Q 1. ***Rosenbrock's Valley Problem***
Consider the Rosenbrock's Velley function:

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

(a) *Show that it has a global minimum at (x, y) = (1,1) where f(x, y) = 0.* ***(5 Marks)***

Solution:
1. The Rosenbrock's Velley function is nonnegative, i.e., $f(x,y) \geq 0, \forall(x,y) \in \mathbb{R}^2$;
2. The $f(x,y) = 0$ holds **if and only if** both terms are zero:

$$\begin{cases} 1-x=0 \\ y-x^2=0 \end{cases} \implies \begin{cases} x=1 \\ y=1^2=1 \end{cases}.$$

3. Find critical point by calculating first-order partial derivatives:

$$\nabla f = \begin{bmatrix} -2(1-x) - 400x(y-x^2) \\ 200(y-x^2) \end{bmatrix}$$

Set $\nabla f = 0$, we can obtain $x = y = 1$, the only critical point is (1, 1).
4. Check whether Hessian Matrix H(1,1) is positive definite:
The second-order partial derivatives:

$$\frac{\partial^2 f}{\partial x^2} = 2 + 400(3x^2 - y), \quad \frac{\partial^2 f}{\partial y^2} = 200, \quad \frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x} = -400x.$$

So Hessian Matrix at (1, 1):

$$H(1,1) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}_{(1,1)} = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}.$$

$Tr(H) = 802 + 200 = 1002 > 0, det(H) = 802 * 200 - (-400)^2 = 400 > 0$, solve $\lambda^2 - Tr(H)\lambda + det(H) = 0$: $\lambda_1 = 1001.6, \lambda_2 = 0.4$ (both positive). Since all eigenvalues are positive, H(1,1) is **positive definite**, confirming (1, 1) as the local minimum.
Thus, $(1, 1)$ is the only point where $f(x,y) = 0$, confirming it as the global minimum.□

Now suppose that the starting point is randomly initialized in the open interval (-1, 1) for x and y, find the global minimum using:

(b) ***Steepest (Gradient) descent method****:*

$$w(k+1) = w(k) - \eta g(k)$$

with learning rate η = 0.001. Here the weight vector refers to the two-dimensional vector [x, y]. Record the number of iterations when f(x, y) converges to (or very close to) 0 and plot out the trajectory of (x, y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum. What would happen if a larger learning rate, say η = 1.0, is used? ***(5 Marks)***

Solution:
The Gradient Decrease method was implemented to minimize the Rosenbrock function:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla f(\mathbf{w}^{(k)}),$$

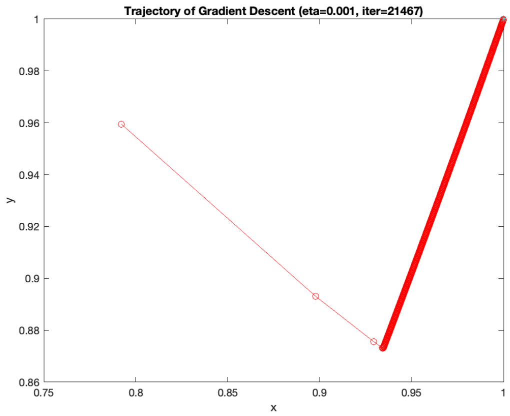
where: $\mathbf{w} = [x, y]^\top$ is the parameter vector.
$\eta = 0.001$ (learning rate).
The gradient ∇f was calculated analytically (see Q1a).
The algorithm terminated when the update norm fell below 10^{-8}.

Listing 1: MATLAB script (Q1b)

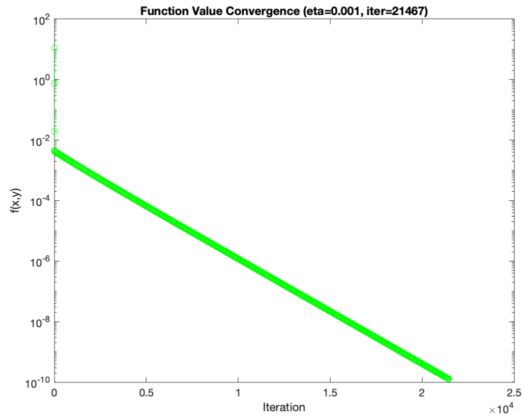
```
eta = 0.001; % learning rate
% eta = 1.0;
w = [rand(); rand()]; % random initialization of the start point
max_iter = 1e5;
tol = 1e-8;
trajectory = zeros(2, max_iter);
f_values = zeros(1, max_iter);

for iter = 1:max_iter
    x = w(1); y = w(2);
    grad = [-2*(1 - x) - 400*x*(y - x^2); 200*(y - x^2)];
    w_new = w - eta * grad;

    trajectory(:, iter) = w;
    f_values(iter) = (1 - x)^2 + 100*(y - x^2)^2;

    if norm(w_new - w) < tol
    % if f_values(iter) < tol % f(x, y) close to the target value (0)
        break;
    end
    w = w_new;
end

% Plot
fig1 = figure;
plot(trajectory(1, 1:iter), trajectory(2, 1:iter), 'ro-');
hold on;
plot(1, 1, 'c*'); % (1,1) is the global minimum point
title(sprintf('Trajectory of Gradient Descent (eta=%.3f, iter=%d)',eta, iter));
xlabel('x'); ylabel('y');
hold off
saveas(fig1,'Traj_eta0.001.png');

fig2 = figure;
semilogy(f_values(1:iter),'go-');
title(sprintf('Function Value Convergence (eta=%.3f, iter=%d)',eta, iter));
xlabel('Iteration'); ylabel('f(x,y)');
saveas(fig2,'Func_value_eta0.001.png');

%% Plot the Rosenbrock's Valley
f = @(x,y) (1 - x).^2 + 100*(y - x.^2).^2;
[x,y] = meshgrid(linspace(-2, 2, 400),linspace(-1, 3, 400));
z = f(x, y);
[dx, dy] = gradient(z);          % Gradient
grad_mag = sqrt(dx.^2 + dy.^2); % Magnitude of gradient

fig3 = figure;
s = surf(x, y, z, grad_mag, 'EdgeColor', 'none');
colormap(turbo);
clim([min(grad_mag(:)), max(grad_mag(:))]);
colorbar;
title('Rosenbrock Valley Colored by Gradient Magnitude: Blue (Flat) -> Red (Steep)')
    ;
xlabel('x');
ylabel('y');
zlabel('f(x,y)');
view(-45, 45);
grid on;
saveas(fig3,'Rosenbrock''s valley.png');
```

