EE5904/ME5404 Neural Networks: Homework #3
XU NUO, A0313771H

Q1. Function Approximation with RBFN

Consider using RBFN to approximate the following function:

$$y = 1.2\sin(\pi x) - \cos(2.4\pi x), \quad for\ x \in [-1.6, 1.6]$$

The training set is constructed by dividing the range [−1.6, 1.6] using a uniform step length 0.08, while the test set is constructed by dividing the range [−1.6, 1.6] using a uniform step length 0.01. Assume that the observed outputs in the training set are corrupted by random noise as follows.

$$y(i) = 1.2\sin\big(\pi x(i)\big) - \cos\left(2.4\pi\big(\pi x(i)\big) + 0.3n(i)\right.$$

where the random noise $n(i)$ is Gaussian noise with zero mean and stand deviation of one, which can be generated by MATLAB command *randn*. Note that the test set is not corrupted by noises. Perform the following computer experiments:

a). Use the **Exact Interpolation Method** (as described on pages 17-26 in the slides of lecture five) and determine the weights of the RBFN. Assume the RBF is Gaussian function with standard deviation of 0.1. Evaluate the approximation performance of the resulting RBFN using the test set.

b). Follow the strategy of **Fixed Centers Selected at Random** (as described on page 38 in the slides of lecture five), randomly select 20 centers among the sampling points. Determine the weights of the RBFN. Evaluate the approximation performance of the resulting RBFN using test set. Compare it to the result of part a).

c). Use the same centers and widths as those determined in part a) and apply the regularization method as described on pages 43-46 in the slides for lecture five. Vary the value of the regularization factor and study its effect on the performance of RBFN.

A1.

a). The code for constructing a radial basis function network (RBFN) for function approximation using the Exact Interpolation Method is as follows:

```
clc; clear; close all;

%% Generate training and test data
x_train = -1.6:0.08:1.6; % Training set with a uniform step length 0.08
y_train = 1.2*sin(pi*x_train) - cos(2.4*pi*x_train) + 0.3*randn(size(x_train)); % Add
random noise
x_test = -1.6:0.01:1.6; % Test set with a uniform step length 0.01
y_test = 1.2*sin(pi*x_test) - cos(2.4*pi*x_test); % Ground truth without noise

%% Construct RBF Kernel Matrix for Training Data
N = length(x_train); % Number of training points
sigma = 0.1; % Standard deviation for Gaussian RBF
Phi_train = zeros(N, N); % Initialize Gaussian Kernel
```

```
for i = 1:N
    for j = 1:N
        Phi_train(i, j) = exp(-((x_train(i) - x_train(j))^2) / (2 * sigma^2)); % Gaussian
Kernel
    end
end

%% Solve for weights w
w = Phi_train \ y_train';  % Compute weights using w = inv(Phi) * d

%% Compute RBF Output on Test Set
M = length(x_test);  % Number of test points
Phi_test = zeros(M, N);
for i = 1:M
    for j = 1:N
        Phi_test(i, j) = exp(-((x_test(i) - x_train(j))^2) / (2 * sigma^2));
    end
end
y_pred = Phi_test * w; % Compute predicted values

%% Plot results
figure;
plot(x_test, y_test, 'b-', 'LineWidth', 2); hold on; % True function
plot(x_test, y_pred, 'r--', 'LineWidth', 2);         % RBFN approximation
scatter(x_train, y_train, 'ko', 'MarkerFaceColor', 'k'); % Training points
legend('True Function', 'RBFN Approximation', 'Training Data');
xlabel('x');
ylabel('y');
title('Function Approximation using RBFN (Exact Interpolation)');
grid on;
```

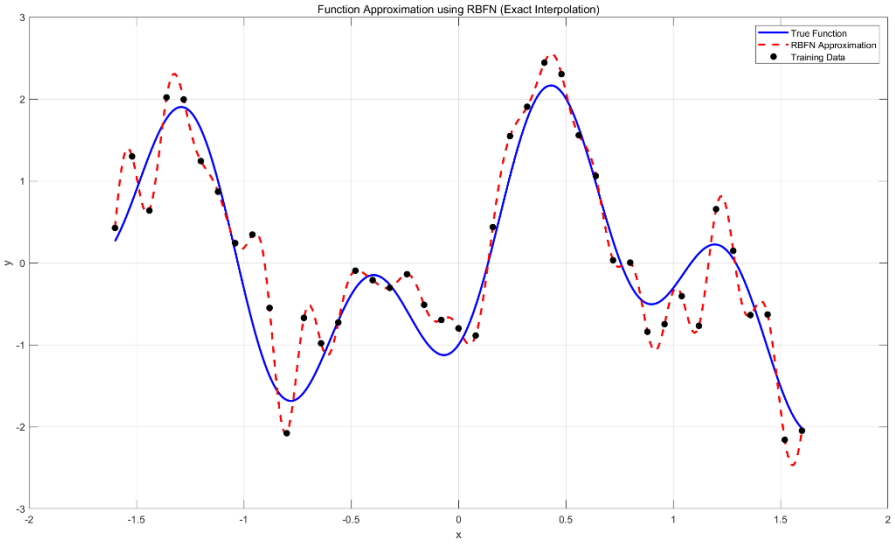

Fig. 1. Function Approximation using Exact Interpolation

In this part, the interpolation function is a linear combination of a set of radial basis functions. Exact interpolation means that for every sample point, the interpolation function must pass through that point exactly. Therefore, we can write the linear equation system as:

$$\sum_{j=1}^{N} w_j \phi_j(x_i) = y_i \quad i = 1,2,3,\dots,N$$

Here, $\phi_j(x_i)$ is the radial basis function, ω_j is the weight to be determined. We can solve the weights by

$$w = \Phi^{-1} Y$$

Figure 1 shows the result of function approximation using the exact interpolation method. From the figure, it can be observed that due to the addition of random noise in the training set, the black points do not fall on the actual function curve. Since the exact interpolation method is used, the approximated function passes through all the training data points, causing the test set curve to follow the general trend of the original function but not coincide with it. This demonstrates that the exact interpolation method leads to model overfitting and is unable to handle cases where noise is present in the training data, making the model incapable of adapting to new data.

b). The code for constructing an RBFN for function approximation using the Fixed Centers Selected at Random Method is as follows:

```
clc; clear; close all;


%% Generate training and test data
x_train = -1.6:0.08:1.6; % Training set with a uniform step length 0.08
y_train = 1.2*sin(pi*x_train) - cos(2.4*pi*x_train) + 0.3*randn(size(x_train)); % Add
random noise
x_test = -1.6:0.01:1.6; % Test set with a uniform step length 0.01
```

```
y_test = 1.2*sin(pi*x_test) - cos(2.4*pi*x_test); % Ground truth without noise

%% Select random centers
num_centers = 20; % Number of randomly selected centers
center_indices = randperm(length(x_train), num_centers); % Randomly select 20 indices
centers = x_train(center_indices); % Select the corresponding 20 points in x_train as
the centers of RBF by indices

%% Compute σ based on max distance
d_max = max(pdist(centers')); % Calculate the maximum distance between the chosen
centres
sigma = d_max / sqrt(2 * num_centers); % Adaptive sigma

%% Construct RBF Kernel Matrix
N = length(x_train); % Number of training points
Phi_train = zeros(N, num_centers); % Initialize Gaussian Kernel

for i = 1:N
    for j = 1:num_centers
        Phi_train(i, j) = exp(-((x_train(i) - centers(j))^2) * (num_centers /
d_max^2)); % Gaussian Kernel
    end
end

%% Solve for weights w using Least Squares
w = pinv(Phi_train' * Phi_train) * (Phi_train' * y_train');

%% Compute RBF Output on Test Set
M = length(x_test);
Phi_test = zeros(M, num_centers);
for i = 1:M
    for j = 1:num_centers
        Phi_test(i, j) = exp(-((x_test(i) - centers(j))^2) * (num_centers / d_max^2));
    end
end
y_pred = Phi_test * w; % Compute predicted values

%% Plot Results
figure;
plot(x_test, y_test, 'b-', 'LineWidth', 2); hold on;
plot(x_test, y_pred, 'r--', 'LineWidth', 2);
scatter(x_train, y_train, 'ko', 'MarkerFaceColor', 'k');
scatter(centers, zeros(size(centers)), 'ro', 'MarkerFaceColor', 'r', 'SizeData', 100);
legend('True Function', 'RBFN Approximation', 'Training Data', 'Selected Centers');
```

```
xlabel('x');
ylabel('y');
title('Function Approximation using RBFN (Fixed Centers Selected at Random)');
grid on;
```

