Course EE5904 Neural Networks

Assignment 3 Report

Lecturer: **Assoc. Prof. Xiang Cheng**

Author: XU YIMIAN
Student ID: A0295779Y

National University of Singapore

Mar 30, 2025

Contents

List of Tables

List of Figures

Q1. Function approximation with RBFN

(a) Exact Interpolation Method

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.

Solution:

The goal of exact interpolation method is to find a function $f(x)$ that passes through all the training data points. The centers of the RBFs are the data points x_i, and the basis functions are Gaussian functions with standard deviation of 1.0. The outputs for each sampling points: $f(x_j) = \sum_{i=1}^{N} w_i \varphi(\| x_j - x_i \|) = d_j$, j=1,2,...,N, which can be simplified as $\Phi w = d$. Provided the inverse of Φ exists, the unique solution can be computed by: $w = \Phi^{-1} d$. Using MATLAB, to solve this equation, we can use $w = \Phi \backslash d$. Then the prediction values can be computed by: $\hat{y} = \Phi w$.

Listing 1: MATLAB script (Q1a)

```
% Q1a: Exact Interpolation Method
% generate training data:
x_train = -1.6:0.08:1.6;
y_train = 1.2*sin(pi*x_train)-cos(2.4*pi*x_train)+0.3*randn(size(x_train));

% compute RBF matrix:
sigma = 0.1;
centers = x_train; % number of hidden units = number of data points
Phi = exp(-pdist2(x_train', centers').^2/(2*sigma^2));

% solve the weights:
w = Phi\y_train'; % exact interpolation Φ( * w = y_train), w = Φ^(-1)*y_train

% test set evaluation:
x_test = -1.6:0.01:1.6;
Phi_test = exp(-pdist2(x_test', centers').^2/(2*sigma^2));
y_pred = Phi_test * w;
test_mse = mean((y_pred - (1.2*sin(pi*x_test) - cos(2.4*pi*x_test))').^2);

% Visualization
figure('Name','Q1a: Exact Interpolation');
hold on;

% Generate true function values
x_true = -1.6:0.01:1.6;
y_true = 1.2*sin(pi*x_true) - cos(2.4*pi*x_true);

% Plot true function
plot(x_true, y_true, 'b-', 'LineWidth', 1.5, 'DisplayName','True Function');

% Plot noisy training data
scatter(x_train, y_train, 50, 'filled', 'MarkerEdgeColor','k',...
    'MarkerFaceColor','r', 'DisplayName','Noisy Training Data');

% Plot RBFN approximation
plot(x_test, y_pred, '--', 'LineWidth', 2, 'DisplayName','RBFN Approximation');

title('Exact Interpolation Method  (=0.1)');
xlabel('Input x'); ylabel('Output y');
legend('Location','NorthWest');
grid on;
hold off;
```

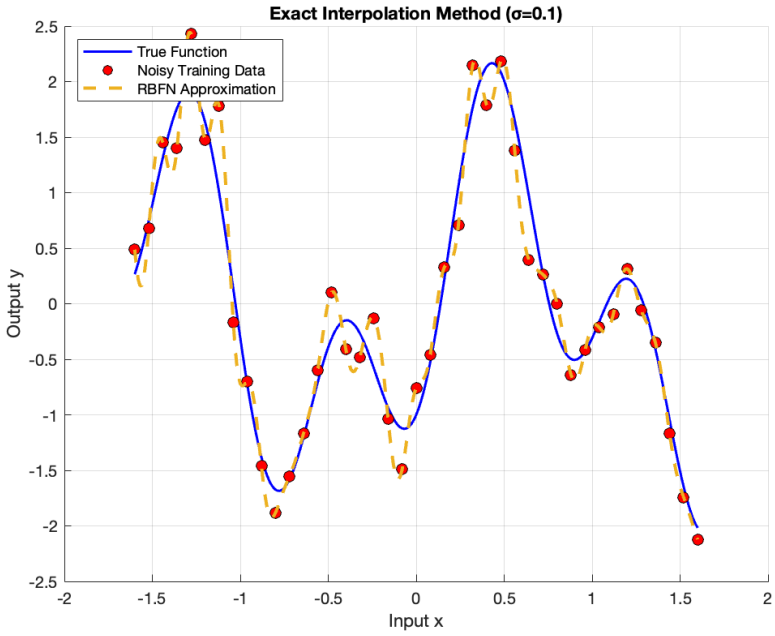


Figure 1: Q1a: Exact Interpolation Method

Results and analysis:
test_mse = 0.0690
The true function vs RBFN approximation are shown in Figure 1, showing the noisy training data points significantly effect the RBFN performance, which is poor due to perfect fitting.

(b) Fixed Centers Selected at Random

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).
Solution:
Steps are as following:
1. Select Centers Randomly from the training data points: $\mu_i \in x_{train}$, i = 1, 2, ..., M, and take them as the centers of RBFs.
2. Compute maximum distance between any two selected centers, $d_{max} = \max_{i,j} \| \mu_i - \mu_j \|$, then set the width of RBF as $\sigma_i = \frac{d_{max}}{\sqrt{2M}}$, which ensures that individual RBFs are neither too peaked nor too flat, reasonably over the data space.
3. Construct the RBF matrix Φ for training set: Φ (N×M) is constructed by $\varphi_j(x_i)$, where $\varphi_j(x_i) = \exp(-\frac{M}{d_{max}^2} \| x_i - \mu_j \|^2)$, j=1,2,...,M, i=1,2,..., N.
4. Compute weights: $w = \Phi \backslash d$.
5. Compute predictions: $\hat{y} = \Phi w$.

Listing 2: MATLAB script (Q1b)

```
%% Q1b: Fixed Centers Selected at Random
% randomly select 20 centers:
M = 20;
idx = randperm(length(x_train), M);
centers_random = x_train(idx);

```

```
% calculate pairwise distances between centers:
pairwise_dist = pdist(centers_random');

% find maximum distance between centers:
d_max = max(pairwise_dist);

% compute sigma using d_max:
sigma = d_max/sqrt(2*M);

% compute RBF matrix:
Phi_rand = exp(-pdist2(x_train', centers_random').^2 / (2*sigma^2));

% solve for weights (Least Square):
w_rand = Phi_rand \ y_train';

% test set evaluation:
Phi_test_rand = exp(-pdist2(x_test', centers_random').^2 / (2*sigma^2));
y_pred_rand = Phi_test_rand * w_rand;
test_mse_rand = mean((y_pred_rand - (1.2*sin(pi*x_test) - cos(2.4*pi*x_test))').^2)

% Visualization:
figure('Name','Q1b: Fixed Centers Comparison');
hold on;

% Plot true function
plot(x_true, y_true, 'b-', 'LineWidth', 1.5, 'DisplayName','True Function');

% Plot both approximations
plot(x_test, y_pred, '--', 'LineWidth', 2, 'DisplayName','Exact Interpolation');
plot(x_test, y_pred_rand, 'g-.', 'LineWidth', 2, 'DisplayName','Random Centers');

% Highlight random centers
scatter(centers_random, zeros(size(centers_random)), 100, 'kpentagram',...
    'filled', 'DisplayName','Selected Centers');

title('Comparison of Approximation Methods');
xlabel('Input x'); ylabel('Output y');
legend('Location','NorthWest');
grid on;
hold off;
```

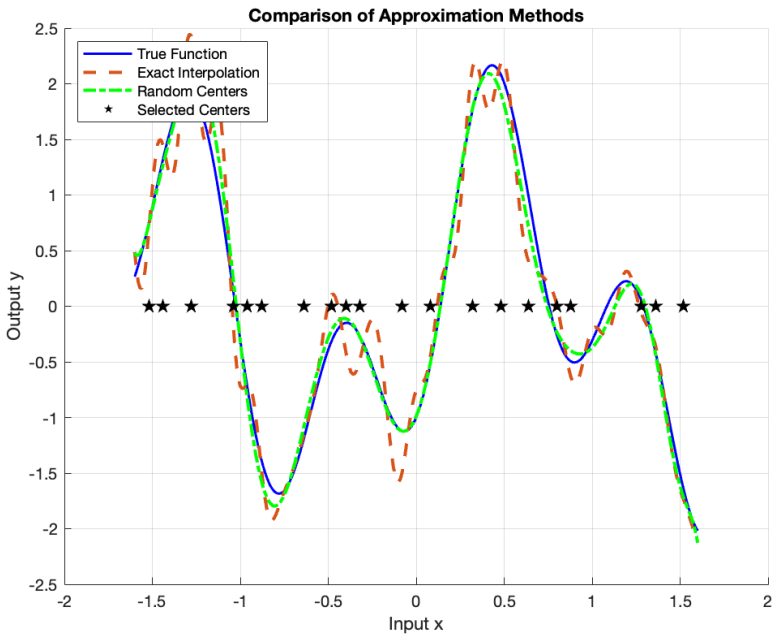


Figure 2: Q1b: Fixed Centers Selected at Random

Results and analysis:
test_mse_rand = 0.0148 (< 0.0690)
Compared to the "Exact Interpolation Method" in Q1(a), "fixed centers selected at random" strategy performs better with a much smaller MSE. As shown in Figure 2, the approximation curve is highly consistent with the true function. Reducing the number of hidden units in RBFN suppresses overfitting by constraining model complexity, thereby achieving robust generalization performance in noise-corrupted function approximation tasks.

(c) Regularization Method

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.
Solution:
With regularization method, we can cope with over-fitting by controlling the smoothness of mapping functions. One simple way is to add a penalty term into the cost function:

$$F(w) = \frac{1}{2}(d - \Phi w)^T (d - \Phi w) + \frac{1}{2}\lambda w^T w$$

Solve the weights by minimizing F(w): $w = (\Phi^T \Phi + \lambda I)^(-1)\Phi^T d$. The influence of the regularization factor λ: Larger $\lambda \rightarrow$ Smaller weights $\rightarrow$ Smoother mapping.

Listing 3: MATLAB script (Q1c)

```
% Regularization factors
lambda_values = [0, 1e-4, 1e-2, 0.1, 1, 10];
test_mses = zeros(size(lambda_values));
y_pred_reg_res = {};

% precompute Phi' * Phi, Phi' * y_train'
term1 = Phi' * Phi;
term2 = Phi' * y_train';

% w = (Phi'*Phi+lambda*I)\(Phi'*d)
for i = 1:length(lambda_values)
    lambda = lambda_values(i);
    w_reg = (term1 + lambda * eye(size(Phi,2))) \ (term2);
    y_pred_reg = Phi_test * w_reg;
    y_pred_reg_res{i} = y_pred_reg;
    test_mses(i) = mean((y_pred_reg - (1.2*sin(pi*x_test) - cos(2.4*pi*x_test))').^2);
    fprintf ("=%f, MSE=%f\n", lambda, test_mses(i));
end

% Visualization
figure('Name','Q1c: Regularization MSE');
semilogx(lambda_values, test_mses, 'bo-', 'LineWidth', 2, 'MarkerSize', 8);
title('Test MSE vs Regularization Factor');
xlabel('Regularization Factor ()');
ylabel('Mean Squared Error (MSE)');
grid on;

% Fitting results
fig1 = figure('Name',"RBFN Approximation (Regularization Method)");
screensize = get(0, 'screensize');
set(gcf, 'position', screensize);

for i = 1:length(lambda_values)
    subplot(2, 3, i);
    % Plot ground truth vs predictions
    plot(x_true, y_true, 'b-', 'LineWidth', 1.5, 'DisplayName','True Function');hold on;
    plot(x_test, y_pred_reg_res{i}, 'g--', 'LineWidth', 2, 'DisplayName','RBFN
        Approximation');
    scatter(x_train, y_train, 50, 'filled', 'MarkerEdgeColor','k',...
```

```
    'MarkerFaceColor','r', 'DisplayName','Noisy Training Data');

    % Figure formatting
    title(sprintf(' =%.4f', lambda_values(i)));
    xlabel('Input x'); ylabel('Output y');
    legend('Location','northeast');
    grid on;
    hold off;
end
```

