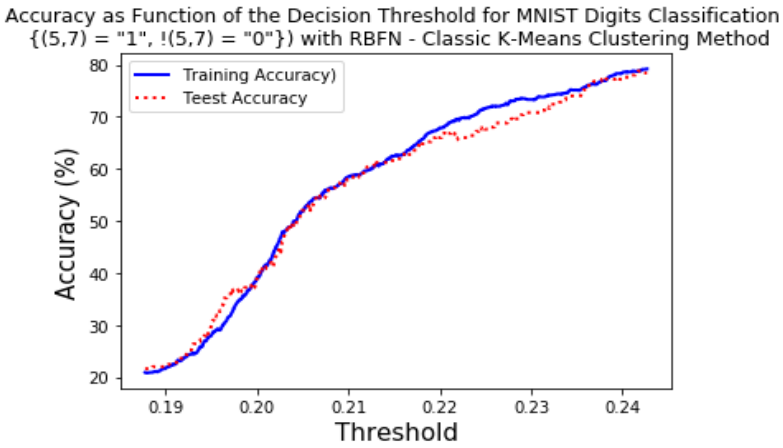
2(c): My K-Means Clustering RBF Implementation:

```
def gaussian(X, Xi, sd):
    return np.exp(-np.sum((X - Xi)**2, axis=1)/(2*sd**2))

def euclidean_distance(X1, X2):
        """Calculate the Euclidean distances between X1 and X2."""
        return np.sqrt(np.absolute(-2*np.dot(X1, X2.T) + np.sum(X2**2,axis=1) + np.sum(X1**2,
axis=1)[:, np.newaxis]))

class Layer_RBFN(object):
    """Base class for the different layers.
    Defines base methods and documentation of methods."""

    def get_output(self):
        """Peroforms a forward step to return the output."""
        return []

class GaussianInterpolationLayer_RBFN_KMC(Layer_RBFN):
    """The hidden layer performs a non-linear transformation to its input."""

    def __init__(self, n_in, n_out):
        """Initialize hidden layer centers.
        n_in is the number of input variables (features per sample).
        n_out is the number of hidden neurons (number of centers).
        sd is the Gaussian Standard Deviation."""
        self.C = np.random.randn(n_out, n_in) * 0.1
        self.sd = 1
```

```
    def train(self, X, M, max_nb_itrs = 1000):
        """ Compute the classifier centers through Classic K-Means Clustering."""
        for itr in range(0, max_nb_itrs):
            C_old = np.copy(self.C)
            dists = euclidean_distance(X, self.C)
            labels = np.argmin(dists, axis=1)
            for idx in range(0,M):
                tmp = np.copy(X)
                #Setting all samples from other classes to (0)
                tmp[labels != idx] = 0
                #Removing all samples from other classes
                tmp = tmp[~np.all(tmp == 0, axis=1)]
                #Calculating the new 'idx' center
                self.C[idx] = np.mean(tmp, axis=0)
            C_upd = np.absolute(np.sum(self.C - C_old))
            if itr%10 == 0:
                print("LAST CENTER UPDATE: {}".format(C_upd))
            if C_upd < 1e-3:
                break
        self.sd = np.max(euclidean_distance(self.C, self.C))/np.sqrt(2*M)

    def get_params_array(self):
        """Return array of the centers & the sd."""
        return np.array(self.C), np.asscalar(self.sd)

    def get_output(self, X):
        """Perform the forward step transformation."""
        return np.array([gaussian(x.reshape(1,X.shape[1]), self.C, self.sd) for x in X])

class LinearLayer_RBFN_KMC(Layer_RBFN):
    """The linear layer performs a linear transformation to its input."""

    def __init__(self, N):
        """Initialize hidden layer parameters.
        n_in is the number of input variables (hidden nodes)."""
        self.W = np.random.randn(N, 1) * 0.1
        self.b = 0

    def train(self, X, T, L2_rate = 1, regularization = False):
        bias_inputs = np.full((X.shape[0]), 1)
        X = np.column_stack((bias_inputs, X))
        if regularization:
            I = np.diag(np.concatenate([np.array([1]), np.full((self.W.shape[0]), 1)]))
            self.W = np.linalg.inv(X.T.dot(X) + L2_rate*I).dot(X.T).dot(T)[1:]
            self.b = np.linalg.inv(X.T.dot(X) + L2_rate*I).dot(X.T).dot(T)[0]
        else:
            self.W = np.linalg.pinv(X).dot(T)[1:]
            self.b = np.linalg.pinv(X).dot(T)[0]

    def get_params_array(self):
        """Return arrays of the parameters."""
        return np.array(self.W), np.array(self.b)

    def get_output(self, X):
        """Perform the forward step linear transformation."""
        if(np.isscalar(X) or X.shape == (1,)):
            return X*self.W + self.b
        else:
            return X.dot(self.W) + self.b

    def get_cost(self, Y, T):
        return (Y-T).sum()/Y.shape[0]

def forward_step_KMC(input_samples, layers):
    """
    Compute and return the forward activation of each layer in layers."""
    activations = [input_samples] # List of layer activations
    X = input_samples
    for layer in layers:
        Y = layer.get_output(X)  # Get the output of the current layer
        activations.append(Y)  # Store the output for future processing
        X = activations[-1]  # Set the current input as the activations of the previous layer
    return activations  # Return the activations of each layer
```