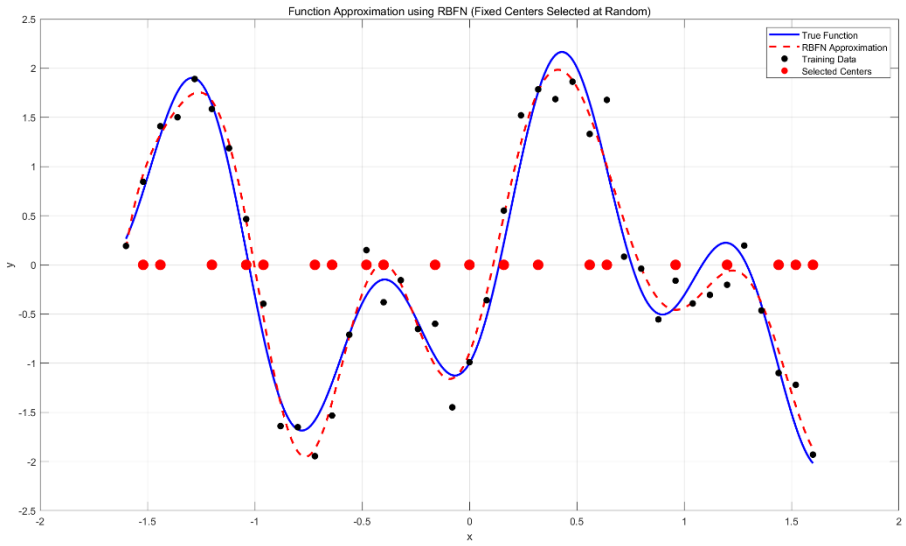

Fig. 2. Function Approximation using Fixed Centers Selected at Random

In this part, function approximation is performed using the Fixed Centers Selected at Random method, which directly selects 20 points randomly from the training data as the centers of radial basis functions, computes the Gaussian kernel response of each training data point to the selected centers, and solves for the weights using the least squares method.

$$w = (\Phi^T \Phi)^{-1} \Phi^T y$$

By comparing the results of these two figures, it can be concluded that due to the presence of noise in the training data, the exact interpolation method fails to ignore these noisy points, leading to overfitting to the training data and poor fitting performance on the test set. In contrast, “Fixed Centers Selected at Random Method” constructs radial basis functions using randomly chosen centers, which no longer strictly pass through all training data points but instead perform a smoother approximation, thereby reducing sensitivity to noise. This makes the RBF network more robust to noise and enhances its generalization ability.

c). The code for constructing an RBFN for function approximation using Exact Interpolation Method with different regularization factors is as follows:

```
clc; clear; close all;

%% Generate training and test data
x_train = -1.6:0.08:1.6; % Training set with a uniform step length 0.08
```

```
y_train = 1.2*sin(pi*x_train) - cos(2.4*pi*x_train) + 0.3*randn(size(x_train)); % Add
random noise
x_test = -1.6:0.01:1.6; % Test set with a uniform step length 0.01
y_test = 1.2*sin(pi*x_test) - cos(2.4*pi*x_test); % Ground truth without noise

%% Construct RBF Kernel Matrix for Training Data
N = length(x_train); % Number of training points
sigma = 0.1; % Standard deviation for Gaussian RBF
Phi_train = zeros(N, N); % Initialize Gaussian Kernel

for i = 1:N
    for j = 1:N
        Phi_train(i, j) = exp(-((x_train(i) - x_train(j))^2) / (2 * sigma^2)); % Gaussian
Kernel
    end
end

%% Calculate the weights under different regularization parameters λ
lambda_values = [0, 1e-6, 1e-4, 1e-2, 1, 10, 40]; % Different λ
w_results = cell(length(lambda_values), 1);
y_pred_results = cell(length(lambda_values), 1);

for k = 1:length(lambda_values)
    lambda = lambda_values(k);
    w = (Phi_train' * Phi_train + lambda * eye(N)) \ (Phi_train' * y_train');
    w_results{k} = w;  % Store weights for different λ
    M = length(x_test);
    Phi_test = zeros(M, N);
    for i = 1:M
        for j = 1:N
            Phi_test(i, j) = exp(-((x_test(i) - x_train(j))^2) / (2 * sigma^2)); %
Compute RBF Output on Test Set
        end
    end

    y_pred_results{k} = Phi_test * w; % Compute predicted values
end

%% Draw multiple subplots, each corresponding to a different λ
figure;
num_lambda = length(lambda_values);
num_cols = 3;
num_rows = ceil(num_lambda / num_cols);
```

```
for k = 1:num_lambda
    subplot(num_rows, num_cols, k);
    hold on;
    plot(x_test, y_test, 'b-', 'LineWidth', 2);
    plot(x_test, y_pred_results{k}, 'r--', 'LineWidth', 2);
    scatter(x_train, y_train, 'ko', 'MarkerFaceColor', 'k');
    legend('True Function', 'RBFN Approximation', 'Training Data', 'FontSize', 4.5);
    xlabel('x');
    ylabel('y');
    title(sprintf('\\lambda = %.1e', lambda_values(k)));
    grid on;
    hold off;
end

sgtitle('Function Approximation using RBFN with Regularization (Exact Interpolation)');
```

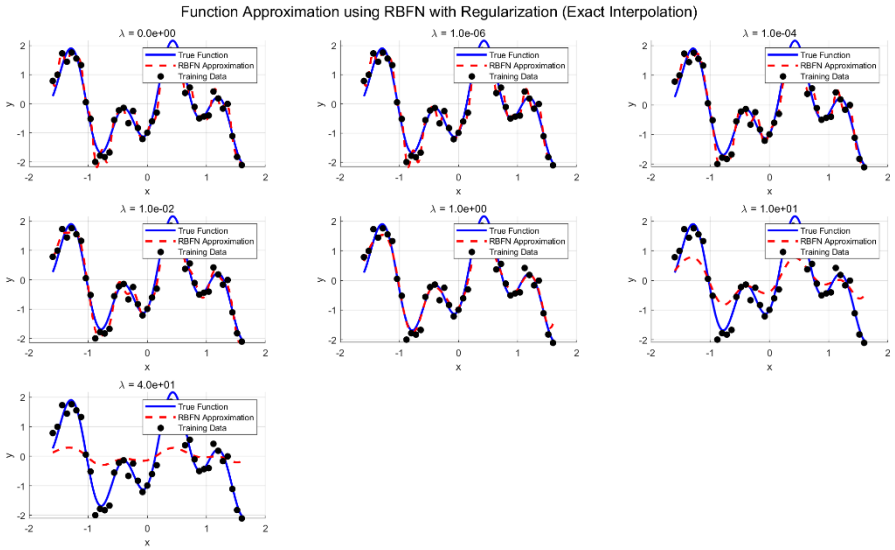

Fig. 3. Function Approximation using Exact Interpolation Method with regularization

In this part, the least squares method is still used to solve the weights, but a regularization factor is added:

$$w = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$$

As can be seen from the figure, the model will completely fit the training data causing overfitting when there is no regularization. When λ is small like 10^{-6} and 10^{-4}, the model can still fit the training data well but there is still a certain degree of overfitting. When λ increases to the range of $10^{-2}\sim1$, the fitting curve begins to become smoother while still maintaining a good fitting effect. But when λ becomes too large, the regularization constraint on the weights is too strong, resulting in over-smoothing of the fitting curve, and the model becomes too simple to effectively capture the complex patterns of the data, resulting in underfitting.