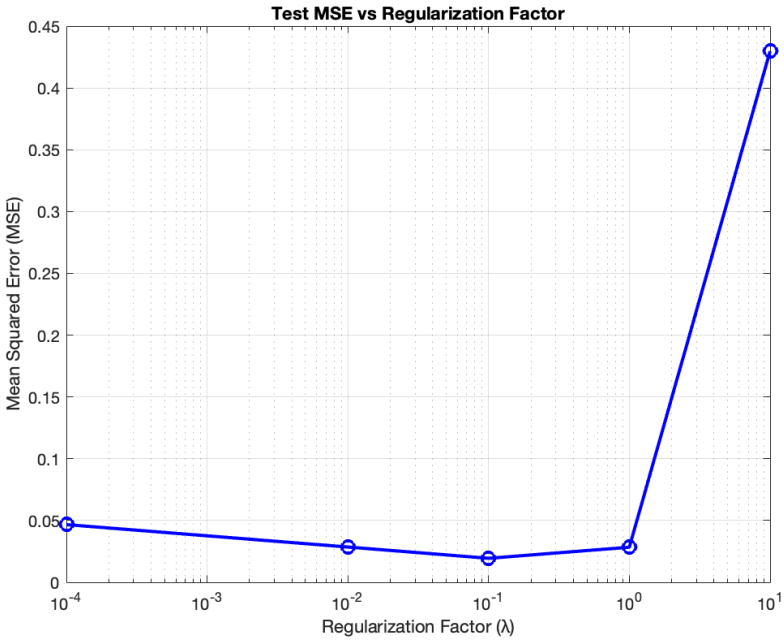


Figure 3: Q1c: MSE (Regularization Method)

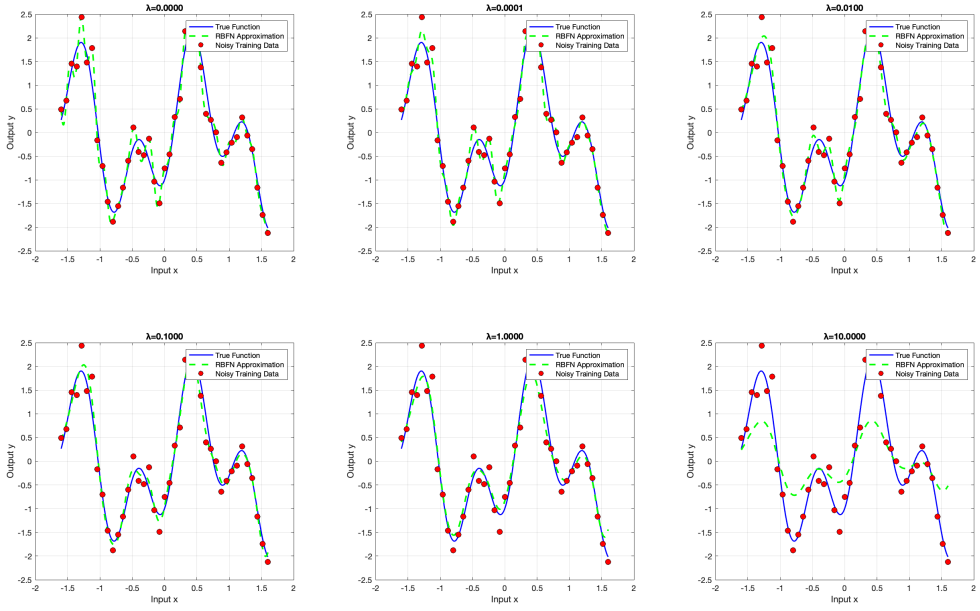


Figure 4: Q1c: Approximation (Regularization Method)

Results and analysis:
From the Table 1 we can see that the optimal regularization factor is λ=0.1, achieving lowest test MSE(0.019). This parameter balances noise suppression and feature preservation. As λ increasing to 1, and then 10, the test MSE becomes larger, indicating the over-smoothing effect of regularization, which means RBFN fails to capture function shape because weight magnitudes are suppressed excessively.

Table 1: Q1c: Comparison of different regularization factors

λ value	Test MSE	Behavior
0	0.069025	Severe overfitting
0.0001	0.046681	Partial noise suppression
0.01	0.028548	Partial noise suppression
0.1	0.019423	Optimal balance
1	0.028384	Beginning of oversmoothing
10	0.429659	Excessive smoothing

Q2. Handwritten Digits Classification using RBFN

My matric number is A0295779Y, so I choose classes 7 and 9. The images in the selected two classes are assigned the label ”1” and the remaining eight classes are asssigned the label ”0”. The sample numbers of different classes in training set and test set are shown in Table 2.

Table 2: Q2a: Sample number of dataset

Label	Train data	Test data	All data
1 (No. 7 and 9)	217	56	273
0 (Others)	783	194	977
Total	1000	250	1250

(a) Exact Interpolation Method and Apply Regularization

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.
Solution:

1. RBFN implement: Centers: All training samples (N =1000), Gaussian Kernel: σ = 100.
2. The weights of RBFN with regularization are computed by: $w = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T d$, same as Q1(c).
3. Performances of RBFNs without regularization and with varying regularization factors are evaluated.
   Threshold Sweeping: 1000 thresholds (τ) between [min(TrPred), max(TrPred)).
   Matrics Calculation:
   Accuracy = (TN + TP)/(TN + FP + FN + TP)

Precision = TP/(TP + FP)
Recall (TPR) = TP/(TP + FN)
FPR = FP/(FP + TN)

Listing 4: MATLAB script (Q2a)

```
% load data and select classes:
load './mnist_m.mat';

% select classes 7 and 9 (based on my matric number) and reassign label as "1";
class1 = 7;
class2 = 9;
Train_Label = double(train_classlabel == class1 | train_classlabel == class2);
Test_Label = double(test_classlabel == class1 | test_classlabel == class2);

Train_Data = train_data;
Test_Data = test_data;

Train_Idx = find(train_classlabel==class1 | train_classlabel==class2);
Test_Idx = find(test_classlabel==class1 | test_classlabel==class2);
class1_num = length(Train_Idx)+length(Test_Idx);

% evaluate performance
function [results, optimal_idx] = evaluate_performance(TrPred, TePred, TrLabel, TeLabel,
    results)
    %This generates 1,000 candidate thresholds spanning the full range of predictions (
        min(TrPred) to max(TrPred))
    num_thresholds= 1000;
    thresholds = linspace(min(TrPred), max(TrPred), num_thresholds+1);
    thresholds= thresholds(1:num_thresholds);

    TrN = length(TrLabel);
    TeN = length(TeLabel);

    for i = 1:num_thresholds
        % Predict Class 0 if the output < t, Class 1 if the output   t
        t = thresholds(i);

        % Training set metrics
        TrPredClass = TrPred >= t;
        TN = sum(~TrPredClass & ~TrLabel');
        FP = sum(TrPredClass & ~TrLabel');
        FN = sum(~TrPredClass & TrLabel');
        TP = sum(TrPredClass & TrLabel');
        % Test set metrics
        TePredClass = TePred >= t;
        teTN = sum(~TePredClass & ~TeLabel');
        teFP = sum(TePredClass & ~TeLabel');
        teFN = sum(~TePredClass & TeLabel');
        teTP = sum(TePredClass & TeLabel');
        % Store results
        results.Thr(i)=t;
        results.TrAcc(i) = (TN + TP) / TrN;
        results.TeAcc(i) = (teTN + teTP) / TeN;
        results.ConfusionMatrices{i} = struct(...
            'Train', [TN FP; FN TP], ...
            'Test', [teTN teFP; teFN teTP]...
        );
    end

    % Find optimal threshold index
    [maxTeAcc, optimal_idx] = max(results.TeAcc);

    % Get optimal confusion matrices
    optimal_cm = results.ConfusionMatrices{optimal_idx};

    % Additional metrics display
    fprintf('Max TeAcc: %.4f%%\n', maxTeAcc*100);
    fprintf('Optimal Threshold: %.4f\n', thresholds(optimal_idx));

```

```
    fprintf('Training Precision: %.2f%%, Recall (TPR): %.2f%%, FPR: %.2f%%\n', ...
    100*optimal_cm.Train(2,2)/(optimal_cm.Train(2,2)+optimal_cm.Train(1,2)+eps), ...
    100*optimal_cm.Train(2,2)/(optimal_cm.Train(2,2)+optimal_cm.Train(2,1)+eps), ...
    100*optimal_cm.Train(1,2)/(optimal_cm.Train(1,2)+optimal_cm.Train(1,1)+eps));

    fprintf('Test Precision: %.2f%%, Recall (TPR): %.2f%%, FPR: %.2f%%\n', ...
    100*optimal_cm.Test(2,2)/(optimal_cm.Test(2,2)+optimal_cm.Test(1,2)), ...
    100*optimal_cm.Test(2,2)/(optimal_cm.Test(2,2)+optimal_cm.Test(2,1)), ...
    100*optimal_cm.Test(1,2)/(optimal_cm.Test(1,2)+optimal_cm.Test(1,1)));

    % Plot confusion matrices
    class_names = {'Class 0', 'Class 1'};
    plot_confusion_matrix(optimal_cm, class_names);
end

function plot_confusion_matrix(cm, class_names)
    figure('Position', [100 100 600 250]);

    % Training confusion matrix
    subplot(1,2,1);
    confusionchart(cm.Train, class_names);
    title('Training Set Confusion Matrix');

    % Test confusion matrix
    subplot(1,2,2);
    confusionchart(cm.Test, class_names);
    title('Test Set Confusion Matrix');

    % Formatting
    colormap(parula);
    set(gcf, 'Color', 'w');
end

% build RBFN with sigma=100
sigma = 100;
lambda_values = [0, 1e-2, 0.1, 1, 10];

centers = Train_Data';  % Centers = training samples
Phi_train = exp(-pdist2(Train_Data', centers).^2 / (2*sigma^2));
Phi_test = exp(-pdist2(Test_Data', centers).^2/(2*sigma^2));

% solve with regularization
term1 = Phi_train' * Phi_train;
term2 = Phi_train' * Train_Label';

n =length(lambda_values);
num_thresholds = 1000;
% Preallocate memory
results = struct();
results.Thr = zeros(1, num_thresholds);
results.TrAcc = zeros(1, num_thresholds);
results.TeAcc = zeros(1, num_thresholds);
results.ConfusionMatrices = cell(num_thresholds, 1);
resultsArray = repmat(results, 1, n);
optimal_idx = cell(n);

for i = 1:n
    lambda = lambda_values(i);
    if lambda == 0
        w = Phi_train \ Train_Label'; % non-regularization
    else
        w = (term1 + lambda * eye(size(Phi_train,2)))\term2;
    end
    TrPred = Phi_train*w;
    TePred = Phi_test*w;

    % evaluate performance (different )
    fprintf(' =%.4f\n', lambda);
    [resultsArray(i), optimal_idx{i}] = evaluate_performance(TrPred, TePred, Train_Label,
        Test_Label, resultsArray(i));
    resultsArray(i).lambda = lambda;
end
```

Results and analysis:
λ=0.0000
Max TeAcc: 75.6000%
Optimal Threshold: 18.1818
Training Precision: 66.67%, Recall (TPR): 0.92%, FPR: 0.13%
Test Precision: 30.77%, Recall (TPR): 7.14%, FPR: 4.64%

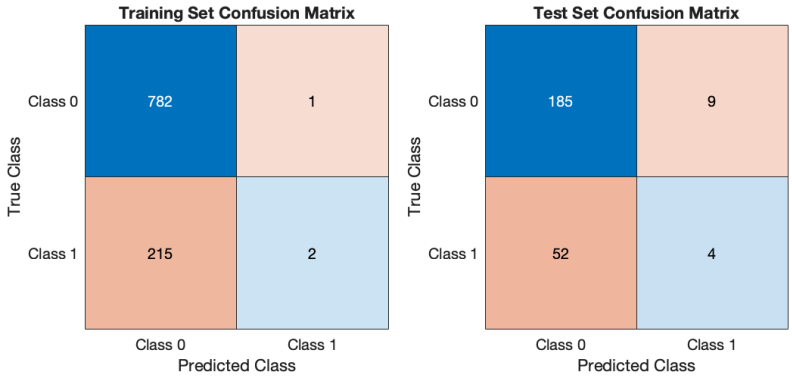


Figure 5: Q2a: Confusion Matrix ($\lambda = 0$)