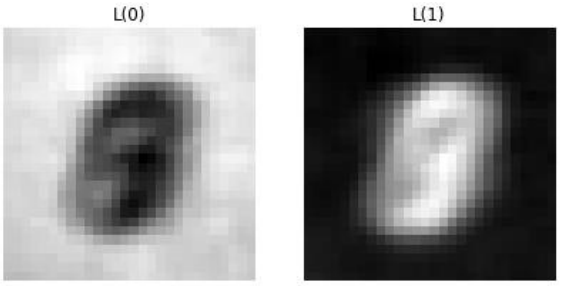
```python
def train_network_RBFN_KMC(layers, X_Train, T_Train):
    start_time = time.time()
    print("Starting Training")
    layers[0].train(X_Train, HIDDEN_M, MAX_NB_ITRS)
    Y = layers[0].get_output(X_Train)
    layers[1].train(Y, T_Train, L2_rate, regularization)
    activations = forward_step_RBFN(X_Train, layers)  # Get the activations
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Train)
    end_time = time.time()
    t_time = end_time - start_time
    return layers, cost, t_time, Y

def plot_RBFN(cost, t_time):
    #Print time for training
    m, s = divmod(t_time, 60)
    h, m = divmod(m, 60)
    print("OVERALL TIME FOR TRAINING: {}h:{}m:{:.5f}s".format(h,m,s))
    print("TRAINING COST: {:.5f}".format(cost))

def test_network_RBFN(layers, X_Test, T_Test):
    # Get results of test data
    activations = forward_step_RBFN(X_Test, layers)  # Get activation of test samples
    Y = activations[-1]
    cost = layers[1].get_cost(Y, T_Test)
    print("TEST COST: {:.5f}".format(cost))
    return Y
```

```
>>>>>> RBFN_Q2C Classic K-Means Clustering Architecture <<<<<<
INPUT LAYER NODES: 784
HIDDEN LAYER (Gaussian Interpolation) NODES: 2
OUTPUT LAYER (Lin) NODES: 1
TEST COST: -0.00639
```

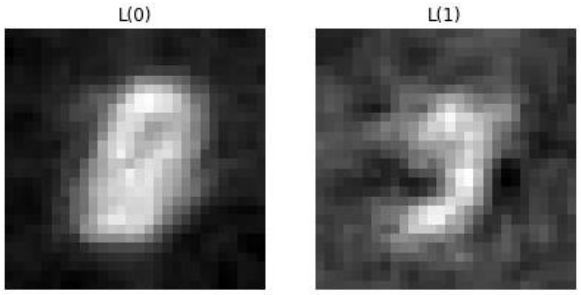



```
CENTER 0:
  MEAN=0.57925, SD=0.04940
CENTER 1:
  MEAN=0.36699, SD=0.08175
CLASS 0:
  MEAN=0.45674, SD=0.02775
CLASS 1:
  MEAN=0.46808, SD=0.02164
```

The RBFN Centers:



The Class Means:



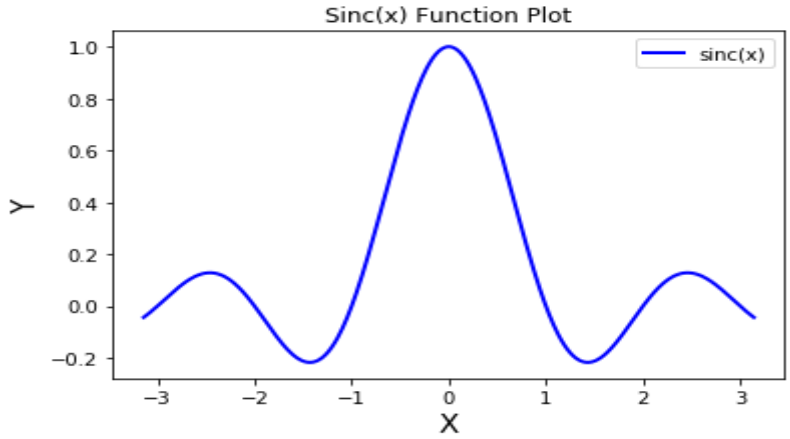
As expected, the centers look much more as class 0 and quite like each other (when ignoring the reversed b/w). This is due to class 0 having 4 times more training samples than class 1. This imbalance sums up the overall performance of the solutions, which are inherently limited by a big bias towards class 0.