Q2. Handwritten Digits Classification using RBFN

In this task, you will build a handwritten digits classifier using RBFN. The training data is provided in **MNIST_M.mat**. Each binary image is of size 28*28. There are 10 classes in MNIST_M.mat; please **select** two classes according to the last two different digits of your matric number (e.g. A0642311, choose classes 3 and 1; A1234567, choose classes 6 and 7). The images in the selected two classes should be assigned the label "1" for this question's binary classification task, while images in all the remaining eight classes should be assigned the label "0". You are required to complete the following tasks:

a). Use Exact Interpolation Method and apply regularization. Assume the RBF is Gaussian function with standard deviation of 100. Firstly, determine the weights of RBFN without regularization and evaluate its performance; then vary the value of regularization factor and study its effect on the resulting RBFNs' performance.

b). Follow the strategy of "Fixed Centers Selected at Random" (as described in page 38 of lecture five). Randomly select 100 centers among the training samples. Firstly, determine the weights of RBFN with widths fixed at an appropriate size and compare its performance to the result of a); then vary the value of width from 0.1 to 10000 and study its effect on the resulting RBFNs' performance.

c). Try classical "K-Mean Clustering" (as described in pages 39-40 of lecture five) with 2 centers. Firstly, determine the weights of RBFN and evaluate its performance; then visualize the obtained centers and compare them to the mean of training images of each class. State your findings.

A2.

a). The last 2 digits of my matric number are 7 and 1, so "7 class" and "1 class" are selected and labeled "1" accordingly. All the images in the remaining classes are assigned label "0". The code for constructing an RBFN using Exact Interpolation Method with different regularization factors is as follows:

```
clc; clear; close all;

%% Load the MNIST dataset
load mnist_m.mat;

%% Set the labels corresponding to 7 and 1 to 1, and the rest to 0 (Matric number:
A0313771H)
class1 = 7;
class2 = 1;

% Data processing for training and testing
Train_ClassLabel = zeros(size(train_classlabel));
Train_ClassLabel(train_classlabel == class1 | train_classlabel == class2) = 1;
Test_ClassLabel = zeros(size(test_classlabel));
Test_ClassLabel(test_classlabel == class1 | test_classlabel == class2) = 1;
```

```
% Convert the label to double data type to avoid warnings
Train_ClassLabel = double(Train_ClassLabel);
Test_ClassLabel = double(Test_ClassLabel);

Train_Data = train_data;
Test_Data = test_data;

%% Construct RBF Kernel Matrix for Training Data
N = length(Train_ClassLabel);
sigma = 100; % Standard deviation of 100
Phi_train = zeros(N, N); % Initialize Gaussian Kernel

for i = 1:N
    for j = 1:N
        Phi_train(i, j) = exp(-((Train_Data(:,i) - Train_Data(:,j))' * (Train_Data(:,i) -
Train_Data(:,j))) / (2 * sigma^2));
    end
end

%% Calculate the weights under different regularization parameters λ and predict the
output of test set
lambda_values = [0, 1e-6, 1e-4, 1e-2, 1, 10, 40]; % Different λ
w_results = cell(length(lambda_values), 1);
y_pred_results = cell(length(lambda_values), 1);

for k = 1:length(lambda_values)
    lambda = lambda_values(k);

    if lambda == 0

        % Calculate the weight without regularization separately because the calculation
formula is different
        w = Phi_train \ Train_ClassLabel';
    else

        % Calculate weights with regularization using LSE
        w = pinv(Phi_train' * Phi_train + lambda * eye(N)) * (Phi_train' *
Train_ClassLabel');
    end

    w_results{k} = w;

    % Compute the test set RBF kernel matrix
```

```
    M = length(Test_ClassLabel);
    Phi_test = zeros(M, N);

    for i = 1:M
        for j = 1:N
            Phi_test(i, j) = exp(-((Test_Data(:,i) - Train_Data(:,j))' * (Test_Data(:,i)
- Train_Data(:,j))) / (2 * sigma^2));
        end
    end

    % Calculate predicted output
    TrPred = Phi_train * w;
    TePred = Phi_test * w;

%% Calculate classification accuracy and visualize
    TrAcc = zeros(1,1000);
    TeAcc = zeros(1,1000);
    thr = zeros(1,1000);
    TrN = length(Train_ClassLabel);
    TeN = length(Test_ClassLabel);

    for i = 1:1000
        t = (max(TrPred)-min(TrPred)) * (i-1)/1000 + min(TrPred);
        thr(i) = t;
        TrAcc(i) = (sum(Train_ClassLabel(TrPred<t)==0) +
sum(Train_ClassLabel(TrPred>=t)==1)) / TrN;
        TeAcc(i) = (sum(Test_ClassLabel(TePred<t)==0) +
sum(Test_ClassLabel(TePred>=t)==1)) / TeN;
    end

    % Plot the accuracy curve
    figure;
    plot(thr,TrAcc,'.- ',thr,TeAcc,'^-');legend('tr','te');
    xlabel('Threshold');
    ylabel('Accuracy');
    title(sprintf('RBFN Accuracy vs. Threshold (\\lambda = %.1e)', lambda_values(k)));
    grid on;
    hold off;
end
```

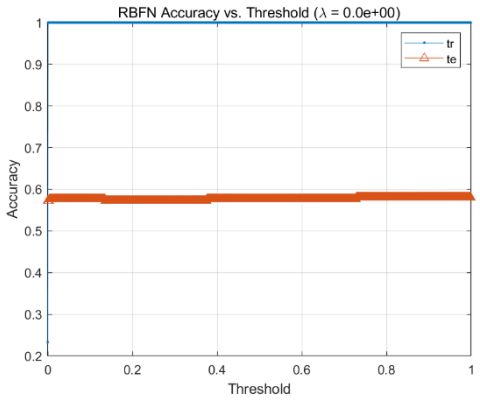

Fig. 4. Exact Interpolation Method

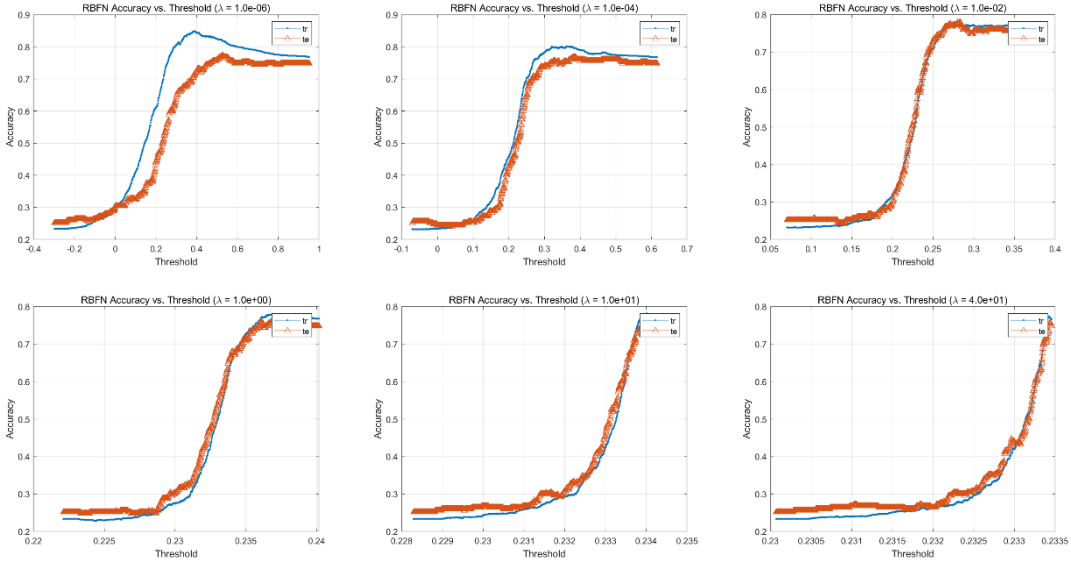

Fig. 5. Exact Interpolation Method with Different Regularization Factors

Without the addition of a regularization factors, the model precisely passes through all training data due to the use of the exact interpolation method. As a result, the training accuracy remains at 100% regardless of the threshold. However, due to overfitting, changes in the threshold have almost no effect on the test accuracy, making the model ineffective for classification. Increasing the regularization factor suppresses overfitting to some extent, but it also smooths the curve, causing the accuracy increase to become more gradual. When $\lambda = 1$ the model performs relatively well, achieving a classification accuracy of around 80%.