λ=0.0100
Max TeAcc: 78.0000%
Optimal Threshold: 0.2805
Training Precision: 42.22%, Recall (TPR): 17.51%, FPR: 6.64%
Test Precision: 52.17%, Recall (TPR): 21.43%, FPR: 5.67%

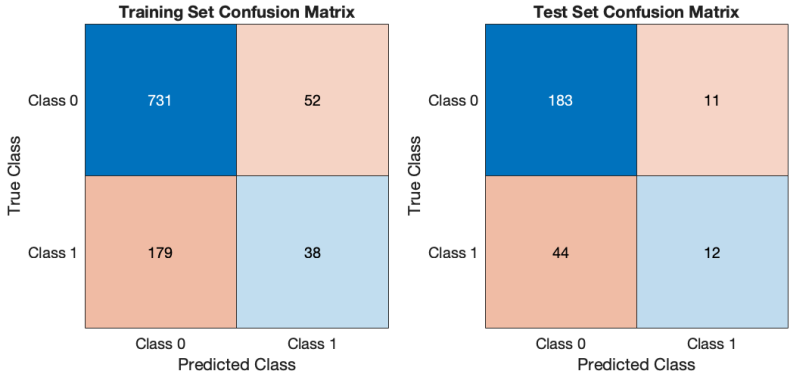


Figure 6: Q2a: Confusion Matrix ($\lambda = 0.01$)

λ=0.1000
Max TeAcc: 80.0000%
Optimal Threshold: 0.2305
Training Precision: 54.46%, Recall (TPR): 25.35%, FPR: 5.87%
Test Precision: 60.71%, Recall (TPR): 30.36%, FPR: 5.67%

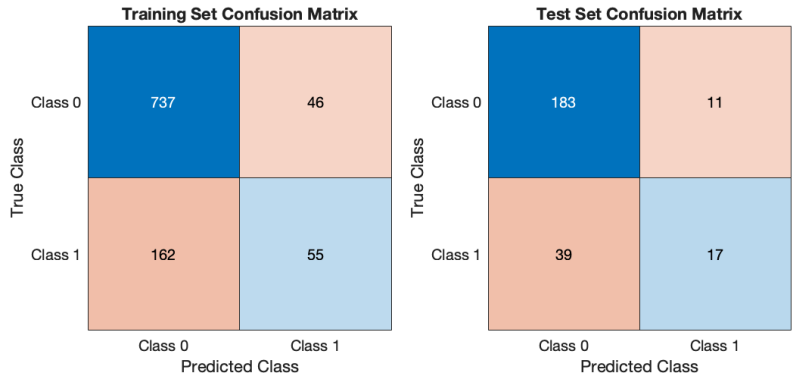


Figure 7: Q2a: Confusion Matrix ($\lambda = 0.1$)

λ=1.0000
Max TeAcc: 78.8000%
Optimal Threshold: 0.2276
Training Precision: 48.28%, Recall (TPR): 6.45%, FPR: 1.92%
Test Precision: 80.00%, Recall (TPR): 7.14%, FPR: 0.52%

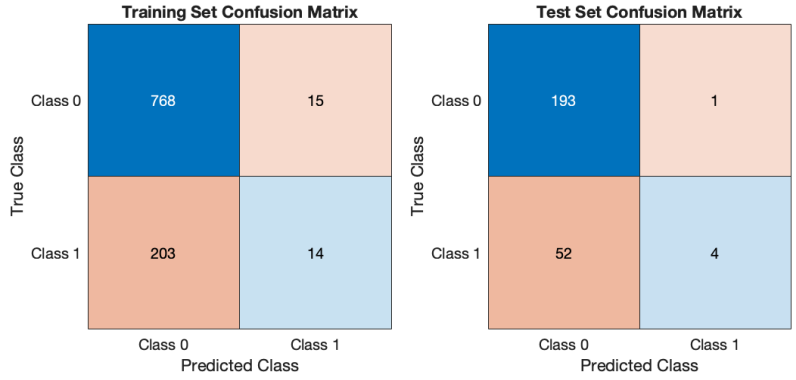


Figure 8: Q2a: Confusion Matrix ($\lambda = 1$)

λ=10.0000
Max TeAcc: 78.4000%
Optimal Threshold: 0.2201
Training Precision: 39.39%, Recall (TPR): 5.99%, FPR: 2.55%
Test Precision: 75.00%, Recall (TPR): 5.36%, FPR: 0.52%

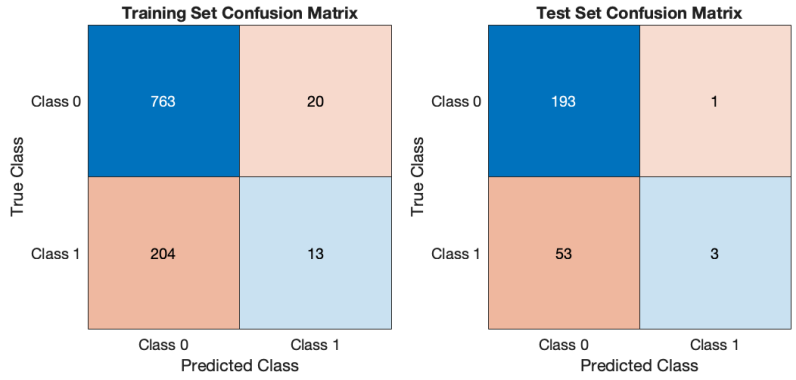


Figure 9: Q2a: Confusion Matrix ($\lambda = 10$)

The **classification accuracy vs threshold curves** with varying λ are shown in Figure 10, presenting the S-shaped curve characteristics: the left end indicates nearly all samples classified as positive (threshold $\tau \leftarrow$min prediction), and the right end indicates nearly all samples classified as negative (threshold $\tau \rightarrow$max prediction), and the slope of transition zone in the middle reflects discriminative power (optimal threshold at maximum curvature).
As the figure shows, the regularization impacts on curve morphology:
1. λ=0 (No regularization): flatter accuracy-threshold curve, showing insensitive threshold (most predictions already clustered at extremes and moving threshold through mid-range barely changes TN/TP counts);
2. λ=0.1 (Optimal): steepest accuracy-threshold transition, indicating calibrated confidence and threshold sensitivity (predictions distributed across decision boundary and small threshold changes significantly alter TN/TP counts).
From the command line outputs, we can also see the impact of class imbalance: optimal threshold shifted right to acquire better accuracy. High test accuracy is achieved by simply predicting all negatives.

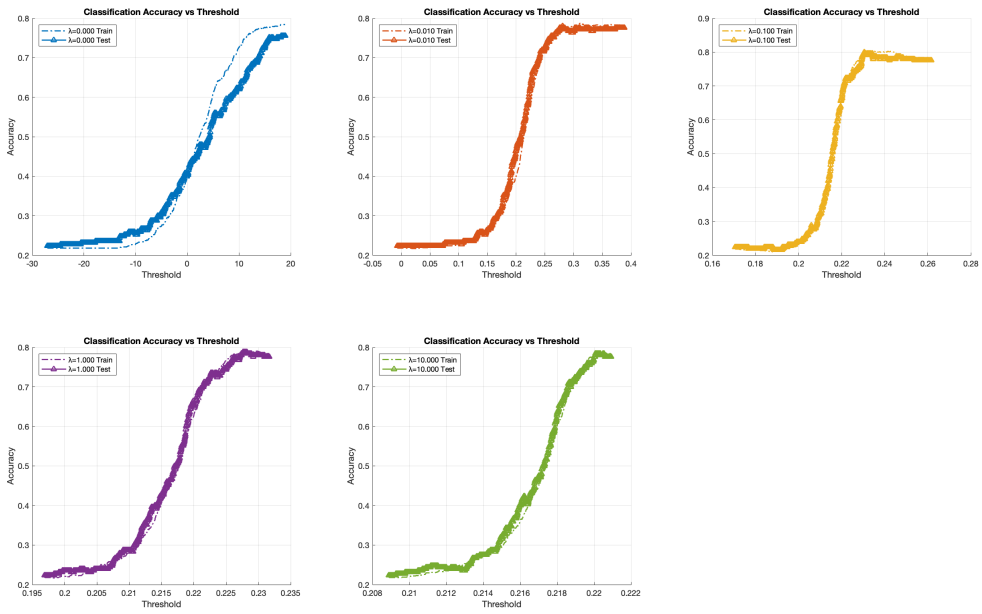


Figure 10: Q2a: Classification Accuracy vs Threshold (changing λ)

The **performance of RBFN** with different λ on the test set are shown in Table 3. The matrices in the table below are calculated using the threshold that provides the maximum test accuracy. Also, extreme class imbalance in the dataset causes an asymmetric precision-recall trade-off, where high recall compromises precision. High precision (80%) at λ =1 comes with 7.14% recall, and recall improvement at λ =0.1 requires precision sacrifice.

Table 3: Q2a: Regularization Effects

λ	Max TeAcc	Te Precision	Te Recall	Te FPR
0	75.60%	30.77%	7.14%	4.64%
0.01	78.00%	52.17%	21.43%	5.67%
0.1	**80.00%**	60.71%	**30.36%**	5.67%
1	78.80%	80.00%	7.14%	0.52%
10	78.40%	75.00%	5.36%	0.52%

The **confusion matrices** of training set and test set using different λ are shown in Figure 5 to 9. We observed that the FPR sensitivity is relatively high, even 5% FPR introduced 10+ false positives due to large negative class.

From all the results, it is easy to tell the **optimal regularization** λ=0.1, achieving the best balance between accuracy and recall rate (80% accuracy, 30% recall). The performance of exact interpolation method without regularization is poor due to overfitting. As λ increases from zero, the precision and recall improve simultaneously, while with bigger λ above 0.1, FPR increases at recall cost, showing that stronger regularization can increase prediction caution and reduce false alarms but sacrifices positive class detection, meaning that the RBFN predicts only high-confidence positive instances, which also explains the lower recall rate.

(b) Fixed Centers Selected at Random

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.

Solution:

Steps are similar to Q1b, the first sigma value is calculated by $\sigma = \frac{d_{max}}{\sqrt{2M}}$. And then, I try σ=0.1, 1, 10, 100, 1000, 10000 respectively, and show the performance by command line outputs and confusion matrices (from Figure 12 to Figure 17). The accuracy vs threshold curves for different sigma values are shown in Figure 18.