Q3: My SOM Implementation:

```
def euclidean_distance(X1, X2):
        """Calculate the Euclidean distances between X1 and X2."""
        return np.sqrt(np.absolute(-2*np.dot(X1, X2.T) + np.sum(X2**2,axis=1) + np.sum(X1**2,
axis=1)[:, np.newaxis]))

class SOMLayer(object):
    """Self Organizing Map class."""

    def __init__(self, M, N, n_in):
        """Initialize the SOM classifier. The centers are initialised randomly
        Args:
            n_in (int): The number of input variables (features per sample). Indicates the
number of weights per neuron.
            M, N (int): The dimensions of the neuron map.
        """
        self.W = np.random.randn(M, N, n_in)
        self.N_N_Dist = self.compute_map_distances(M, N)
        self.EW_0 = np.sqrt(M**2 + N**2)/2

    def compute_map_distances(self, M, N):
        """ Compute the distances between aall neuron in the SOM.
            Args:
                M,N (int): The SOM dimensions. """
        map_distances = np.zeros((M, N, M, N))
        for idx_win_M in range(M):
            for idx_win_N in range(N):
                for idx_M in range(M):
                    for idx_N in range(N):
                        map_distances[idx_win_M, idx_win_N, idx_M, idx_N] =\
                        np.power(float(idx_win_M) - float(idx_M), 2) +
np.power(float(idx_win_N) - float(idx_N), 2)
        return map_distances

    def get_winner_distances(self, m, n):
        """ Get the distances between a neuron & all other neurons in the SOM.
            Args:
                m,n (int): The winner neuron coordinates. """
        return self.N_N_Dist[m, n, :, :]

    def train(self, X, Learning_Rate_0, max_nb_itrs = 1000, verbose = False):
        """ Compute the SOM neuron weights."""
        # Calculating the time-constant controlling the effective width decay rate.
        t1 = max_nb_itrs/np.log(self.EW_0)
        # Calculating the time-constant controlling the learning rate decay rate.
        t2 = max_nb_itrs
        start_time = time.time()
        print("Starting Training")
        # TRAINING - self-organizing & convergence phase
        for itr in range(0, max_nb_itrs):
            W_old = np.copy(self.W)
            # Step 1 - randomly selecting an input vector
            idx = np.random.randint(X.shape[0], size=1)
            # Step 2 - competitive process: Finding the winner neuron
            dists_X_N = np.zeros((self.W.shape[0], self.W.shape[1]))
            for idx_m, m in enumerate(self.W):
                for idx_n, n in enumerate(m):
                    dists_X_N[idx_m][idx_n] =
np.asscalar(euclidean_distance(X[idx].reshape(1, X.shape[1]), n.reshape(1, self.W.shape[2])))
            label_closest_m = int(np.modf(np.argmin(dists_X_N)/self.W.shape[1])[1])
            label_closest_n = np.argmin(dists_X_N) - label_closest_m*self.W.shape[1]
            # Step 3A - Get the distances b/w the winner neuron and all neurons in the map
            dists_W_N = self.get_winner_distances(label_closest_m, label_closest_n)
            # Step 3B - Calculating the time-varying components
            Effective_Width = self.EW_0*np.exp(-itr/t1)
            Neighborhood_Function = np.exp(-(dists_W_N**2) / (2*Effective_Width**2))
            Learning_Rate = np.max((Learning_Rate_0*np.exp(-itr/t2), 0.01))
            # Step 3C - Updating the weights
            for idx_m, m in enumerate(self.W):
                for idx_n, n in enumerate(m):
                    self.W[idx_m][idx_n] = n +
Learning_Rate*Neighborhood_Function[idx_m][idx_n]*(X[idx] - n)
```

```
            # Step 4 - Check for convergence or iteration threshold.
            W_upd = np.absolute(np.sum(self.W - W_old))
            if verbose and X.shape[1] == 2 and (itr%int(max_nb_itrs*0.1)) == 0:
                print("ITERATION: {}".format(itr))
                plot_SOM(self.W, X, time.time() - start_time)
            elif verbose and X.shape[1] > 2 and (itr%int(max_nb_itrs*0.1)) == 0:
                SOM_W, W_X_SOM, W_T_SOM = self.get_winners(X)
                print("ITERATION: {}".format(itr))
                plot_SOM_LAB(W_T_SOM, time.time() - start_time)
            if (itr%int(max_nb_itrs*0.1)) == 0:
                print("ITERATION: {} WEIGHTS UPDATE: {}".format(itr, W_upd))
        print("MAX NUMBER OF ITERATIONS REACHED!")
        end_time = time.time()
        return end_time - start_time

    def get_mapping(self, X):
        """Classify the input samples according to the map of the neurons.
            Args:
                X (NumPy Array.shape(P,K)): Input samples.
            Returns:
                self.W (NumPy Array.shape(M,N,K+1)): The SOM Network Neurons Weights.
                X_SOM (NumPy Array.shape(P,K+1)): The Input Samples Marked By The Closest SOM
Network Neuron.
        """
        X_MAP = np.zeros(X.shape)
        for idx_x, x in enumerate(X):
            dists_x_N = np.zeros((self.W.shape[0], self.W.shape[1]))
            for idx_m, m in enumerate(self.W):
                for idx_n, n in enumerate(m):
                    dists_x_N[idx_m][idx_n] = np.asscalar(euclidean_distance(x.reshape(1,
X.shape[1]),
                                                                             n.reshape(1,
self.W.shape[2])))
            idx_closest_m = int(np.modf(np.argmin(dists_x_N)/self.W.shape[1])[1])
            idx_closest_n = np.argmin(dists_x_N) - idx_closest_m*self.W.shape[1]
            X_MAP[idx_x] = self.W[idx_closest_m][idx_closest_n]
        return self.W, X_MAP

    def get_winners(self, X):
        """Build the contextual (semantic) map of the neurons.
            Args:
                X (NumPy Array): Input samples.
            Returns:
                self.W (NumPy Array.shape(M,N,K+1)): The SOM Network Neurons Weights.
                W_X_SOM (NumPy Array.shape(M,N,K)): The SOM Network Neurons Marked By The
Closest Input Sample.
                W_X_SOM (NumPy Array.shape(M,N)): The SOM Network Neurons Marked By The
Closest Input Sample's Label.
        """
        dists_N_X = np.zeros((self.W.shape[0], self.W.shape[1], X.shape[0]))
        for idx_m, m in enumerate(self.W):
            for idx_n, n in enumerate(m):
                dists_N_X[idx_m][idx_n] = euclidean_distance(n.reshape(1, self.W.shape[2]),
X)
        W_X_SOM = np.zeros((self.W.shape[0], self.W.shape[1], X.shape[1]-1))
        W_T_SOM = np.zeros((self.W.shape[0], self.W.shape[1]))
        for idx_m, m in enumerate(dists_N_X):
            for idx_n, n in enumerate(m):
                W_X_SOM[idx_m][idx_n] = X[np.argmin(n)][:-1]
                W_T_SOM[idx_m][idx_n] = X[np.argmin(n)][-1]
        return self.W, W_X_SOM, W_T_SOM

    def get_winner(self, x):
        """Get the coordinates of the winner neuron for the input sample (x)."""
        dists_x_N = np.zeros((self.W.shape[0], self.W.shape[1]))
        for idx_m, m in enumerate(self.W):
            for idx_n, n in enumerate(m):
                dists_x_N[idx_m][idx_n] = np.asscalar(euclidean_distance(x.reshape(1,
self.W.shape[2]), n.reshape(1, self.W.shape[2])))
            idx_closest_m = int(np.modf(np.argmin(dists_x_N)/self.W.shape[1])[1])
            idx_closest_n = np.argmin(dists_x_N) - idx_closest_m*self.W.shape[1]
        return idx_closest_m, idx_closest_n
```