b). The code for constructing an RBFN using the Fixed Centers Selected at Random Method with different widths is as follows:

```
clc; clear; close all;


%% Load the MNIST dataset
load mnist_m.mat;
```

```
%% Set the labels corresponding to 7 and 1 to 1, and the rest to 0 (Matric number:
A0313771H)
class1 = 7;
class2 = 1;

% Data processing for training and testing
Train_ClassLabel = zeros(size(train_classlabel));
Train_ClassLabel(train_classlabel == class1 | train_classlabel == class2) = 1;
Test_ClassLabel = zeros(size(test_classlabel));
Test_ClassLabel(test_classlabel == class1 | test_classlabel == class2) = 1;

% Convert the label to double data type to avoid warnings
Train_ClassLabel = double(Train_ClassLabel);
Test_ClassLabel = double(Test_ClassLabel);

Train_Data = train_data;
Test_Data = test_data;

%% Select 100 random centers from the training data
num_centers = 100; % Number of randomly selected centers
center_indices = randperm(size(Train_Data, 2), num_centers); % Randomly select 100
indices
centers = Train_Data(:, center_indices); % Select the corresponding 100 points in
Train_Data as the centers of RBF by indices

%% Compute σ based on max distance
d_max = max(pdist(centers'));
sigma_adaptive = d_max / sqrt(2 * num_centers);  % Adaptive sigma for RBF

%% Define sigma values (including adaptive sigma)
sigma_values = [sigma_adaptive, 0.1, 1, 10, 100, 1000, 10000];

%% Iterate over different sigma values and compute accuracy
N = length(Train_ClassLabel);
M = length(Test_ClassLabel);

for s = 1:length(sigma_values)
    sigma = sigma_values(s);

    % Compute RBF kernel matrix for training set
    Phi_train = zeros(N, num_centers);
    for i = 1:N
        for j = 1:num_centers
```

```
            Phi_train(i, j) = exp(-((Train_Data(:,i) - centers(:,j))' * (Train_Data(:,i)
- centers(:,j))) / (2 * sigma^2));
        end
    end

    % Compute weights
    w = Phi_train \ Train_ClassLabel';

    % Compute RBF kernel matrix for test set
    Phi_test = zeros(M, num_centers);
    for i = 1:M
        for j = 1:num_centers
            Phi_test(i, j) = exp(-((Test_Data(:,i) - centers(:,j))' * (Test_Data(:,i) -
centers(:,j))) / (2 * sigma^2));
        end
    end

    % Compute predicted output
    TrPred = Phi_train * w;
    TePred = Phi_test * w;

%% Compute classification accuracy for different thresholds
    TrAcc = zeros(1,1000);
    TeAcc = zeros(1,1000);
    thr = zeros(1,1000);
    TrN = length(Train_ClassLabel);
    TeN = length(Test_ClassLabel);

    for i = 1:1000
        t = (max(TrPred)-min(TrPred)) * (i-1)/1000 + min(TrPred);
        thr(i) = t;
        TrAcc(i) = (sum(Train_ClassLabel(TrPred<t)==0) +
sum(Train_ClassLabel(TrPred>=t)==1)) / TrN;
        TeAcc(i) = (sum(Test_ClassLabel(TePred<t)==0) +
sum(Test_ClassLabel(TePred>=t)==1)) / TeN;
    end

%% Plot threshold vs. accuracy curve
    figure;
    plot(thr,TrAcc,'.- ',thr,TeAcc,'^-');legend('tr','te');
    xlabel('Threshold');
    ylabel('Accuracy');
    if sigma == sigma_adaptive
        title(sprintf('Fixed Centers (Adaptive Sigma = %.2f)', sigma));
```

```
    else
        title(sprintf('Fixed Centers (Width = %.2f)', sigma));
    end
    grid on;
end
```

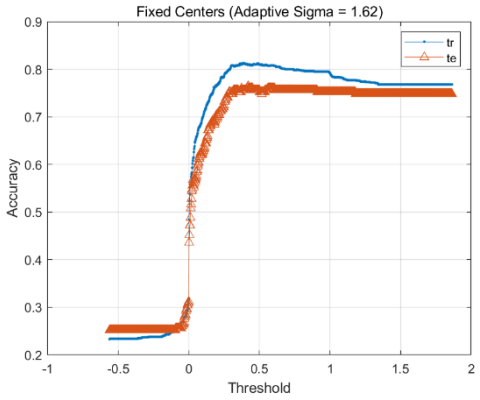

Fig. 6. Fixed Centers Selected at Random with Width Fixed at an Appropriate Size

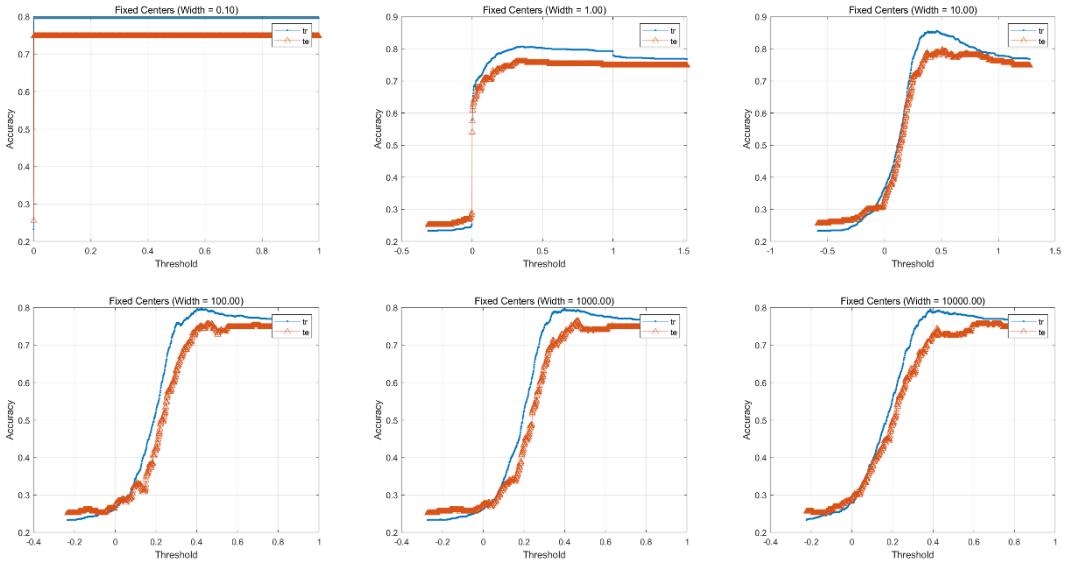

Fig. 7. Fixed Centers Selected at Random with Various Widths

As mentioned in the slides, the simplest and quickest approach to setting the RBF parameters is to take their centers as fixed at M points selected at random from the N datapoints, and to take their widths to be equal and fixed at an appropriate size for the distribution of data points. We can define RBFs centered at μ_i:

$$\varphi_i(x) = e^{-\frac{M}{d_{max}^2}||x-\mu_i||^2}$$

Where d_{max} is the maximum distance between the chosen centers. The spreads are

$$\sigma_i = \frac{d_{max}}{\sqrt{2M}}$$

which ensures that individual RBFs are neither too peaked or too flat. The adaptive sigma calculated by this method is 1.62 as presented in Figure 6.

As can be seen from Figure 7, when the RBFN network width is narrow, the influence range of each basis function is narrow. This will cause the network to be very sensitive to every detail in the training data, and the network performs well on the training set.