Listing 5: MATLAB script (Q2b)

```
%% Part 1: Fixed Centers with Optimal Sigma
% randomly select 100 centers:
rng(2025);% For reproducibility
num_centers = 100;
center_idx = randperm(size(Train_Data,2), num_centers);
centers = Train_Data(:, center_idx)';

% Calculate d_max between selected centers
dist_matrix = pdist(centers); % Pairwise distances
d_max = max(dist_matrix);

% Optimal sigma calculation
sigma_recommended = d_max / sqrt(2*num_centers);

%% Part 2: Sigma Parameter Sweep
% Sigma range setup
sigma_values = [sigma_recommended, logspace(-1, 4, 6)]; % 0.1 to 10000 (6 points)
num_sigmas = length(sigma_values);

resultsArray = repmat(results, 1, num_sigmas);
optimal_idx = cell(num_sigmas);
```

```
22
23 %% Main Loop
24 for s = 1:num_sigmas
25     sigma = sigma_values(s);
26     % Calculate RBF features
27     Phi_train = exp(-pdist2(Train_Data', centers).^2/(2*sigma^2));
28     Phi_test = exp(-pdist2(Test_Data', centers).^2/(2*sigma^2));
29
30     % Solve weights WITHOUT regularization
31     w = Phi_train \ Train_Label';
32
33     % Get predictions
34     TrPred = Phi_train * w;
35     TePred = Phi_test * w;
36
37     % Evaluate performance
38     fprintf(' =%.4f\n', sigma);
39     [resultsArray(s), optimal_idx{s}] = evaluate_performance(TrPred, TePred, Train_Label,
           Test_Label, resultsArray(s));
40     resultsArray(s).sigma = sigma;
41 end
42
43 % Visualization
44 % Plot accuracy curve vs threshold (using different sigma)
45 figure('Name', 'Classification Accuracy vs Threshold (changing )');
46 screensize = get(0, 'screensize');
47 set(gcf, 'position', screensize);
48
49 % Accuracy curve
50 for s = 1:length(resultsArray)
51     subplot(floor(sqrt(num_sigmas)),ceil(num_sigmas/floor(sqrt(num_sigmas))),s);
52     hold on;
53     color_order = lines(num_sigmas);
54     plot(resultsArray(s).Thr,resultsArray(s).TrAcc,'-.','Color', color_order(s,:), '
           LineWidth', 1.5);
55     plot(resultsArray(s).Thr,resultsArray(s).TeAcc,'^-','Color', color_order(s,:), '
           LineWidth', 1.5);
56     legend(sprintf(' =%.2f Train', sigma_values(s)), sprintf(' =%.2f Test', sigma_values(s
           )));
57     title('Classification Accuracy vs Threshold');
58     xlabel('Threshold');
59     ylabel('Accuracy');
60     legend('Location','northwest');
61     grid on;
62     axis auto;
63     hold off;
64 end
```

Results and analysis:
Key observations and interpretations are as follows: Part 1: Analysis of the accuracy-threshold curve for σ=1.4227

1. **Curve Behavior Interpretation**

   - **Threshold < 0 (Accuracy ≈ 20%)**: The model predicts all samples as **class 1 (positive)**. Since class 1 constitutes only 22.4% of the test set (56/250), accuracy is dominated by the majority class (class 0). This reflects a failure mode where the model defaults to predicting the minority class but achieves low true positive rate (TPR=3.57%).
   - **Threshold Near 0 (Stepwise Accuracy Increase)**: A slight increase in threshold begins to classify some samples as negative. True negatives (TN) rise rapidly, causing accuracy to jump to about 78%.
   - **Threshold > 0.25 (Accuracy Stabilizes at nearly 78%)**: The model predicts most samples as negative, aligning accuracy with the majority class prevalence (77.6%). This indicates the model cannot distinguish classes and relies on class imbalance.

2. **Comparison with Q2a (Full-Sample Centers + Regularization)**

   - **Q2a (λ=0.1)**:
     - Test accuracy: 80% (slightly higher than Q2b's 78%).
     - Sigmoidal curve with a steep mid-range (threshold≈0.23) indicates a stable decision boundary enabled by regularization.
   - **Q2b (σ=1.4227)**:
     - Test accuracy=78% but relies on predicting all samples as negative (TPR=3.57% vs. Q2a's 30.36%).
     - Stepwise curve reveals random centers and narrow width lead to insufficient feature coverage, forcing the model to exploit noise.

σ=1.4227
Max TeAcc: 78.0000%
Optimal Threshold: 0.4848
Training Precision: 100.00%, Recall (TPR): 13.82%, FPR: 0.00%
Test Precision: 66.67%, Recall (TPR): 3.57%, FPR: 0.52%

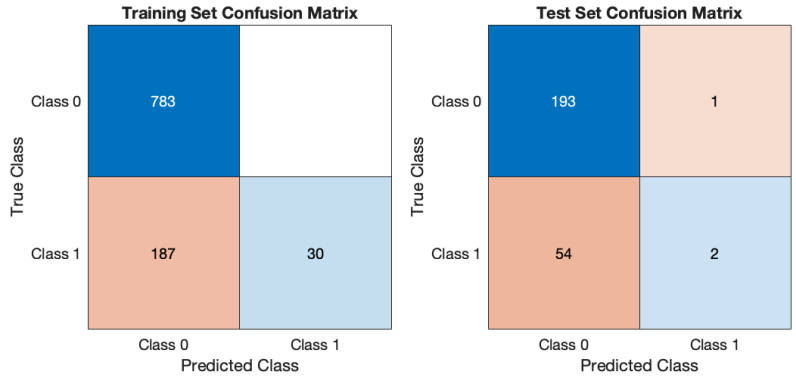


Figure 11: Q2b: Confusion Matrix ($\sigma = 1.4227$)

σ=0.1000
Max TeAcc: 78.0000%
Optimal Threshold: 0.4848
Training Precision: 100.00%, Recall (TPR): 13.82%, FPR: 0.00%
Test Precision: 66.67%, Recall (TPR): 3.57%, FPR: 0.52%

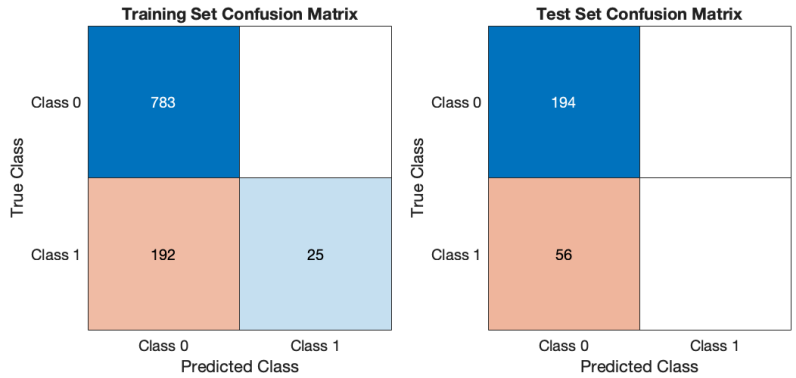


Figure 12: Q2b: Confusion Matrix ($\sigma = 0.1$)

σ=1.0000
Max TeAcc: 77.6000%
Optimal Threshold: 0.2752
Training Precision: 71.11%, Recall (TPR): 14.75%, FPR: 1.66%
Test Precision: 50.00%, Recall (TPR): 5.36%, FPR: 1.55%

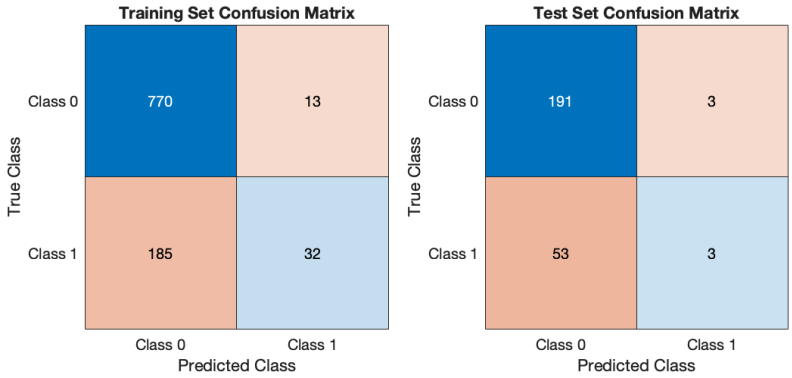


Figure 13: Q2b: Confusion Matrix ($\sigma = 1$)

σ=10.0000
Max TeAcc: 83.6000%
Optimal Threshold: 0.4858
Training Precision: 90.82%, Recall (TPR): 41.01%, FPR: 1.15%
Test Precision: 82.61%, Recall (TPR): 33.93%, FPR: 2.06%

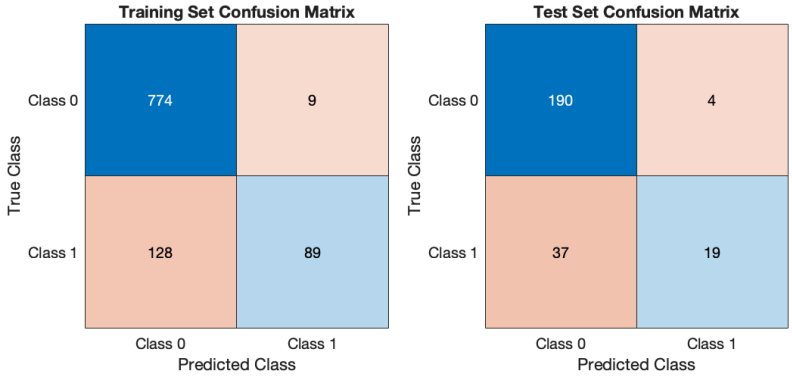


Figure 14: Q2b: Confusion Matrix ($\sigma = 10$)

σ=100.0000
Max TeAcc: 79.6000%
Optimal Threshold: 0.3585
Training Precision: 68.54%, Recall (TPR): 28.11%, FPR: 3.58%
Test Precision: 64.71%, Recall (TPR): 19.64%, FPR: 3.09%

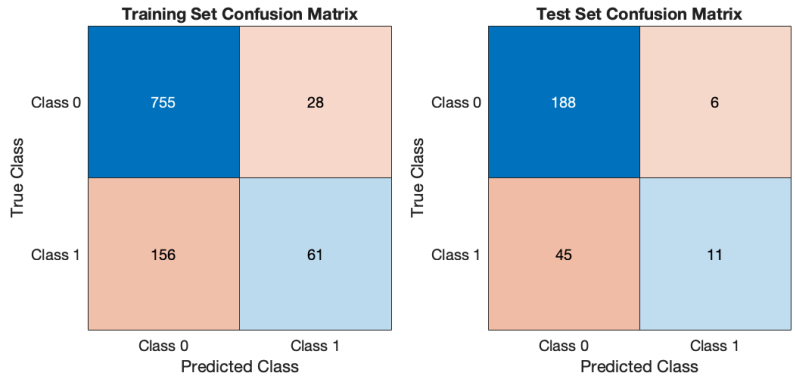


Figure 15: Q2b: Confusion Matrix ($\sigma = 100$)

σ=1000.0000
Max TeAcc: 77.6000% Optimal Threshold: 0.2170
Training Precision: 0.00%, Recall (TPR): 0.00%, FPR: 0.13%
Test Precision: NaN%, Recall (TPR): 0.00%, FPR: 0.00%

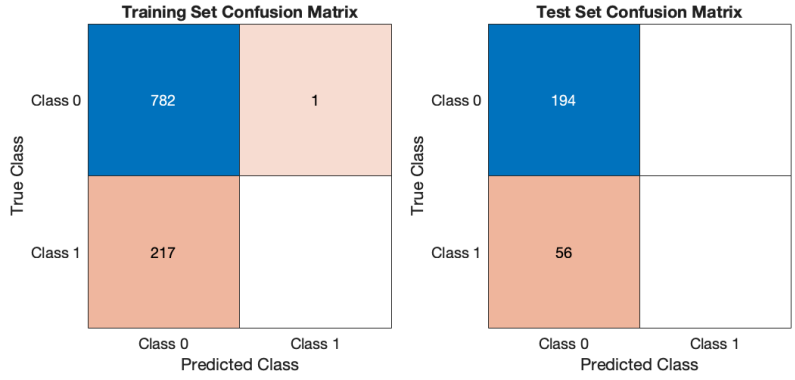


Figure 16: Q2b: Confusion Matrix ($\sigma = 1000$)

σ=10000.0000
Max TeAcc: 77.6000% Optimal Threshold: 0.2170
Training Precision: 0.00%, Recall (TPR): 0.00%, FPR: 0.13%
Test Precision: NaN%, Recall (TPR): 0.00%, FPR: 0.00%

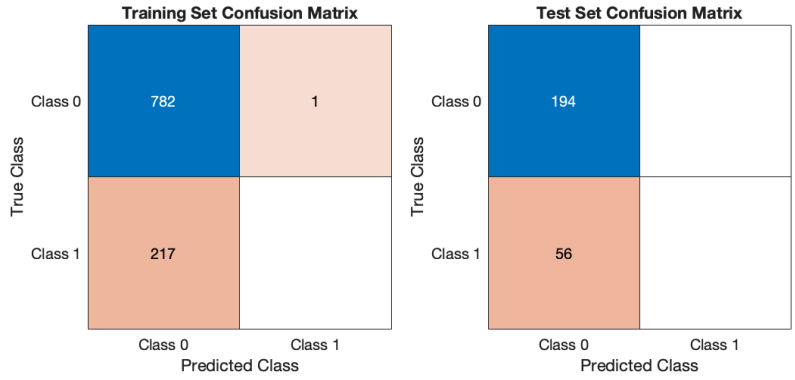


Figure 17: Q2b: Confusion Matrix ($\sigma = 10000$)