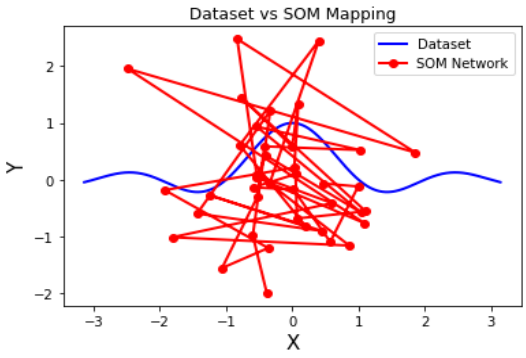
3(a):



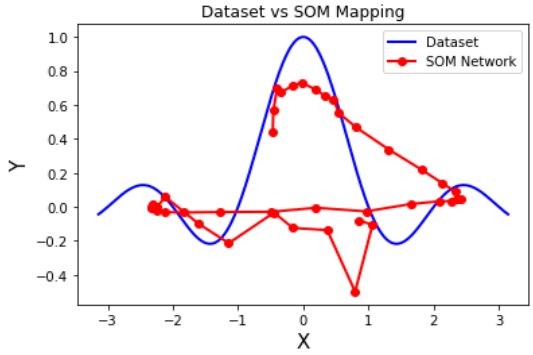

```
>>>>>> Self-Organizing Map Architecture <<<<<<
MAP NODES: 1x40
WEIGHTS PER NODE: 2
```

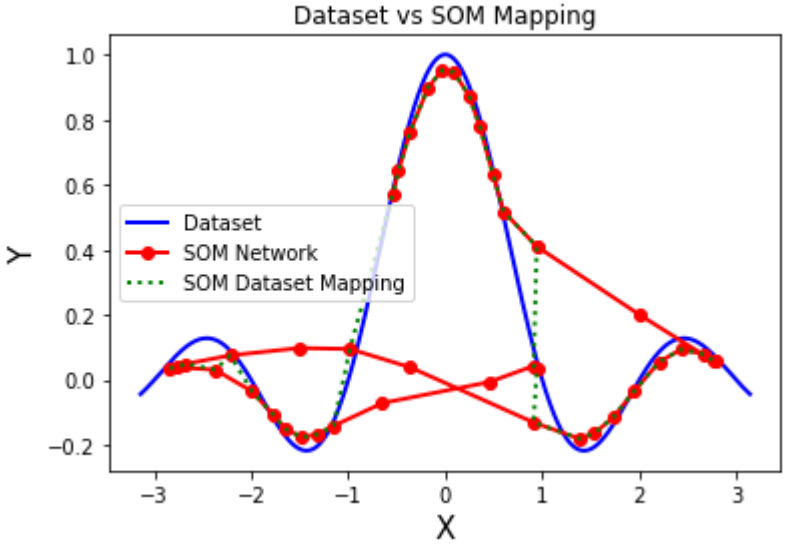
1000 Iterations Learning

```
ITERATION: 0
```