As the width of the RBF network increases, the influence range of each basis function will become wider. The accuracy of the model will also slow down as the threshold increases, and the final classification accuracy is 70%-80%.

c). The code for constructing an RBFN using “K-Mean Clustering” is as follows:

```
clc; clear; close all;

%% Load the MNIST dataset
load mnist_m.mat;

%% Set the labels corresponding to 7 and 1 to 1, and the rest to 0 (Matric number:
A0313771H)
class1 = 7;
class2 = 1;

% Data processing for training and testing
Train_ClassLabel = double(train_classlabel == class1 | train_classlabel == class2);
Test_ClassLabel = double(test_classlabel == class1 | test_classlabel == class2);
Train_Data = train_data;
Test_Data = test_data;

%% Select only class 7 and 1 for K-Means clustering
trainIdx = find(train_classlabel == class1 | train_classlabel == class2);
KMeans_Data = Train_Data(:, trainIdx);
KMeans_Labels = Train_ClassLabel(trainIdx);

%% Apply Manual K-Means Clustering (2 centers)
num_clusters = 2;
max_iters = 20;  % Maximum iterations
rng(1); % Set random seed for reproducibility

% Initialize centers randomly from the selected data
rand_indices = randperm(size(KMeans_Data, 2), num_clusters);
centers = KMeans_Data(:, rand_indices);

% K-Means Iteration
for iter = 1:max_iters
    % Assign each sample to the closest center
```

```
    cluster1 = [];
    cluster2 = [];

    for i = 1:size(KMeans_Data, 2)
        d1 = norm(KMeans_Data(:, i) - centers(:, 1));
        d2 = norm(KMeans_Data(:, i) - centers(:, 2));

        if d1 < d2
            cluster1 = [cluster1, KMeans_Data(:, i)];
        else
            cluster2 = [cluster2, KMeans_Data(:, i)];
        end
    end

    % Update centers
    new_center1 = mean(cluster1, 2);
    new_center2 = mean(cluster2, 2);

    % Convergence check
    if norm(new_center1 - centers(:, 1)) < 1e-6 && norm(new_center2 - centers(:, 2)) <
1e-6
        break;
    end

    centers(:, 1) = new_center1;
    centers(:, 2) = new_center2;
end

%% Compute σ based on max distance
d_max = max(pdist(centers'));
sigma_adaptive = d_max / sqrt(2 * num_clusters);

%% Construct RBF Kernel Matrix for Training Data
N = length(Train_ClassLabel);
M = length(Test_ClassLabel);
Phi_train = zeros(N, num_clusters);

for i = 1:N
    for j = 1:num_clusters
        Phi_train(i, j) = exp(-((Train_Data(:,i) - centers(:,j))' * (Train_Data(:,i) -
centers(:,j))) / (2 * sigma_adaptive^2));
    end
end
```

```
%% Compute weights (RBFN Training)
w = Phi_train \ Train_ClassLabel';

%% Construct RBF Kernel Matrix for Test Data
Phi_test = zeros(M, num_clusters);
for i = 1:M
    for j = 1:num_clusters
        Phi_test(i, j) = exp(-((Test_Data(:,i) - centers(:,j))' * (Test_Data(:,i) -
centers(:,j))) / (2 * sigma_adaptive^2));
    end
end

%% Compute Predicted output
TrPred = Phi_train * w;
TePred = Phi_test * w;

%% Compute classification accuracy for different thresholds
TrAcc = zeros(1,1000);
TeAcc = zeros(1,1000);
thr = linspace(min(TePred), max(TePred), 1000);
TrN = length(Train_ClassLabel);
TeN = length(Test_ClassLabel);

for i = 1:1000
    t = thr(i);
    TrAcc(i) = (sum(Train_ClassLabel(TrPred<t)==0) + sum(Train_ClassLabel(TrPred>=t)==1))
/ TrN;
    TeAcc(i) = (sum(Test_ClassLabel(TePred<t)==0) + sum(Test_ClassLabel(TePred>=t)==1)) /
TeN;
end

%% Plot Threshold vs. Accuracy
figure;
plot(thr,TrAcc,'.- ',thr,TeAcc,'^-');legend('Train Accuracy','Test Accuracy');
xlabel('Threshold');
ylabel('Accuracy');
title('K-Means Clustering with RBFN (2 Centers)');
grid on;

%% Compute Mean Images for Each Class (0-9)
unique_classes = unique(train_classlabel);
num_classes = length(unique_classes);
mean_images = zeros(size(train_data,1), num_classes); % Store mean of each class
```

```
for i = 1:num_classes
    class = unique_classes(i);
    mean_images(:, i) = mean(train_data(:, train_classlabel == class), 2);
end

%% Visualize the Mean Images of Each Class (0-9)
figure;
for i = 1:num_classes
    subplot(2,5,i);
    imshow(reshape(mean_images(:,i), 28, 28), []);
    title(['Mean of Class ', num2str(unique_classes(i))]);
end
sgtitle('Mean Images of Each Class (0-9)');

%% Visualize the Centers and Class Means for 7 & 1
% Compute mean images for class 7 and 1
mean_class1 = mean_images(:, unique_classes == class1);
mean_class2 = mean_images(:, unique_classes == class2);

% Compute Euclidean distance between cluster centers and mean images
dist1 = norm(centers(:,1) - mean_class1);
dist2 = norm(centers(:,1) - mean_class2);

% Assign correct labels to cluster centers
if dist1 < dist2
    center1_label = class1;
    center2_label = class2;
else
    center1_label = class2;
    center2_label = class1;
end

%% Visualize K-Means Centers Compared to Class Means
figure;
subplot(2,2,1);
imshow(reshape(mean_class1, 28, 28), []);
title(['Mean of Class ', num2str(class1)]);

subplot(2,2,2);
imshow(reshape(mean_class2, 28, 28), []);
title(['Mean of Class ', num2str(class2)]);

subplot(2,2,3);
imshow(reshape(centers(:,2), 28, 28), []);
```

```
title(['K-Means Center for ', num2str(center2_label)]);

subplot(2,2,4);
imshow(reshape(centers(:,1), 28, 28), []);
title(['K-Means Center for ', num2str(center1_label)]);

sgtitle('Comparison of K-Means Centers with Class Means');
```

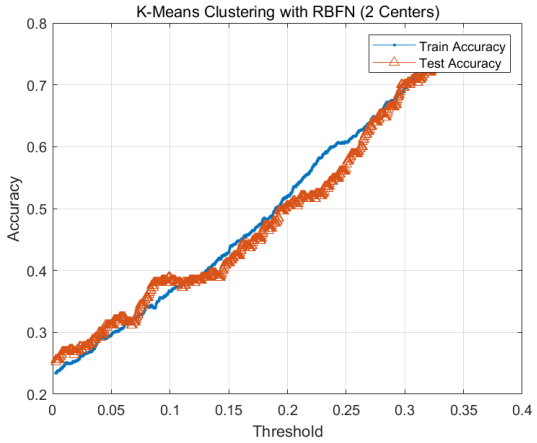

Fig. 8. K-Mean Clustering

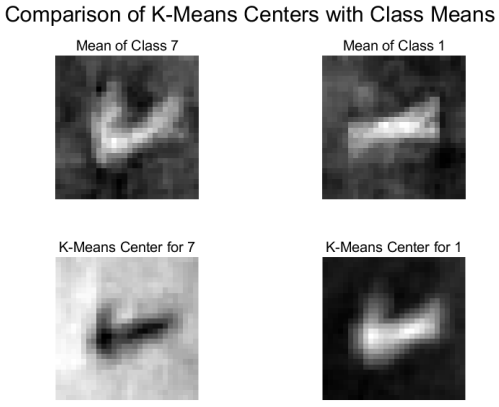

Fig. 9. Comparison of K-Means Centers with Class means (Class 7 and 1)

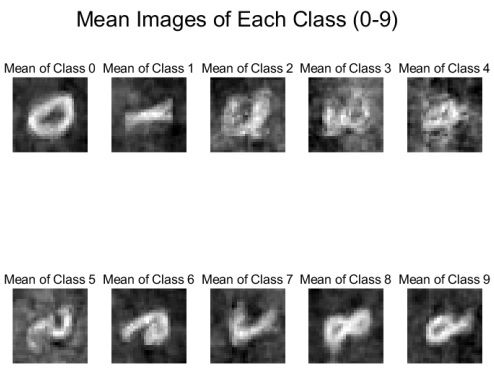

Fig. 10. Mean Images of Each Class (0-9)

K-Means clustering is used to select the center points of RBF neurons, and these centers are then used to compute the response of the radial basis kernel function. However, in this task, we only use two RBF kernels. When the number of kernels is too small, the model struggles to adequately capture the data distribution. Since RBFNs require a sufficient number of centers to form an appropriate decision boundary, having only two RBF kernels limits the classification accuracy of the model.
From Figure 8, we can observe that as the threshold increases, the model's accuracy gradually improves. This indicates that the model's output distribution is relatively smooth, without drastic changes in decision boundaries.
Observing Figure 9, we can see that the centers obtained by K-Means clustering share similar structures with the class means. This indicates that K-Means can effectively identify representative centers of the data. By using the K-Means Clustering method, we can effectively avoid issues such as overfitting, an excessive number of hidden neurons, and the need for a large training dataset to achieve good classification performance.