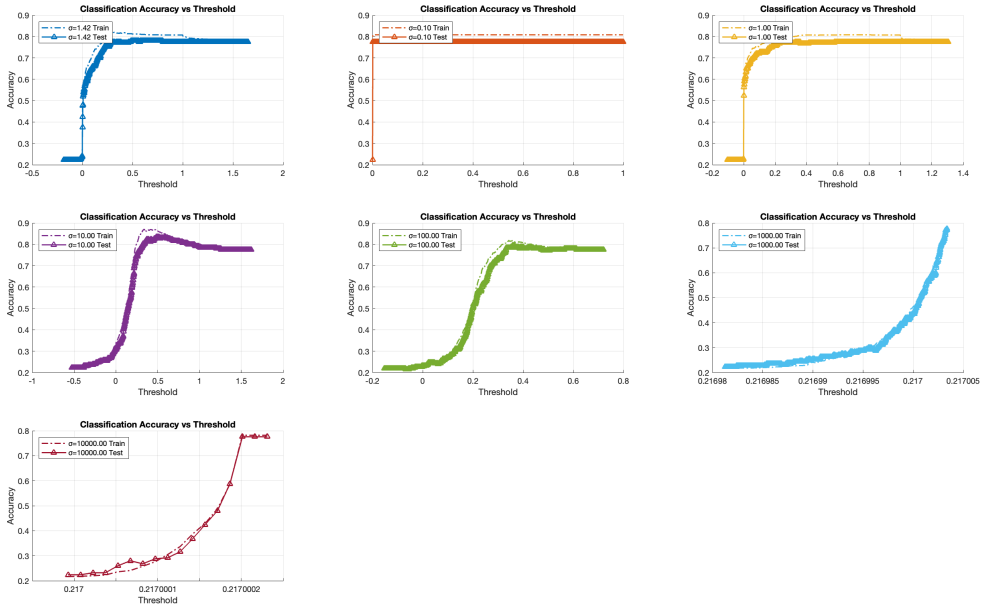


Figure 18: Q2b: Classification Accuracy vs Threshold (changing σ)

Part 2: Impact of varying σ on RBFN performance

1. σ too small (0.1 to 1): Narrow width activate only nearest neighbors, leading to overfitting training noise. The outcomes are high training precision (100% and 77.11%) but near-zero test TPR.

2. Moderate σ (10): Kernels cover local features, enabling meaningful weight learning, resulting in superior test accuracy (83.6%) compared to Q2a (80%), proving random centers with optimal σ outperform full centers with regularization.

3. σ too large ($\geq$100): Wide kernels blur local distinctions, degrading the model to a linear classifier. The outcome is that accuracy aligns with majority class prevalence, TPR $\approx$0

Conclusion:

1. σ is critical, if it is too small, the RBFN will be overfitting, indicating poor generalization, if it is too large, the model will collapse. Only if it is moderate, the model will balance locality and generalization and get optimal performance.

2. With proper σ, random centers (100) outperform full centers with regularization, reducing computational load. While this method requires σ tuning.

(c) K-Mean Clustering

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.

Solution:

In this part, K-mean clustering is used to find 2 centers of the RBFN, and the width parameter can be computed by $\sigma_i = \frac{d_{max}}{\sqrt{2M}}$. Compute the weights of RBFN without regularization.

Listing 6: MATLAB script (Q2c)

```
% K-means clustering
num_clusters = 2;
[~, centers] = kmeans(Train_Data', num_clusters, 'MaxIter', 10000, 'Replicates', 5);

%% Compute Class Means
% Get class indices
pos_idx = find(Train_Label == 1);  % Positive class (digits 7 & 9)
neg_idx = find(Train_Label == 0);  % Negative class (other digits)

% Compute mean images
mean_pos = mean(Train_Data(:, pos_idx), 2);
mean_neg = mean(Train_Data(:, neg_idx), 2);

%% Visualization
figure('Position', [100 100 1200 400])

% K-means centers
subplot(2,4,1)
imshow(reshape(centers(1,:),28,28))
title('K-means Center 1')

subplot(2,4,2)
imshow(reshape(centers(2,:),28,28))
title('K-means Center 2')

% Class means
subplot(2,4,3)
imshow(reshape(mean_pos,28,28))
title('Class 1 Mean (7&9)')

subplot(2,4,4)
imshow(reshape(mean_neg,28,28))
title('Class 0 Mean (Others)')

% Difference visualization
subplot(2,4,5)
imshow(reshape(centers(1,:)-mean_pos',28,28), [])
title('Center1 - Class1 Mean')

subplot(2,4,6)
imshow(reshape(centers(1,:)-mean_neg',28,28), [])
title('Center1 - Class0 Mean')

subplot(2,4,7)
imshow(reshape(centers(2,:)-mean_pos',28,28), [])
title('Center2 - Class1 Mean')

subplot(2,4,8)
imshow(reshape(centers(2,:)-mean_neg',28,28), [])
title('Center2 - Class0 Mean')


% find optimal threshold (based on F1-Score)
function [results, optimal_threshold, metrics] = find_optimal_threshold_adv(TrPred,TePred
    ,TrLabel,TeLabel,results)
    num_thresh= 1000;
    thresholds = linspace(min(TrPred), max(TrPred), num_thresh+1);
    thresholds= thresholds(1:num_thresh);

    TrN = length(TrLabel);
    TeN = length(TeLabel);

    metrics = struct(...
        'threshold', num2cell(thresholds),...
        'precision', zeros(num_thresh,1),...
        'recall', zeros(num_thresh,1),...
        'specificity', zeros(num_thresh,1),...
        'f1', zeros(num_thresh,1),...
        'gmean', zeros(num_thresh,1),...
        'youden', zeros(num_thresh,1));
```

```
70
71      for i = 1:num_thresh
72          t = thresholds(i);
73          % Training set metrics
74          TrPredClass = TrPred >= t;
75          TN = sum(~TrPredClass & ~TrLabel');
76          FP = sum(TrPredClass & ~TrLabel');
77          FN = sum(~TrPredClass & TrLabel');
78          TP = sum(TrPredClass & TrLabel');
79          % Test set metrics
80          TePredClass = TePred >= t;
81          teTN = sum(~TePredClass & ~TeLabel');
82          teFP = sum(TePredClass & ~TeLabel');
83          teFN = sum(~TePredClass & TeLabel');
84          teTP = sum(TePredClass & TeLabel');
85
86          % Store results(FOR COMPARISION)
87          results.Thr(i)=t;
88          results.TrAcc(i) = (TN + TP) / TrN;
89          results.TeAcc(i) = (teTN + teTP) / TeN;
90          results.ConfusionMatrices{i} = struct(...
91              'Train', [TN FP; FN TP], ...
92              'Test', [teTN teFP; teFN teTP]...
93          );
94
95          % Store metrics(FOR OPTIMAL THRESHOLD)
96          metrics(i).precision = TP/(TP + FP + eps);
97          metrics(i).recall = TP/(TP + FN + eps);
98          metrics(i).specificity = TN/(TN + FP + eps);
99          metrics(i).f1 = 2*(metrics(i).precision*metrics(i).recall)/...
100                         (metrics(i).precision + metrics(i).recall + eps);
101         metrics(i).gmean = sqrt(metrics(i).recall * metrics(i).specificity);
102         metrics(i).youden = metrics(i).recall + metrics(i).specificity - 1;
103     end
104
105     % find optimal threshold in terms of metrics
106     [~, idx_acc] = max(results.TeAcc);
107     [~, idx_f1] = max([metrics.f1]);
108     [~, idx_gmean] = max([metrics.gmean]);
109     [~, idx_youden] = max([metrics.youden]);
110
111     opt_idx = idx_f1;
112
113     % Additional metrics display
114     fprintf('Optimal Threshold: %.4f,Test accuracy: %.4f%%, F1-Score: %.4f, G-Mean: %.4f,
            Youden''s J: %.4f\n',...
115         thresholds(opt_idx), 100*results.TeAcc(opt_idx), metrics(opt_idx).f1, metrics(
                opt_idx).gmean, metrics(opt_idx).youden);
116
117     optimal_threshold = struct(...
118         'f1', thresholds(idx_f1),...
119         'gmean', thresholds(idx_gmean),...
120         'youden', thresholds(idx_youden));
121
122     % Get optimal confusion matrices
123     optimal_cm = results.ConfusionMatrices{opt_idx};
124
125     fprintf('Training Precision: %.2f%%, Recall (TPR): %.2f%%, FPR: %.2f%%\n', ...
126     100*metrics(opt_idx).precision, ...
127     100*metrics(opt_idx).recall, ...
128     100*(1-metrics(opt_idx).specificity));
129
130     fprintf('Test Precision: %.2f%%, Recall (TPR): %.2f%%, FPR: %.2f%%\n', ...
131     100*optimal_cm.Test(2,2)/(optimal_cm.Test(2,2)+optimal_cm.Test(1,2)), ...
132     100*optimal_cm.Test(2,2)/(optimal_cm.Test(2,2)+optimal_cm.Test(2,1)), ...
133     100*optimal_cm.Test(1,2)/(optimal_cm.Test(1,2)+optimal_cm.Test(1,1)));
134
135     % Plot confusion matrices
136     class_names = {'Class 0', 'Class 1'};
137     plot_confusion_matrix(optimal_cm, class_names);
138 end
139
```

```
%% Train RBF Network and Evaluation
% Calculate sigma (based on center distance)
d_max = norm(centers(1,:)-centers(2,:));
sigma = d_max/sqrt(2*num_clusters);

% Construct RBF feature matrix
Phi_train = exp(-pdist2(Train_Data', centers).^2/(2*sigma^2));
Phi_test = exp(-pdist2(Test_Data', centers).^2/(2*sigma^2));

% Solve weights (without regularization)
w = Phi_train \ Train_Label';

% Prediction and evaluation
TrPred = Phi_train * w;
TePred = Phi_test * w;

[results,optimal_threshold, metrics] = find_optimal_threshold_adv(TrPred, TePred,
    Train_Label, Test_Label, results);

% Visualization
figure('Name', 'Classification Accuracy vs Threshold (k-means)');
hold on;
plot(results.Thr,results.TrAcc,'-.', 'LineWidth', 1.5);
plot(results.Thr,results.TeAcc,'^-', 'LineWidth', 1.5);
legend('Train', 'Test');
title('Classification Accuracy vs Threshold (k-means)');
xlabel('Threshold');
ylabel('Accuracy');
legend('Location','northwest');
grid on;
axis auto;

%% Quantitative Analysis

% Helper Function
function [corr_coef, euclidean_dist] = calc_similarity(vec1, vec2)
    % Calculate Pearson correlation and Euclidean distance
    corr_coef = corr(vec1(:), vec2(:));
    euclidean_dist = norm(vec1 - vec2);
end

% Calculate similarity metrics
similarity_metrics = struct();

% Similarity between Center1 and positive class mean
[similarity_metrics.center1_pos_corr, similarity_metrics.center1_pos_dist] = ...
    calc_similarity(centers(1,:), mean_pos);

% Similarity between Center1 and negative class mean
[similarity_metrics.center1_neg_corr, similarity_metrics.center1_neg_dist] = ...
    calc_similarity(centers(1,:), mean_neg);

% Similarity between Center2 and positive class mean
[similarity_metrics.center2_pos_corr, similarity_metrics.center2_pos_dist] = ...
    calc_similarity(centers(2,:), mean_pos);

% Similarity between Center2 and negative class mean
[similarity_metrics.center2_neg_corr, similarity_metrics.center2_neg_dist] = ...
    calc_similarity(centers(2,:), mean_neg);

% Display results
disp(struct2table(similarity_metrics))
```

Results and analysis:

The RBFN model with k-means centers demonstrates poor performance in distinguishing the two selected digit classes across all evaluation metrics, shown in Figure 20 to 22 and MATLAB command line output (also see Table 4).