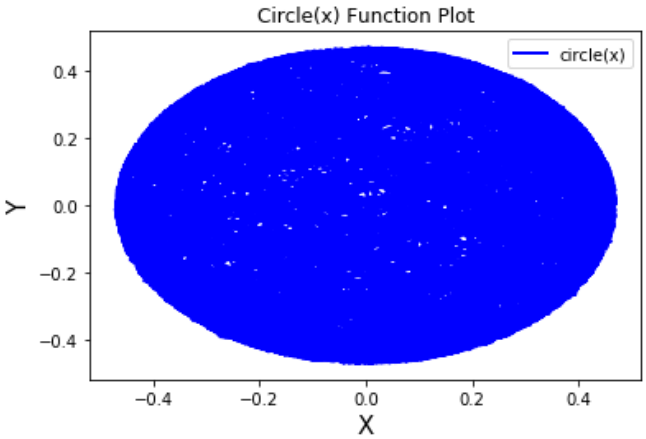



```
ITERATION: 200
```



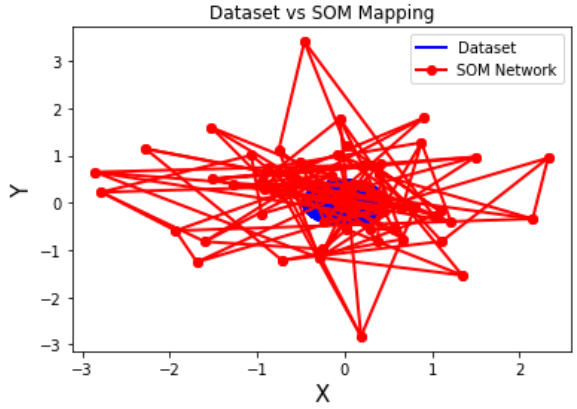


3(b):

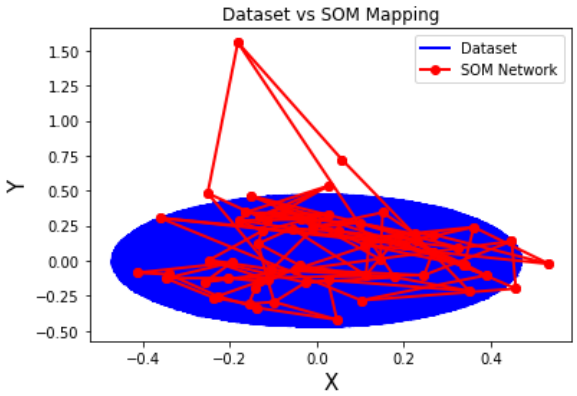



```
>>>>>> Self-Organizing Map Architecture <<<<<<
MAP NODES: 8x8
WEIGHTS PER NODE: 2
```