Q3. Self-Organizing Map (SOM)

a). Write your own code to implement a SOM that maps a 1-dimensional output layer of 40 neurons to a “hat” (sinc function). Display the trained weights of each output neuron as points in a 2D plane, and plot lines to connect every topological adjacent neurons (e.g. the 2nd neuron is connected to the 1st and 3rd neuron by lines).

b). Write your own code to implement a SOM that maps a 2-dimensional output layer of 64 (i.e. 8×8) neurons to a “circle”. Display the trained weights of each output neuron as a point in the 2D plane, and plot lines to connect every topological adjacent neurons (e.g. neuron (2,2) is connected to neuron (1,2) (2,3) (3,2) (2,1) by lines).

c). Write your own code to implement a SOM that clusters and classifies handwritten digits. The training data is provided in **Digits.mat**. The dataset consists of images in 5 classes, namely 0 to 4. Each image with the size of 28*28 is reshaped into a vector and stored in the Digits.mat file. After loading the mat file, you may find the 4 matrix/arrays, which respectively are *train_data*, *train_classlabel*, *test_data* and *test_classlabel*. There are totally 1000 images in the training set and 100 images in the test set. Please omit 2 classes according to the last digit of your matric number with the following rule:

omitted class1 = mod(the last digit, 5), omitted_class2 = mod(the last digit+1, 5). For example, if your matric number is A0642347, ignore classes mod(7,5)=2 and mod(8,5)=3; A1234569, ignore classes 4 and 0. Thus, you need to train a model for a 3-classes classification task. After loading the data, complete the following tasks:

c-1). Print out corresponding conceptual/semantic map of the trained SOM (as described in page 24 of lecture six) and visualize the trained weights of each output neuron on a 10×10 map (a simple way could be to reshape the weights of a neuron into a 28×28 matrix, i.e. dimension of the inputs, and display it as an image). Make comments on them, if any.

c-2) Apply the trained SOM to classify the test images (in test_data). The classification can be done in the following fashion: input a test image to SOM, and find out the winner neuron; then label the test image with the winner neuron's label (note: labels of all the output neurons have already been determined in c-1). Calculate the classification accuracy on the whole test set and discuss your findings.

A3.

a). The code to implement a SOM that maps a 1-dimensional output layer of 40 neurons to a "hat" (sinc function) is as follows:

```
clc; clear; close all;

%% Generate training data (sinc function)
x = linspace(-pi, pi, 400);
trainX = [x; sinc(x)];
num_neurons = 40; % 1D SOM 40 neurons

%% Initialize the weights of 40 neurons (randomly distributed)
w = rand(2, num_neurons) * 2 * pi - pi; % Randomly initialize weights in the range [-π, π]
N = 500; % Number of training iterations
idx_plot = [10, 20, 50, 100:100:N]; % Plot when the appropriate number of iterations is reached
r = 1; % Index of idx_plot
initial_lr = 0.1; % Initial learning rate
sigma0 = num_neurons / 2; % Initial neighborhood width
tau = N / log(sigma0); % Time constant

%% Plot the initial state (iterations = 0)
figure;
hold on;
plot(trainX(1,:), trainX(2,:), '.r', 'MarkerSize', 8); % Plot the original function
plot(w(1,:), w(2,:), 'bo-', 'LineWidth', 2, 'MarkerSize', 6); % Plot SOM neurons
```

```
for j = 1:num_neurons-1
    plot([w(1,j), w(1,j+1)], [w(2,j), w(2,j+1)], 'k-', 'LineWidth', 1.5); % Connect
adjacent SOM neurons
end

xlabel('x');
ylabel('sinc(x)');
title('SOM Mapping of 40 Neurons to sinc Function, Iteration 0');
grid on;
hold off;

%% SOM training and plot mapping results
for t = 1:N
    idx = randi(size(trainX, 2));
    sample = trainX(:, idx); % Randomly select a training sample

    distances = sum((w - sample).^2, 1); % Calculate the Euclidean distance between all
neurons and the sample
    [~, bmu_idx] = min(distances); % Find the best matching neuron

    % Calculate the current learning rate and neighborhood size
    lr = initial_lr * exp(-t / N);
    sigma = sigma0 * exp(-t / tau);

    % Update the weights of all neurons
    for j = 1:num_neurons
        % Calculate the topological distance from the current neuron to the BMU
        d = abs(j - bmu_idx);

        % Calculate the neighborhood function
        h = exp(-d^2 / (2 * sigma^2));

        w(:, j) = w(:, j) + lr * h * (sample - w(:, j));
    end

    if r <= length(idx_plot) && t == idx_plot(r)
        figure;
        hold on;
        plot(trainX(1,:), trainX(2,:), '.r', 'MarkerSize', 8);
        plot(w(1,:), w(2,:), 'bo-', 'LineWidth', 2, 'MarkerSize', 6);
        for j = 1:num_neurons-1
            plot([w(1,j), w(1,j+1)], [w(2,j), w(2,j+1)], 'k-', 'LineWidth', 1.5);
        end
        xlabel('x');
```

```
        ylabel('sinc(x)');
        title(['SOM Mapping of 40 Neurons to sinc Function, Iteration ', num2str(t)]);
        grid on;
        hold off;
        r = r + 1;
    end
end
```

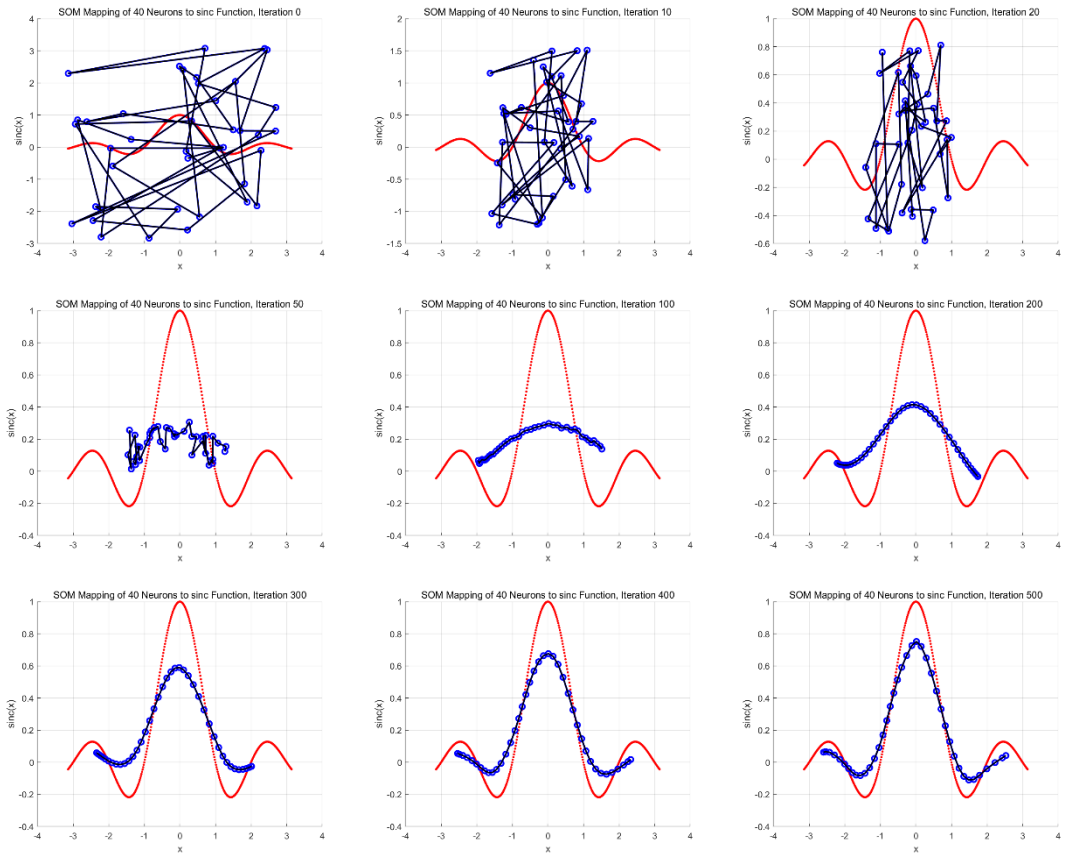

Fig. 11. SOM Process with Different Iterations (sinc function)