Key observations and interpretations are as follows:

1. Threshold Dynamics and Metric Trade-offs

    - F1-Score Optimization:
      The model predicts almost all samples as positive (TPR=96.43%, FPR=97.42%), achieving high recall but catastrophically low precision (22.22%). This indicates the RBFN fails to capture discriminative features, relying on trivial correlations (e.g., slight grayscale shifts).
    - Accuracy Optimization
      The model collapses to predicting all samples as negative (TPR=0%, FPR=0%), exploiting class imbalance (83.2% negative samples in test set). While test accuracy reaches 77.6%, this metric is misleading due to imbalance.
    - G-Mean Optimization
      Balanced TPR=44.64% and FPR=44.85% yield G-Mean=0.4869, indistinguishable from random guessing (G-Mean=0.5). The linear decision boundary in feature space provides no meaningful separation.

2. Failure of k-means Feature Extraction
   The K-Means centers and means of positive/negative classes are shown in Figure 23.

    - Cluster Centers vs. Class Means:
      As shown in Table 5 and 6, the k-means centers show high Pearson correlation with both class means (0.87–0.97 for positive/negative classes), but large Euclidean distances (94.98–108.31), implying they encode global grayscale intensity rather than digit structure. Visualizations confirm centers resemble blurred averages of multiple digits, losing discriminative edges.
    - Spatial Information Loss:
      Flattening 28×28 images to 784-D vectors discards spatial relationships. k-means with Euclidean distance prioritizes pixel-wise intensity similarity, which is irrelevant for digit identity (e.g., digits 7 and 9 share similar stroke orientations but differ topologically).

3. Model Limitations

    - Class Imbalance Amplification
      With 217 positive vs. 783 negative training samples, the model biases toward the majority class. Regularization (not applied here) might mitigate this but cannot fix invalid features.

4. Comparison to Alternative Methods
   The K-Means-based RBFN underperforms both:

    - Exact Interpolation Method: Uses all training samples as centers, capturing finer details but overfitting noise.

    - Random Fixed Centers (100 centers): Stochasticity may accidentally select informative pixels, offering better generalization than systematic grayscale clustering. More centers provide enough complexity of the model to catch enough characteristics for correct classification.

The k-means-RBFN pipeline is fundamentally mismatched to the task. Effective digit classification requires feature extractors sensitive to local structures (e.g., edges, strokes), such as

convolutional kernels or nonlinear SVMs. Future work should explore preprocessing (e.g., edge detection) or alternative clustering methods (e.g., spectral clustering on patch-based features).

Metric	Optimal Threshold	Precision (Te)	Recall (Te)	FPR (Te)	Behavior (Te)
F1-Score	0.0073 (Left)	22.22%	96.43%	97.42%	Predicts all as positive
Accuracy	0.3072 (Right)	NaN	0%	0%	Predicts all as negative
G-Mean	0.1577 (Mid)	22.32%	44.64%	44.85%	Random guessing

Table 4: Q2c: Threshold Dynamics

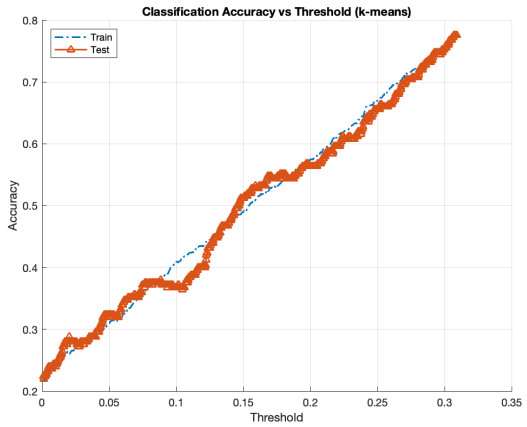


Figure 19: Q2c: Accuracy vs Threshold (K-means)

Optimal Threshold: 0.0073, Test accuracy: 23.6000%, F1-Score: 0.3592, G-Mean: 0.1509, Youden's J: 0.0138
Training Precision: 21.94%, Recall (TPR): 99.08%, FPR: 97.70%
Test Precision: 22.22%, Recall (TPR): 96.43%, FPR: 97.42%

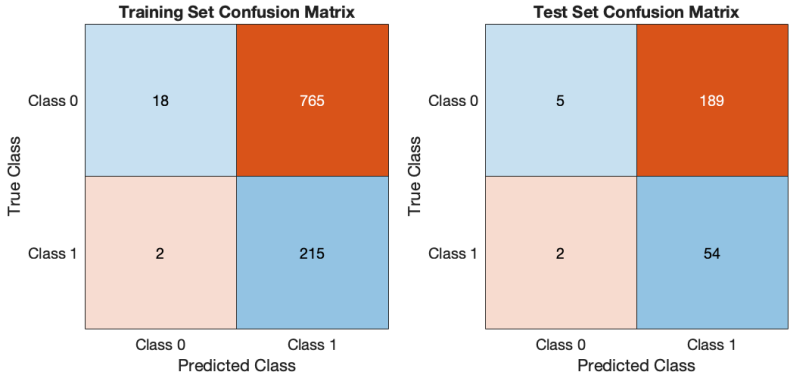


Figure 20: Q2c: Confusion Matrix (τ based on F1-score)

Optimal Threshold: 0.3072,Test accuracy: 77.6000%, F1-Score: 0.0088, G-Mean: 0.0674, Youden's J: -0.0082
Training Precision: 9.09%, Recall (TPR): 0.46%, FPR: 1.28%
Test Precision: NaN%, Recall (TPR): 0.00%, FPR: 0.00%

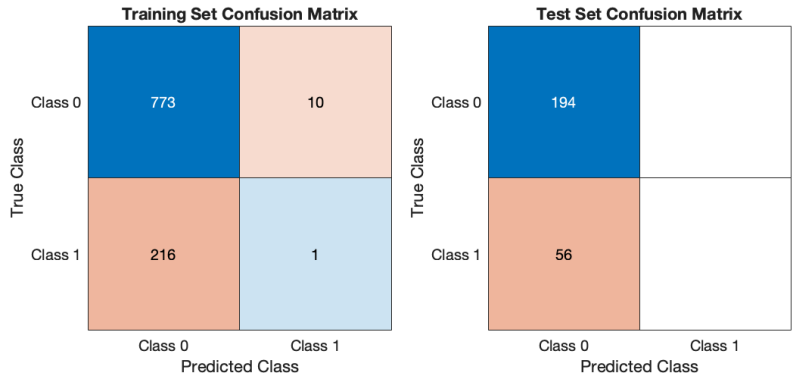


Figure 21: Q2c: Confusion Matrix (τ based on accuracy)

Optimal Threshold: 0.1577,Test accuracy: 52.8000%, F1-Score: 0.2853, G-Mean: 0.4869, Youden's J: -0.0235
Training Precision: 20.85%, Recall (TPR): 45.16%, FPR: 47.51%
Test Precision: 22.32%, Recall (TPR): 44.64%, FPR: 44.85%

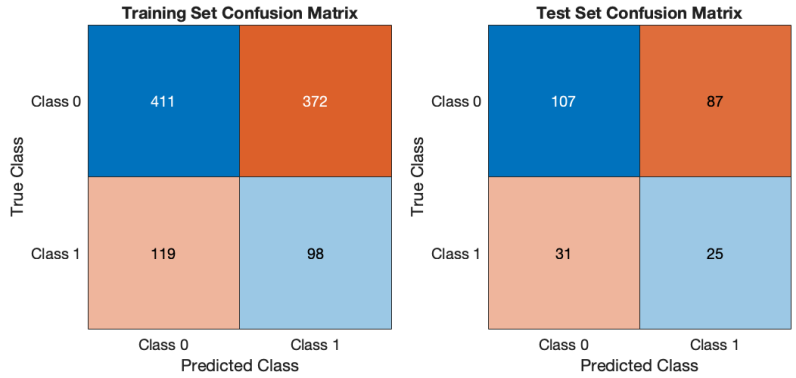


Figure 22: Q2c: Confusion Matrix (τ based on G-Mean)

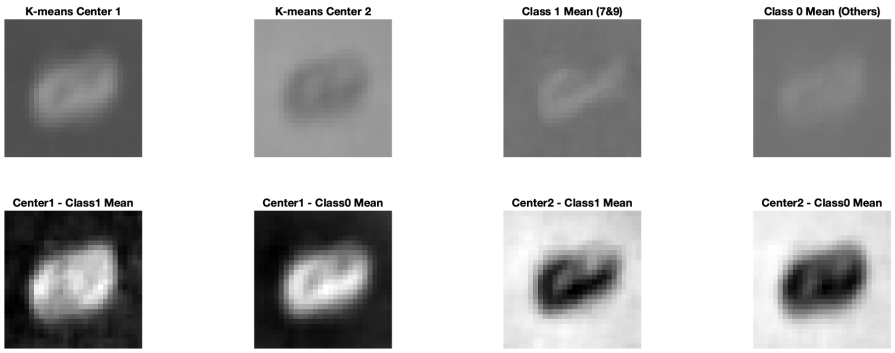


Figure 23: Q2c: Visualization of Centers and Means of Classes

	center 1	center 2
positive mean	0.87014	-0.83482
negative mean	0.96916	-0.91161

Table 5: Q2c: Pearson correlation

	center 1	center 2
positive mean	94.982	108.31
negative mean	98.675	102.42

Table 6: Q2c: Euclidean distance

Q3 Self-Organizing Map (SOM)

(a) 1-D SOM

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). The training points sampled from the "hat" can be obtained by the following code:

```
x = linspace(-pi,pi,400);
trainX = [x; sinc(x)]; % 2x400 matrix
plot(trainX(1,:),trainX(2,:),'+r'); axis equal
```

Solution:
The principle goal of the self-organizing map is to transform an incoming signal pattern of arbitrary dimension into a one- or two-dimensional discrete feature map, and to perform this transformation adaptively in a topologically ordered fashion. Steps are as follows:

1. Data generation
   The training data consists of 400 points sampled from the **sinc function** over the interval [-π, π]:
   $$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

2. SOM parameters
   `num_neurons`: A 1D chain of 40 neurons provides sufficient resolution to approximate the sinc curve.
   `sigma0`: Initialized to half the grid length ($\sigma_0 = 20$), ensuring the neighborhood covers the entire lattice initially.
   `epochs`: Increased to 10,000 for slow convergence (critical for topology preservation).

3. Weight Initialization
   Random initialization avoids bias, allowing the SOM to self-organize. Range [-1, 1] matches the normalized input data (sinc values lie in [-0.217, 1]).