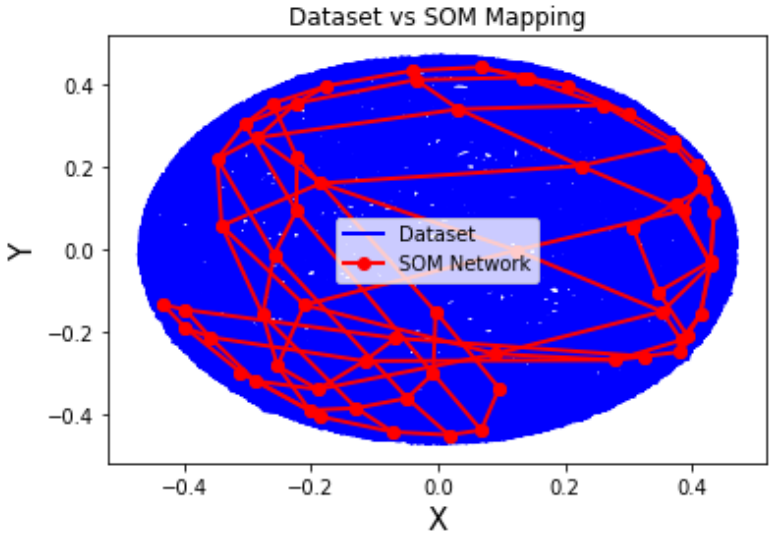
ITERATION: 0



ITERATION: 100



ITERATION: 500



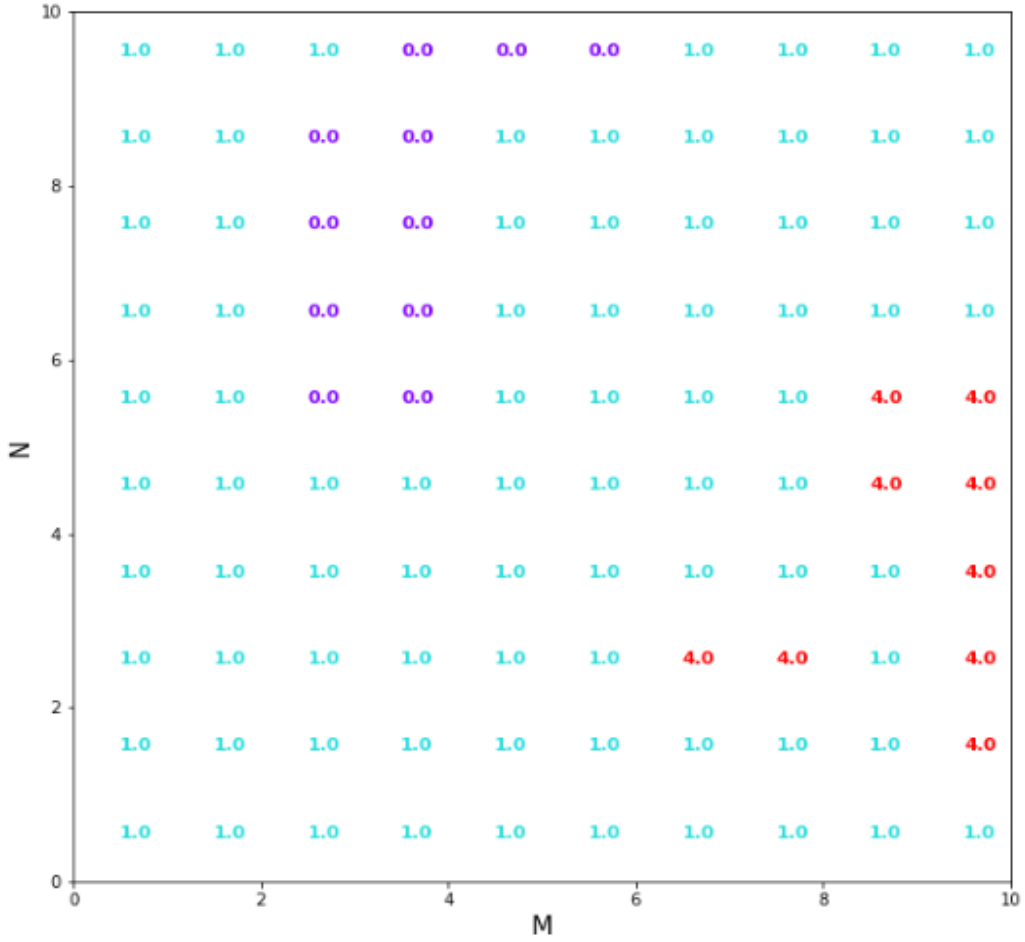
3(c):

```
>>>>>> DIGITS in (2,3) REMOVED <<<<<<
>>>>>> AUGMENTED DIGITS DATASET FEATURE SPACE <<<<<<
>>> FULL <<<
(660, 785)
>>> TRAIN <<<
(600, 784)
(600,)
(600, 785)
>>> TEST <<<
(60, 784)
(60,)
(60, 785)

>>>>>> Self-Organizing Map Architecture <<<<<<
MAP NODES: 10x10
WEIGHTS PER NODE: 785

TRAIN ACCURACY: 62.67% | TEST ACCURACY: 65.00%
```