Figure 11 illustrates the changes in neuron positions as the number of iterations increases. The first image shows the distribution of weights after random initialization. In the initial stage of training, the neuron positions are disordered, and the topological structure is chaotic, with neurons not yet adapted to the data distribution. As training progresses, the arrangement of SOM neurons gradually becomes more structured and converges to the shape of the sinc function while maintaining the continuity of the topological structure. The topological relationships also become smoother over time.

b). The code to implement a SOM that maps a 2-dimensional output layer of 64 neurons to a “circle” is as follows:

```
clc; clear; close all;
```

```
%% Generate training data
X = randn(800,2);
s2 = sum(X.^2,2);
trainX = (X .* repmat(1*(gammainc(s2/2,1).^(1/2))./sqrt(s2),1,2))';

%% SOM initialization
grid_size = [8, 8]; % 8×8 grid for output layer
num_neurons = prod(grid_size); % 64 neurons in total, prod is used to calculate the
product of each element in grid_size
[grid_x, grid_y] = meshgrid(1:grid_size(1), 1:grid_size(2));
neuron_pos = [grid_x(:), grid_y(:)]; % Store the grid coordinates of each neuron
w = rand(2, num_neurons) * 2 - 1;  % Randomly initialize weights in the range [-1, 1]
N = 500; % Number of training iterations
initial_lr = 0.1; % Initial learning rate
sigma0 = max(grid_size) / 2; % Initial neighborhood width
tau1 = N / log(sigma0); % Time constant τ1
tau2 = N; % Time constant τ2
idx_plot = [10, 20, 50, 100:100:N];
r = 1; % Index of idx_plot

%% Plot the initial state (iterations = 0)
figure;
hold on;
plot(trainX(1,:), trainX(2,:), '.r', 'MarkerSize', 6); % Plot the original data
plot(w(1,:), w(2,:), 'bo', 'MarkerSize', 6, 'LineWidth', 2); % Plot SOM neurons

for i = 1:grid_size(1) % Plot neurons in horizontal direction
    for j = 1:grid_size(2)-1
        idx1 = sub2ind(grid_size, i, j);
        idx2 = sub2ind(grid_size, i, j+1);
        plot([w(1,idx1), w(1,idx2)], [w(2,idx1), w(2,idx2)], 'k-', 'LineWidth', 1.5);
    end
end

for i = 1:grid_size(1)-1 % Plot neurons in vertical direction
    for j = 1:grid_size(2)
        idx1 = sub2ind(grid_size, i, j);
        idx2 = sub2ind(grid_size, i+1, j);
        plot([w(1,idx1), w(1,idx2)], [w(2,idx1), w(2,idx2)], 'k-', 'LineWidth', 1.5);
    end
end

title('SOM Training at Iteration 0');
axis equal;
```

```
grid on;
hold off;

%% SOM training and plot mapping results
for t = 1:N
    idx = randi(size(trainX, 2));
    sample = trainX(:, idx); % Randomly select a training sample

    distances = sum((w - sample).^2, 1); % Calculate the Euclidean distance between all
neurons and the sample
    [~, bmu_idx] = min(distances); % Find the best matching neuron

    % Calculate the current learning rate and neighborhood size
    lr = initial_lr * exp(-t / tau2);
    sigma = sigma0 * exp(-t / tau1);

    % Calculate the Euclidean distance from all neurons to the BMU
    neuron_distances = sum((neuron_pos - neuron_pos(bmu_idx, :)).^2, 2);

    h = exp(-neuron_distances / (2 * sigma^2))'; % Calculate the neighborhood function
    w = w + lr * h .* (sample - w); % Update the weights of all neurons

    if r <= length(idx_plot) && t == idx_plot(r)
        figure;
        hold on;
        plot(trainX(1,:), trainX(2,:), '.r', 'MarkerSize', 6);
        plot(w(1,:), w(2,:), 'bo', 'MarkerSize', 6, 'LineWidth', 2);

        for i = 1:grid_size(1)
            for j = 1:grid_size(2)-1
                idx1 = sub2ind(grid_size, i, j);
                idx2 = sub2ind(grid_size, i, j+1);
                plot([w(1,idx1), w(1,idx2)], [w(2,idx1), w(2,idx2)], 'k-', 'LineWidth',
1.5);
            end
        end

        for i = 1:grid_size(1)-1
            for j = 1:grid_size(2)
                idx1 = sub2ind(grid_size, i, j);
                idx2 = sub2ind(grid_size, i+1, j);
                plot([w(1,idx1), w(1,idx2)], [w(2,idx1), w(2,idx2)], 'k-', 'LineWidth',
1.5);
            end
```

```
        end

        title(['SOM Training at Iteration ', num2str(t)]);
        axis equal;
        grid on;
        hold off;
        r = r + 1;
    end
end
```

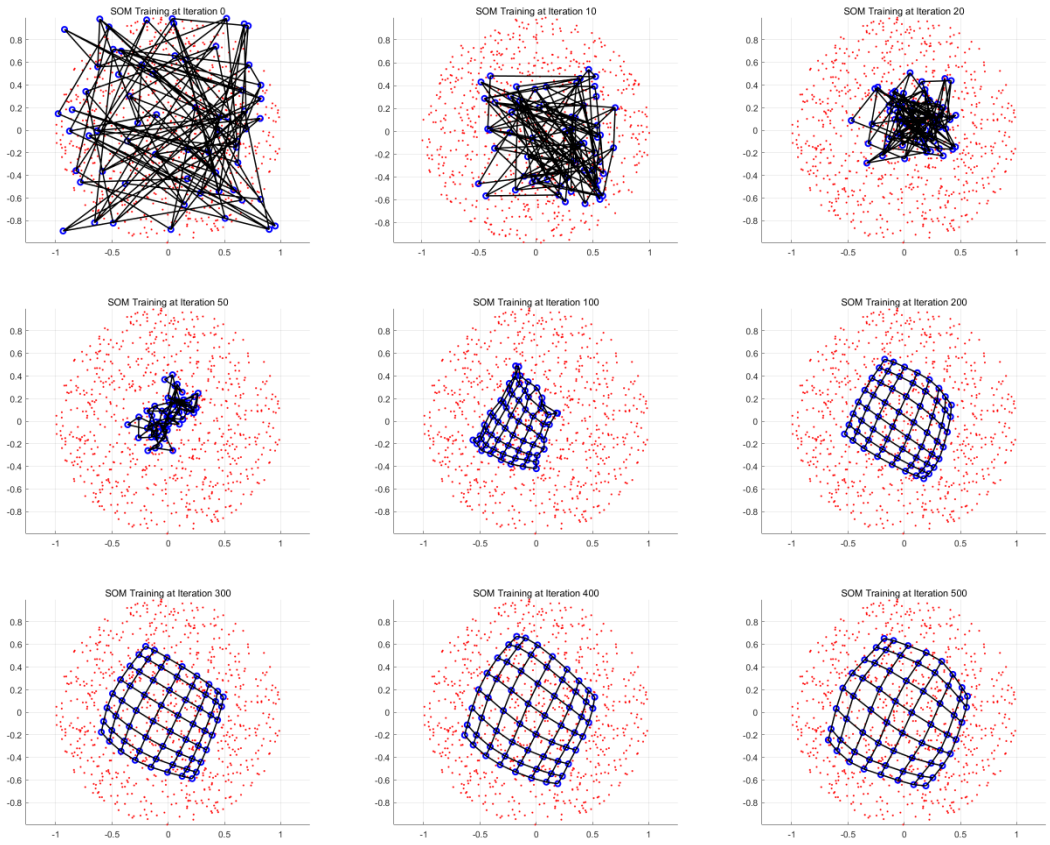

Fig. 12. SOM Process with Different Iterations (circle)

Figure 12 illustrates the changes in neuron positions as the number of iterations increases. The first image shows the distribution of weights after random initialization. In the initial stage of training, the neuron positions are disordered, and the topological structure is chaotic, with neurons not yet adapted to the data distribution. As training progresses, the neurons begin to expand towards the edge of the unit circle, gradually adapting to the topology of the data. They progressively arrange into a regularly shaped grid while maintaining their topological relationships, ultimately conforming to the circular shape of the data.