4. Training Process
   The SOM training involves three phases: Competition, Cooperation, and Adaptation.
   Phase 1: Competition (Finding the Winning Neuron)
   For each input sample $\mathbf{x}_k$, the winning neuron i^* is the one with the smallest Euclidean distance:
   $$i^* = \arg\min_i \|\mathbf{w}_i - \mathbf{x}_k\|^2$$

Phase 2: Cooperation (Neighborhood Function)
The neighborhood function $h_{i,i^*}(t)$ determines how much neighboring neurons are updated. A Gaussian kernel is used:

$$h_{i,i^*}(t) = \exp\left(-\frac{\|r_i - r_{i^*}\|^2}{2\sigma(t)^2}\right)$$

where r_i is the position of neuron i in the lattice.
σ Decay: Exponential decay ensures gradual reduction of neighborhood influence:

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau}\right), \quad \tau = \frac{\text{epochs}}{\ln(\sigma_0)}$$

Phase 3: Adaptation (Weight Update)
Weights are updated to move toward the input sample, scaled by the learning rate $\eta(t)$ and neighborhood $h_{i,i^*}(t)$:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta(t) \cdot h_{i,i^*}(t) \cdot (\mathbf{x}_k - \mathbf{w}_i(t))$$

η Decay: Exponential decay is one possible choice:

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{\tau_2}\right)$$

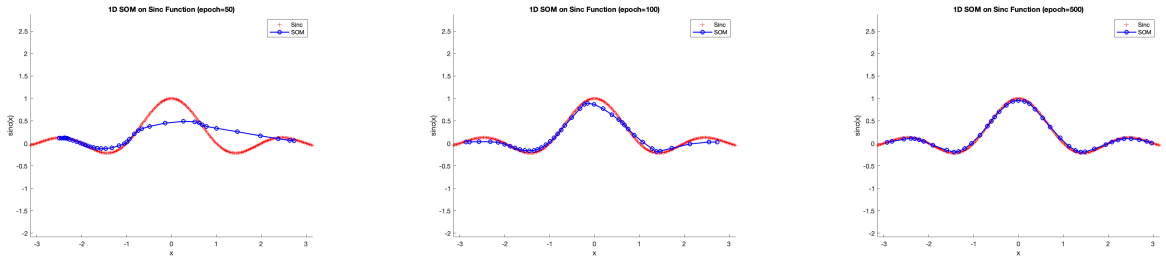
where τ_2 is another time-constant to control the decay rate, here I choose $\tau_2 = \texttt{epochs}$.

Listing 7: MATLAB script (Q3a)

```
%% 1D SOM for Sinc Function
% Generate training data
x = linspace(-pi, pi, 400);
trainX = [x; sinc(x)]; % 2x400 matrix

% SOM parameters
num_neurons = 40;       % Number of neurons in the 1D lattice
epochs = 1000;          % Training iteration
eta0 = 1.0;             % Initial learning rate
sigma0 = num_neurons/2; % Initial neighborhood width (half the grid length)

% weight initialization
rng(2025); % seed for reproducibility
weights = rand(num_neurons, 2)*2 - 1; % Random weights in [-1, 1]

% training process
for epoch = 1:epochs
    % Select samples randomly
    sample_idx = randi(size(trainX,2));
    sample = trainX(:, sample_idx);

    % Competition (Find a winner)
    distances = sum((weights - sample').^2, 2); % Squared Euclidean distance
    [~, winner] = min(distances);

    % Exponential decay
    lr = eta0 * exp(- epoch/epochs); % learning rate decay
    sigma = sigma0 * exp(-epoch/(epochs/log(sigma0))); % neighborhood width decay

    % Adaptation (Weight Update)
    neuron_indices = 1:num_neurons;
    distances_to_winner = abs(neuron_indices - winner);
    neighborhood = exp(-distances_to_winner.^2 / (2*sigma^2));
    delta = lr * neighborhood' .* (sample' - weights);
    weights = weights + delta;
```

```
end

% Visualization
figure;
scatter(trainX(1,:), trainX(2,:), 10, '+r'); hold on;
plot(weights(:,1), weights(:,2), 'b-o', 'LineWidth', 1.5);
legend('Sinc','SOM');
title('1D SOM on Sinc Function');
xlabel('x'); ylabel('sinc(x)');
axis equal;
```

Results and analysis:
As shown in Figure 24, the SOM neurons (blue circles) align along the red sinc curve, preserving the 1D topology. Exponential decay allows aggressive updates early (for coarse topology formation) and fine adjustments later, while requiring careful tuning of η_0 and σ_0 to avoid overshooting or slow convergence.



(a) epoch=50 (b) epoch=100 (c) epoch=500

Figure 24: Q3a: SOM for Sinc function

(b) 2-D SOM

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). The training points sampled from the "circle" can be obtained by the following code:

```
X = randn(800,2);
s2 = sum(X.^2,2);
trainX = (X.*repmat(1*(gammainc(s2/2,1).^(1/2))./sqrt(s2),1,2))'; % 2x800 matrix
plot(trainX(1,:),trainX(2,:),'+r'); axis equal
```

Solution:

1. SOM Parameters
   This time the grid structure is a 2D grid of 8×8 neurons (vs. 1D chain in Q3a), so initialize the neighborhood width as:

$$\sigma_0 = \frac{\sqrt{\text{grid\_size}(1)^2+\text{grid\_size}(2)^2}}{2} = \frac{\sqrt{8^2+8^2}}{2} \approx 5.66$$

2. Distance Metric
   Euclidean distance in 2D grid space (vs. 1D in Q3a):

$$\text{distance} = \sqrt{(i-i_{\text{win}})^2+(j-j_{\text{win}})^2}$$

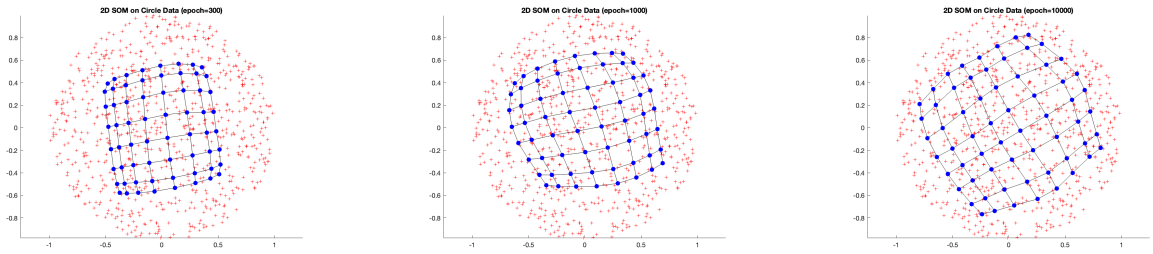
3. Weight Update Rule
   Loop Structure: Nested loops over 2D grid indices (i and j).
   Vectorization: Unlike Q3a's 1D vectorization, 2D updates require explicit loops for clarity.

Listing 8: MATLAB script (Q3b)

```
% Generate training data
rng(2025);
X = randn(800,2);
s2 = sum(X.^2, 2);
trainX = (X.*repmat(1*(gammainc(s2/2,1).^(1/2))./sqrt(s2),1,2))';

% SOM parameters
grid_size = [8 8];            % 2D grid size (8x8 neurons)
epochs = 10000;               % Training iterations
eta0 = 0.1;                   % Initial learning rate
sigma0 = sqrt(grid_size(1)^2+grid_size(2)^2)/2;  % Initial neighborhood width (half of
    grid diagonal)

% Initialize weights
weights = rand(grid_size(1), grid_size(2), 2)*2 - 1;  % Random weights in [-1, 1]

% Training process
for epoch = 1:epochs
    % Randomly select a sample
    sample_idx = randi(size(trainX,2));
    sample = trainX(:, sample_idx);

    % Find winning neuron (minimum Euclidean distance)
    distances = vecnorm(reshape(weights, [], 2) - sample', 2, 2);
    [~, winner_idx] = min(distances);
    [i_win, j_win] = ind2sub(grid_size, winner_idx);

    % Calculate learning rate and neighborhood width
    lr = eta0 * exp(-epoch/epochs);                % Learning rate decay
    sigma = sigma0 * exp(-epoch/(epochs/log(sigma0)));  % Neighborhood width decay

    % Update weights
    for i = 1:grid_size(1)
        for j = 1:grid_size(2)
            distance = sqrt((i - i_win)^2 + (j - j_win)^2);  % Grid distance to winner
            neighborhood = exp(-distance^2/(2*sigma^2));     % Gaussian neighborhood
            weights(i,j,:) = squeeze(weights(i,j,:)) + lr*neighborhood*(sample - squeeze(
                weights(i,j,:)));
        end
    end
end

% Visualization
figure;
scatter(trainX(1,:), trainX(2,:), 10, 'r+'); hold on;  % Plot training data

% Draw neuron connections
for i = 1:grid_size(1)
    for j = 1:grid_size(2)
        % Horizontal connections
        if j < grid_size(2)
            plot([weights(i,j,1), weights(i,j+1,1)],...
                 [weights(i,j,2), weights(i,j+1,2)], 'k-');
        end
        % Vertical connections
        if i < grid_size(1)
            plot([weights(i,j,1), weights(i+1,j,1)],...
                 [weights(i,j,2), weights(i+1,j,2)], 'k-');
        end
    end
end

```

```
scatter(reshape(weights(:,:,1),[],1), reshape(weights(:,:,2),[],1), 50, 'b', 'filled');
    % Plot neuron weights
title(sprintf('2D SOM on Circle Data (epoch=%d)',epochs));
axis equal;
```

Results and analysis:
The neuron weights (blue points) form a deformed grid approximating the circular shape of the input data. Adjacent neurons in the 2D grid remain adjacent in the input space, preserving the manifold structure of the circle.
Horizontal and vertical connections (lines) curve to follow the circular pattern. The originally rectangular grid (Figure 25a) deform into a roughly circular shape (Figure 25c). Connections between neighboring neurons show smooth transitions, avoiding sharp angles.



(a) epoch=300 (b) epoch=1000 (c) epoch=10000

Figure 25: Q3a: SOM for Sinc function

(c) High Dimension

Write your own code to implement a SOM that clusters and classifies handwritten digits. 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).

(c-1) Semantic map and weight visualization

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 $\times$10 map (a simple way could be to reshape the weights of a neuron into a 28 $\times$28 matrix, i.e. dimension of the inputs, and display it as an images). Make comments on them, if any.
Solution:
Key steps are as follows:
Part 1: Data Preprocessing and SOM Training

1. Data Filtering
   My matric number is A0295779Y, so I omit class 0 and 4 from the training set, retaining only classes 1, 2, and 3.

2. SOM Parameter Initialization
   Define a 10x10 grid with input dimension 784 (28x28 images). Initialize the weight matrix `weights` with random values (range [0,1]).

3. Training Process
   Random Sampling: Select a random training sample in each iteration.

BMU Search: Compute Euclidean distances between the sample and all neurons, selecting the neuron with the smallest distance as the Best Matching Unit (BMU).
Weight Update: Update weights of the BMU and its neighbors using a Gaussian neighborhood function. Learning rate and neighborhood radius decay exponentially over time.

Part 2: Label Mapping

1. BMU Assignment
   Compute distance matrix `distances_all` between all training samples and neurons (size: sample number $\times 100$). For each sample, find its BMU index `assignments`.

2. Majority Voting
   For each neuron, collect class labels of all samples mapped to it. Assign the majority class as the neuron's label.

3. Unactivated Neurons
   I found that some neurons may not be assigned valid label, here are some possible reasons:

   - **Under-Training**: Insufficient iterations for convergence.
   - **Initialization Bias**: Random weights may not align with data clusters.
   - **Decay Strategy**: Exponential decay limits weight updates in later epochs.

Part 3: Visualization

1. Weight Heatmap: Combine each neuron's weight vector (28) into a single image, and then normalize each neuron's weights to [0,1] individually to preserve local contrast (Figure 27).

2. Semantic Map: Code color as Red (class 1), Green (class 2), Blue (class 3), Gray (unactivated neurons), shown in Figure 26. And then draw horizontal/vertical lines between adjacent neurons to form the grid structure.