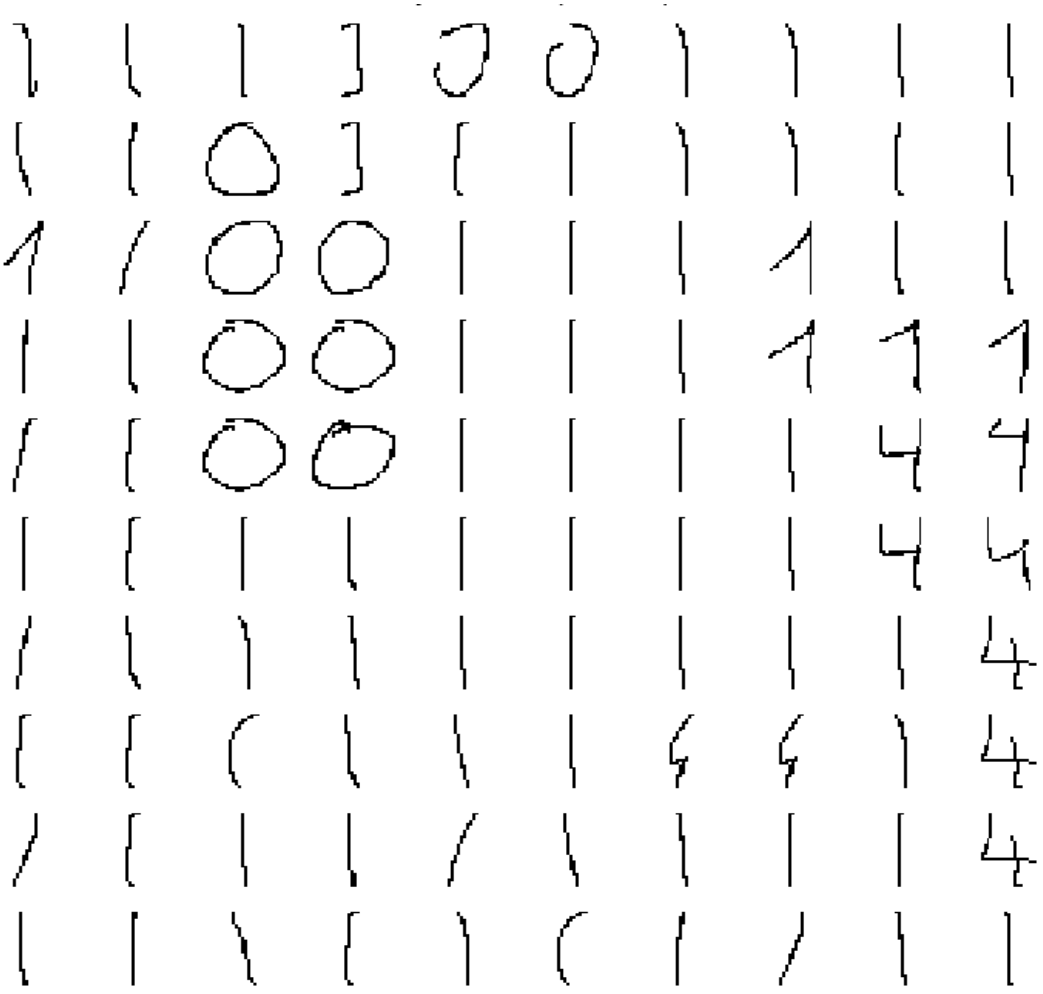
In the first two plot, the SOM network's neurons are visualized by the closest image's label and the close image respectively. It can be clearly seen that the map has organized into grouping the clusters at different areas in the network. The third plot visualizes the SOM network's neurons by their weights[:-1]. The clusters can be observed again. It is interesting to note how the boundary's neurons are a merged sample of the two labels on each side. On the left/down side it can be seen that the neurons did not get much stimulation, in the closest-sample plots, they all are closest to **1,** which might be due to **1** having least distinguishable feature from white noise (just a single line). The reason for unstimulated neurons might be bigger than needed map & random weights initialization (instead of PCA or other case-tailored approach), which distorts the spatial structure of the map. In the last plot, the training input samples are displayed by winner neuron. The same observations can be made – the separate classes have been grouped in different areas of the map and the boundary neurons get hits from multiple labels. It can also be noted that the "noisy", not-stimulated neurons do not win for any sample, which is to be expected as the well-formed neurons would always be closer.

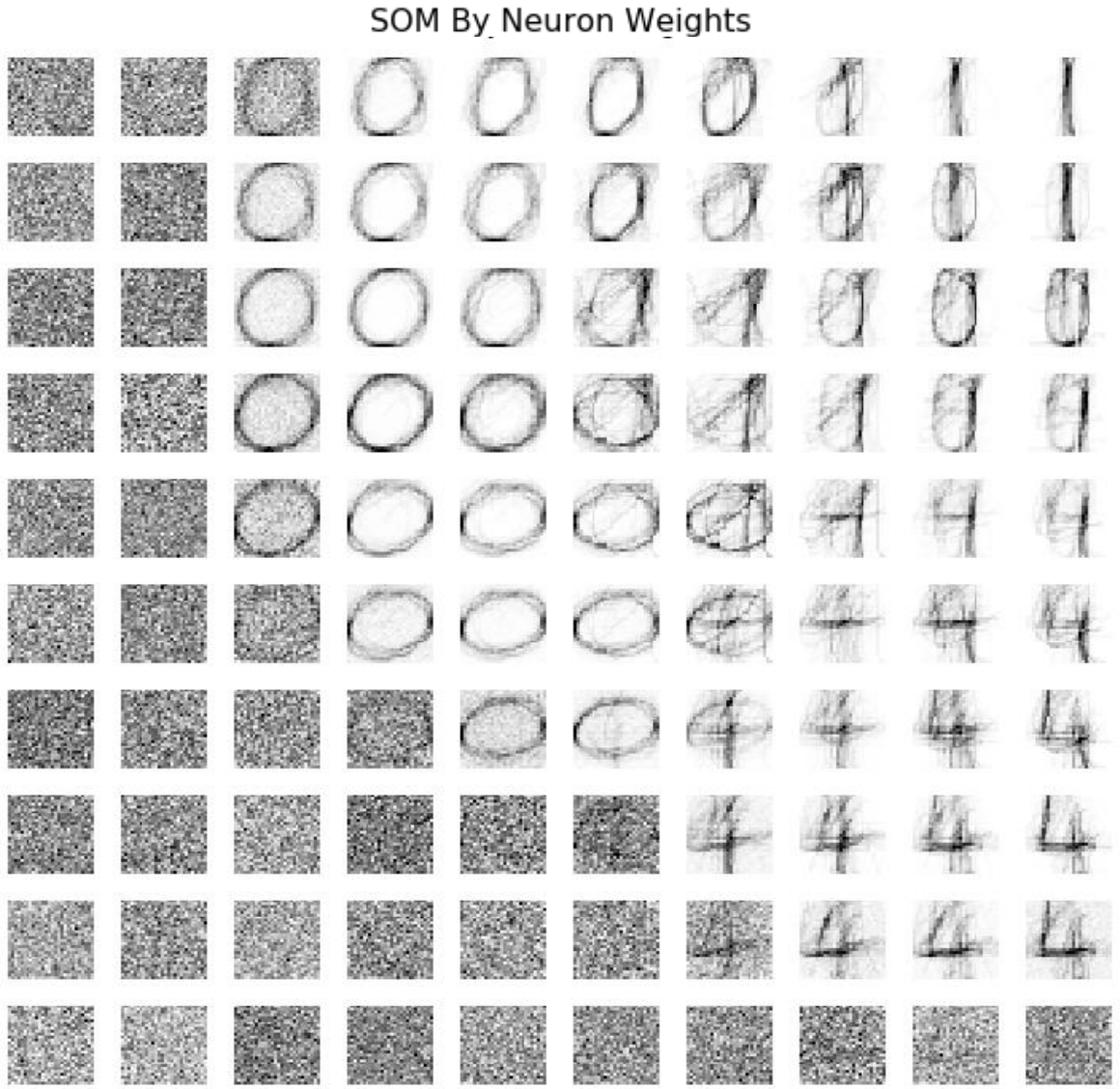
The SOM performs well, with top-1 test accuracy higher than training accuracy and much higher than the 33% pick-at-random probability. The shows the generalization strength of SOMs.