c). My matric number is A0313771H, the last digit is 1. Omitted class1 = mod(1, 5) = 1, omit class2 = mod(1+1, 5) = 2, so class 1 and 2 are omitted. The code to train SOM, plot corresponding conceptual map, visualize the weights and plot the accuracy curve is as follows:

```
clear; clc; close all;

%% Load the Digits dataset
load('Digits.mat')

%% Select training and test data (Metric number: A0313771H) (Omit class 1 and 2)
Train_idx = find(train_classlabel == 0 | train_classlabel == 3 | train_classlabel == 4);
Train_data = train_data(:, Train_idx);
Train_label = train_classlabel(Train_idx);

Test_idx = find(test_classlabel == 0 | test_classlabel == 3 | test_classlabel == 4);
Test_data = test_data(:, Test_idx);
Test_label = test_classlabel(Test_idx);

Train_Number = size(Train_data, 2);
Test_Number = size(Test_data, 2);

%% SOM initialization
N = 1000; % Number of training iterations
t = 0; % Current iterations
idx_record = [0, 10, 20, 50, 100:100:N];
r = 1; % Index of idx_plot
num_neurons = 100; % 2D SOM 100 neurons
w = rand(784, num_neurons);
sigma0 = num_neurons / 2; % Initial neighborhood width
initial_lr = 0.1; % Initial learning rate
tau = N / log(sigma0); % Time constant
Train_accuracy = zeros(1, size(idx_record, 2));
Test_accuracy = zeros(1, size(idx_record, 2));

%% SOM training
while t <= N
    idx = randi(Train_Number);
    sample = Train_data(:, idx); % Randomly select a training sample

    distances = sum((w - sample).^2, 1); % Calculate the Euclidean distance between all
neurons and the sample
    [~, bmu_idx] = min(distances); % Find the best matching neuron

    % Calculate the current learning rate and neighborhood size
    sigma = sigma0 * exp(-t / tau);
    lr = initial_lr * exp(-t / N);
```

```
    for i = 1 : num_neurons
        % Calculate the topological distance from the current neuron to the BMU
        d = (fix((i - 1) / 10) - fix((bmu_idx - 1) / 10)) ^ 2 + (mod(i - 1, 10) -
mod(bmu_idx - 1, 10)) ^ 2;
        h = exp(-d^2 / (2 * sigma^2)); % Calculate the neighborhood function
        w(:, i) = w(:, i) + lr * h * (sample - w(:, i)); % Update the weights
    end

    if t == idx_record(r)
        % Count the number of times each neuron is activated by samples of different
classes
        vote = zeros(5, num_neurons);
        for i = 1 : Train_Number
            sample = Train_data(:, i); % Get a training sample
            [~, bmu_idx] = min(sum((sample - w) .^ 2)); % Find the best matching neuron
            vote(Train_label(i) + 1, bmu_idx) = vote(Train_label(i) + 1, bmu_idx) + 1; %
Vote counts + 1
        end

        neurons_label = zeros(1, num_neurons);
        neurons_val = zeros(1, num_neurons);

        for i = 1 : num_neurons
            [val, bmu_idx] = max(vote(:, i)); % Find the classes with the most votes
            neurons_label(i) = bmu_idx - 1; % Record the label of the neurons
            neurons_val(i) = val; % Record the votes
        end

        % Calculate the accuracy of the test set
        for i = 1 : Test_Number
            sample = Test_data(:, i);
            [~, bmu_idx] = min(sum((sample - w) .^ 2));
            Test_accuracy(r) = Test_accuracy(r) + (neurons_label(bmu_idx) ==
Test_label(i));
        end

        % Calculate the accuracy of the training set
        for i = 1 : Train_Number
            sample = Train_data(:, i);
            [~, bmu_idx] = min(sum((sample - w) .^ 2));
            Train_accuracy(r) = Train_accuracy(r) + (neurons_label(bmu_idx) ==
Train_label(i));
        end
```

```
        Test_accuracy(r) = Test_accuracy(r) / Test_Number;
        Train_accuracy(r) = Train_accuracy(r) / Train_Number;
        r = r + 1;
    end

    t = t + 1;
end

%% Plot the results

% Weights
Trained_weights = [];

for i = 0 : 9
    Weights_row = [];

    for j = 1 : 10
        Weights_row = [Weights_row, reshape(w(:, i*10+j), 28, 28)];
    end

    Trained_weights = [Trained_weights; Weights_row];
end

figure;
imshow(imresize(Trained_weights, 3));
title('Weights Visualization');


% Conceptual map
neurons_label = reshape(neurons_label, 10, 10)';
neurons_val = neurons_val/max(neurons_val); % Store the confidence of each neuron
(normalized vote counts)
neurons_val = reshape(neurons_val, [10,10])';

figure;
img = imagesc(neurons_label);
img.AlphaData = neurons_val;
% Set the transparency. The smaller the value of neurons_val, the more transparent it
is, meaning that the neuron has fewer votes.

for i = [0, 3, 4]
    neurons_label(neurons_label == i) = num2str(i);
end
```

```
label = num2str(neurons_label, '%s');
[x, y] = meshgrid(1:10);
hStrings = text(x(:), y(:), label(:), 'HorizontalAlignment', 'center'); % Label each
neuron's class
title('Conceptual Map');

% Accuracy curves of training set and test set
figure;
hold on;
plot(idx_record, Train_accuracy, 'linewidth', 2);
plot(idx_record, Test_accuracy, 'linewidth', 2);
hold off;
legend('Train Accuracy','Test Accuracy', 'Location', 'southeast');
xlabel('Iterations')
ylabel('Accuracy')
title('Accuracy Curves of Training Set and Test Set');
```

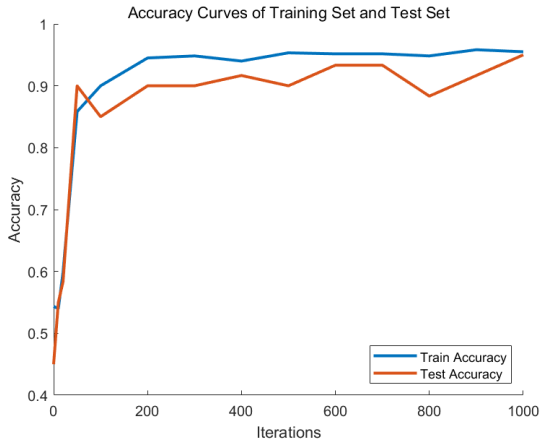

Fig. 13. Accuracy curve of Training Set and Test Set

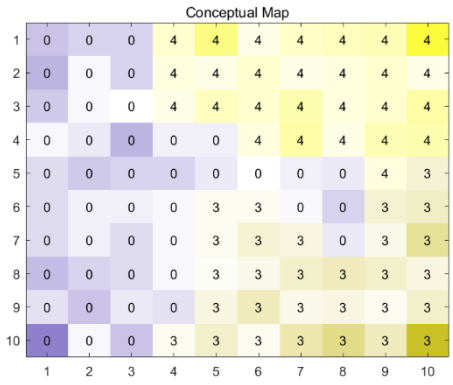

Fig. 14. Conceptual Map

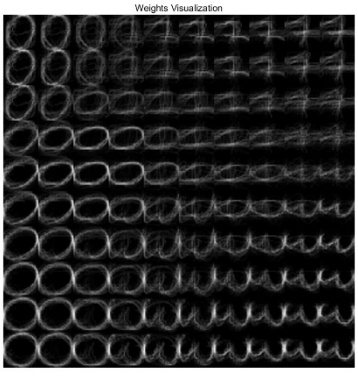

Fig. 15. Weights Visualization

Figure 14 presents the conceptual map formed by the SOM after training. It shows that the training data is classified into three classes: Class 0, Class 3, and Class 4, which aligns with the requirements of the

task. Within the same class, the darker the color block, the more confident the model is in classifying that handwritten digit; conversely, lighter blocks indicate lower confidence in classification.

Figure 15 visualizes the corresponding neuron weights from Figure 14, representing the feature mappings learned by each neuron. By comparing the two figures, it can be observed that the model has effectively adjusted the weights through unsupervised learning, successfully capturing the characteristic patterns of different digits. As a result, the classification outcome is accurate.

Figure 13 shows the classification accuracy of the training and test sets. It can be observed that the SOM training process is relatively stable, with both training and test accuracy converging after 200 iterations. This indicates that the SOM network has effectively adapted to the data distribution, learned reasonable classification boundaries, and demonstrated strong generalization capability.