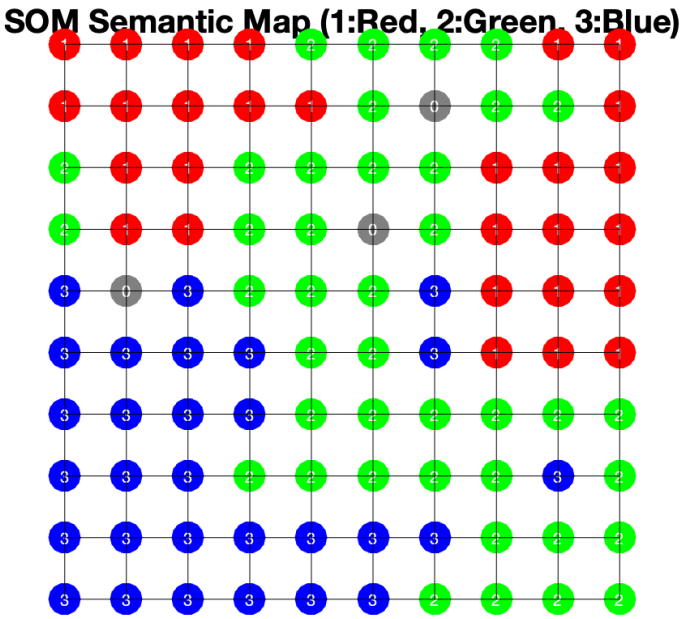


Figure 26: Q3c-1: Semantic Map

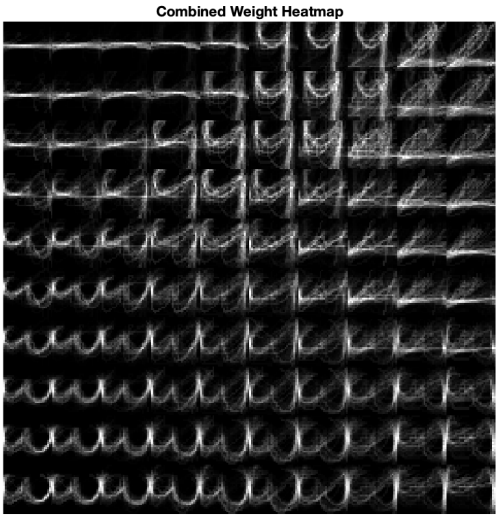


Figure 27: Q3c-1: Weight Visualization

Listing 9: MATLAB script (Q3c-1)

```
%% Data Loading & Preprocessing
% Load dataset
load('./Digits.mat');

% Determine omitted classes based on student ID (last digit=9)
last_digit = 9;
omitted_class1 = mod(last_digit, 5);       % Class 4 (since 9%5=4)
omitted_class2 = mod(last_digit+1, 5);     % Class 0 (since (9+1)%5=0)

```

```
% Filter training data (exclude classes 0 and 4)
train_mask = ~ismember(train_classlabel, [omitted_class1, omitted_class2]);
Train_Data = train_data(:, train_mask);    % [784xN] matrix
Train_Label = train_classlabel(train_mask);% [1xN] vector

%% SOM Training
% Parameters
grid_size = [10 10];       % SOM grid dimensions
input_dim = 784;           % Input dimension (28x28 images)
epochs = 10000;            % Training iterations
eta0 = 0.1;                % Initial learning rate
sigma0 = 5.0;              % Initial neighborhood radius
rng(2025);
weights = rand(grid_size(1), grid_size(2), input_dim); % Random initialization

%% Training Loop
for epoch = 1:epochs
    % Randomly select a sample
    sample_idx = randi(size(Train_Data,2));
    sample = Train_Data(:, sample_idx);

    % Find Best Matching Unit (BMU)
    distances = vecnorm(reshape(weights, [], input_dim) - sample', 2, 2);
    [~, winner_idx] = min(distances);
    [i_win, j_win] = ind2sub(grid_size, winner_idx);

    % Update learning rate and neighborhood radius
    lr = eta0 * exp(-epoch/epochs);               % Exponential decay
    sigma = sigma0 * exp(-epoch/(epochs/log(sigma0)));

    % Update weights for all neurons
    for i = 1:grid_size(1)
        for j = 1:grid_size(2)
            distance = sqrt((i - i_win)^2 + (j - j_win)^2);
            neighborhood = exp(-distance^2/(2*sigma^2));
            weights(i,j,:) = squeeze(weights(i,j,:)) + lr*neighborhood*(sample - squeeze(
                weights(i,j,:)));
        end
    end
end

%% Label Mapping
% Assign BMUs for all training samples
weights_flat = reshape(weights, [], input_dim);        % [100x784]
distances_all = pdist2(Train_Data', weights_flat);     % [N_trainx100]
[~, assignments] = min(distances_all, [], 2);          % BMU indices

% Generate label map using majority voting
label_map = zeros(grid_size);
for neuron_idx = 1:prod(grid_size)
    mask = (assignments == neuron_idx);
    if sum(mask) > 0
        [i,j] = ind2sub(grid_size, neuron_idx);
        label_map(i,j) = mode(Train_Label(mask));      % Majority class
    end
end

%% Weight Heatmap
% Combine all neuron weights into a single image
all_imgs = zeros(28*grid_size(1), 28*grid_size(2));
for i = 1:grid_size(1)
    for j = 1:grid_size(2)
        img = reshape(weights(i,j,:), 28, 28);
        img = (img - min(img(:))) / (max(img(:)) - min(img(:)) + eps); % Normalize
        all_imgs((i-1)*28+1:i*28, (j-1)*28+1:j*28) = img;
    end
end
figure; imshow(all_imgs); title('Combined Weight Heatmap');

fprintf('Unique activated neurons: %d/100\n', numel(unique(assignments)));

%% Semantic Map
```

```
figure; axis equal; hold on;
color_map = [1 0 0; 0 1 0; 0 0 1; 0.5 0.5 0.5]; % RGB for classes 1/2/3/gray

% Plot neurons with class labels
for i = 1:grid_size(1)
    for j = 1:grid_size(2)
        x = j; y = grid_size(1) - i + 1;
        class_id = label_map(i,j);
        if ~ismember(class_id, [1,2,3]), class_id = 4; end % Unlabeled neurons
        scatter(x, y, 400, 'filled', 'MarkerFaceColor', color_map(class_id,:));
        text(x, y, num2str(label_map(i,j)), 'HorizontalAlignment', 'center', 'Color', 'w'
            );
    end
end

% Draw grid connections
for i = 1:grid_size(1)
    for j = 1:grid_size(2)
        x1 = j; y1 = grid_size(1) - i + 1;
        if j < grid_size(2), line([x1, x1+1], [y1, y1], 'Color', 'k'); end % Horizontal
        if i < grid_size(1), line([x1, x1], [y1, y1-1], 'Color', 'k'); end % Vertical
    end
end
title('SOM Semantic Map (1:Red, 2:Green, 3:Blue)','FontSize',20); axis off;
```

Results and analysis:

Comments on weight heatmap and semantic map:

The combined weight heatmap provides critical insights into the feature learning capability of the SOM:

1. Feature Representation:
   Well-defined weight vectors (e.g., visible digit strokes or edges) indicate successful learning of local patterns. Neurons corresponding to distinct classes (1, 2, 3) should exhibit unique structural patterns (e.g., vertical strokes for ”1”, curved shapes for ”2”). Blurred or noisy patches suggest under-training or improper initialization. For example, neurons near grid edges may show less distinct patterns due to sparse activation.

2. Topological Order:
   Smooth transitions between adjacent neurons (e.g., gradual shape variations from left to right) confirm the SOM’s ability to preserve input space topology. Abrupt changes may indicate training instability or insufficient iterations.

The semantic map reflects the SOM’s class distribution and spatial organization:

1. Class Clustering:
   Compact Clusters: Tight groupings of same-colored nodes (e.g., a red cluster for class 1) demonstrate effective class separation.
   Boundary Ambiguity: Mixed-color regions (e.g., green/blue overlaps) highlight challenging areas where classes 2 and 3 share visual similarities.

2. Unactivated Neurons:
   Gray nodes typically appear in regions far from training data distributions. For a 10x10 grid, $\leq$10% unactivated neurons are acceptable.
   Higher rates suggest: (a)Training Issues: Increase epochs or adjust learning rates. (b)Initialization Bias: Use PCA-based initialization.

3. Grid Connectivity: Continuous lines between nodes validate the grid structure. Disconnected lines may arise from rendering artifacts or code errors in grid coordinate calculations.

(c-2) SOM classifier and accuracy evaluation

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.

Solution:

1. Prediction: For each test sample, compute distances to all neurons.Sort distances and iteratively check BMUs until a labeled neuron is found.
2. Fallback Mechanism: If all candidate BMUs are unlabeled, predict the majority class of the training set.
3. Confusion Matrix and Accuracy: First I remove classes 0 and 4, retain 1, 2, 3. Follow above steps to get the prediction and compute the confusion matrix (Figure 28). The diagonal values indicate correct predictions.

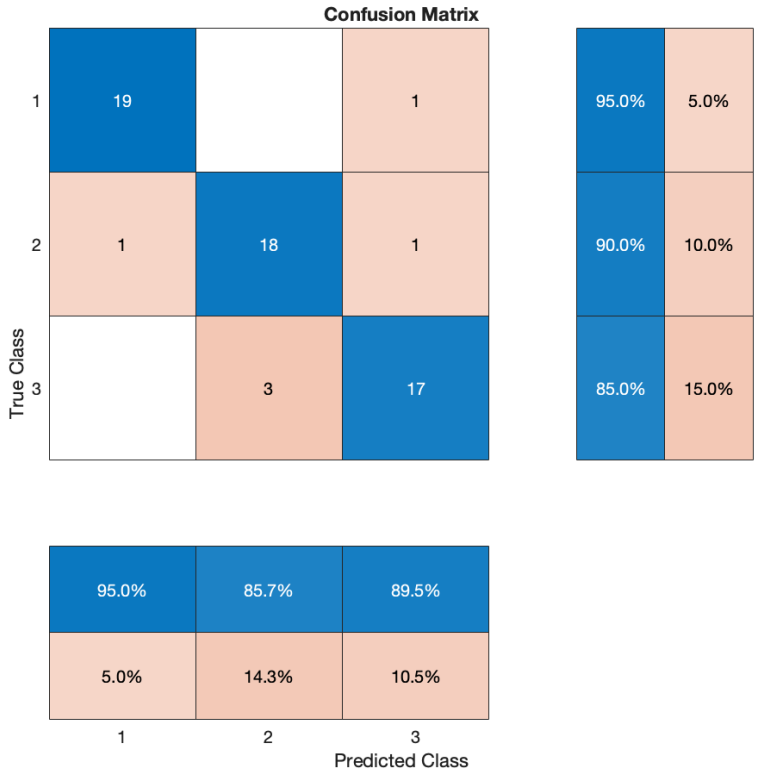


Figure 28: Q3c-2: Confusion Matrix

Listing 10: MATLAB script (Q3c-2)

```
%% Testing & Evaluation
% Prediction with Fallback

% Preprocess test data (exclude classes 0/4)
valid_test_mask = ismember(test_classlabel, [1,2,3]);
Test_Data = test_data(:, valid_test_mask);
Test_Labels = test_classlabel(valid_test_mask);

% Predict labels with BMU skipping unlabeled neurons
test_pred = zeros(1, size(Test_Data,2));
for k = 1:size(Test_Data,2)
    sample = Test_Data(:, k);
    distances = vecnorm(weights_flat - sample', 2, 2);
    [~, sorted_idx] = sort(distances);

    % Find the first valid BMU
    attempt = 1;
    while attempt <= length(sorted_idx)
        winner_idx = sorted_idx(attempt);
        [i_win, j_win] = ind2sub(grid_size, winner_idx);
        if label_map(i_win, j_win) ~= 0, break; end
        attempt = attempt + 1;
    end

    % Fallback to majority class if all BMUs are invalid
    if label_map(i_win, j_win) == 0
        test_pred(k) = mode(Train_Label);
    else
        test_pred(k) = label_map(i_win, j_win);
    end
end

% Confusion Matrix & Accuracy
C = confusionmat(Test_Labels, uint8(test_pred), 'Order', [1,2,3]);
figure; confusionchart(C, {'1','2','3'}, ...
    'Title', 'Confusion Matrix', ...
    'RowSummary', 'row-normalized', ...
    'ColumnSummary', 'column-normalized');

accuracy = sum(diag(C)) / sum(C(:));
fprintf('Test Accuracy: %.2f%%\n', accuracy*100);
```

Results and analysis:

Test Accuracy: 90.00%.

The SOM achieves excellent performance (90% accuracy) on this filtered digit classification task, demonstrating its capability to learn discriminative features and organize them topologically. Key strengths include efficient feature learning and clear semantic mapping. Further improvements could focus on advanced classification layers and hyperparameter optimization. This model is highly suitable for applications requiring interpretable feature maps and moderate computational resources.