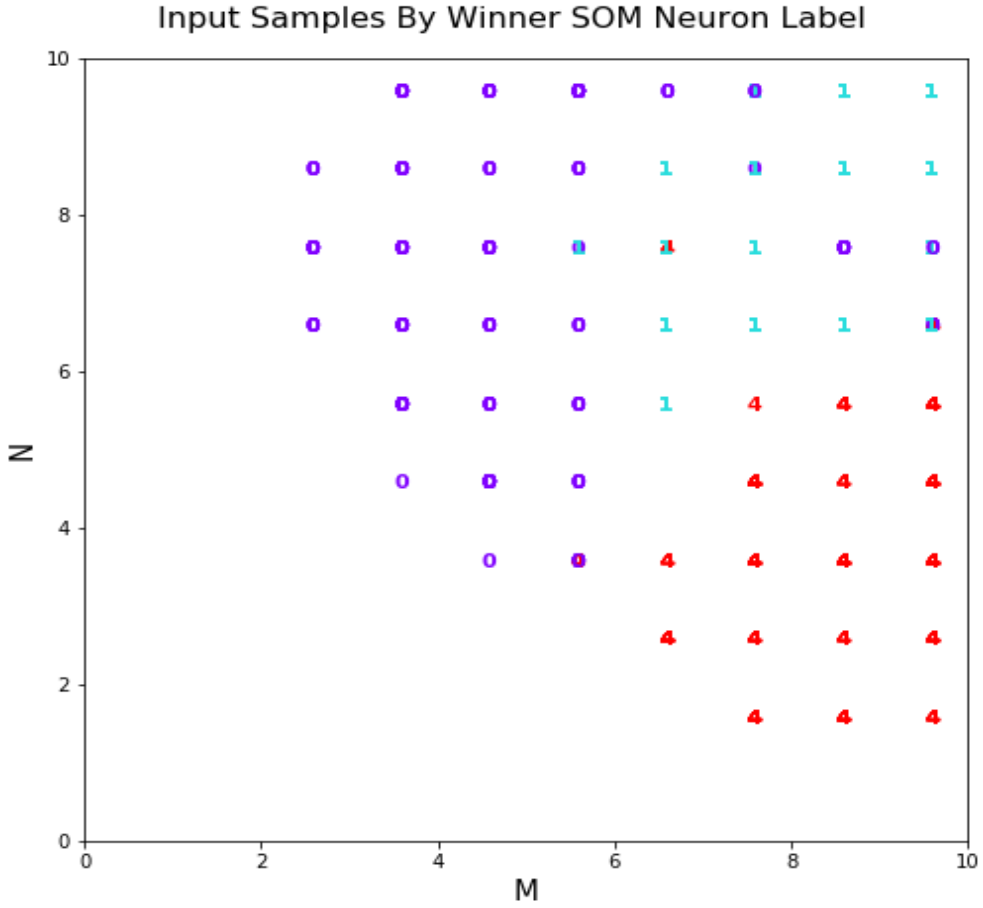
SOM By Closest Input Sample Label



SOM By Closest Input Sample




SOM By Neuron Weights


Input Samples By Winner SOM Neuron Label
N
M
0
2
4
6
8
10