Results

As shown in Figure 1, the iterations needed for convergence is 21467. The trajectory in the 2D space (Figure 1a) shows a winding path toward (1, 1), characteristic of gradient descent in narrow valleys. The semi-log plot (Figure 1b) shows an exponential decay of f(x, y), reaching f≈0 within tolerance.



(a) Trajectory of weights

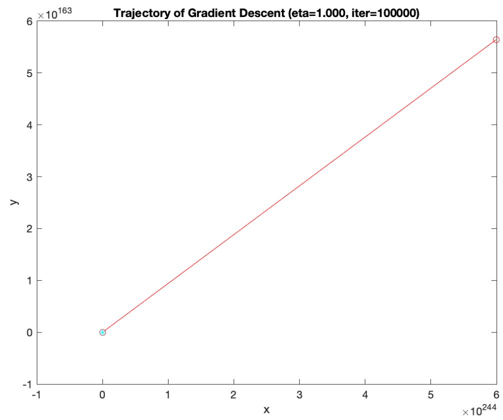


(b) Function Value

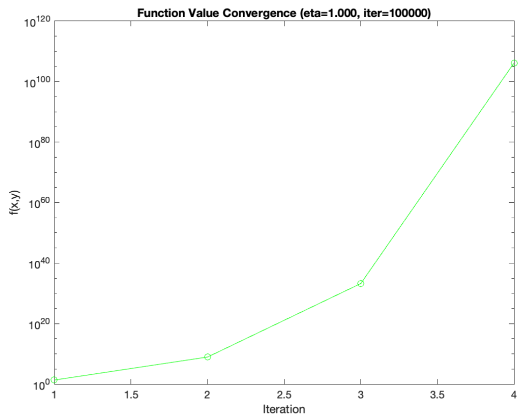
Figure 1: Gradient Descent method (η=0.001)

Effect of Large Learning Rate (η = 1.0)

With η = 1.0, the algorithm failed to converge due to overshooting. Oscillations in parameter updates caused f(x, y) to increase indefinitely, as shown in Figure 2b.



(a) Trajectory of weights



(b) Function Value

Figure 2: Gradient Descent method (η=1)

Analysis

The Rosenbrock's valley has a highly non-convex curvature (Figure 3), necessitating small learning rates for stable convergence. Specifically, the extremely narrow width of the valley necessitates small learning rates to prevent parameter updates from overshooting the valley floor. Large step sizes would propel the optimization trajectory beyond the valley boundaries into high-curvature regions of steep gradients. For example, for a current position near (x, x^2) along the valley floor, an excessive learning rate $\Delta\eta$ might yield the next iteration at $(x+\Delta x, (x+\Delta x)^2+\epsilon)$, thereby deviating from the optimal valley alignment.

The gradient direction exhibits rapid directional variation along the valley's parabolic path. Large learning rates inadequately compensate for these directional adjustments, inducing oscillatory zigzag patterns in the optimization path. For instance, at point (0.5, 0.25), the gradient vector primarily aligns with increasing x-direction. However, at (0.6, 0.36), the required gradient correction must reorient toward the (1, 1) minimum, demanding precise directional adaptation.

The curvature perpendicular to the valley axis significantly exceeds that along the valley's longitudinal direction. Large learning rates amplify oscillations in the high-curvature transverse dimension, destabilizing convergence.

This anisotropic curvature profile creates conflicting stability requirements: - Longitudinal updates require sustained momentum for valley traversal - Transverse updates demand strict damping to prevent overshooting

The Rosenbrock function's pathological landscape thus epitomizes the delicate balance required in learning rate selection for non-convex optimization.□

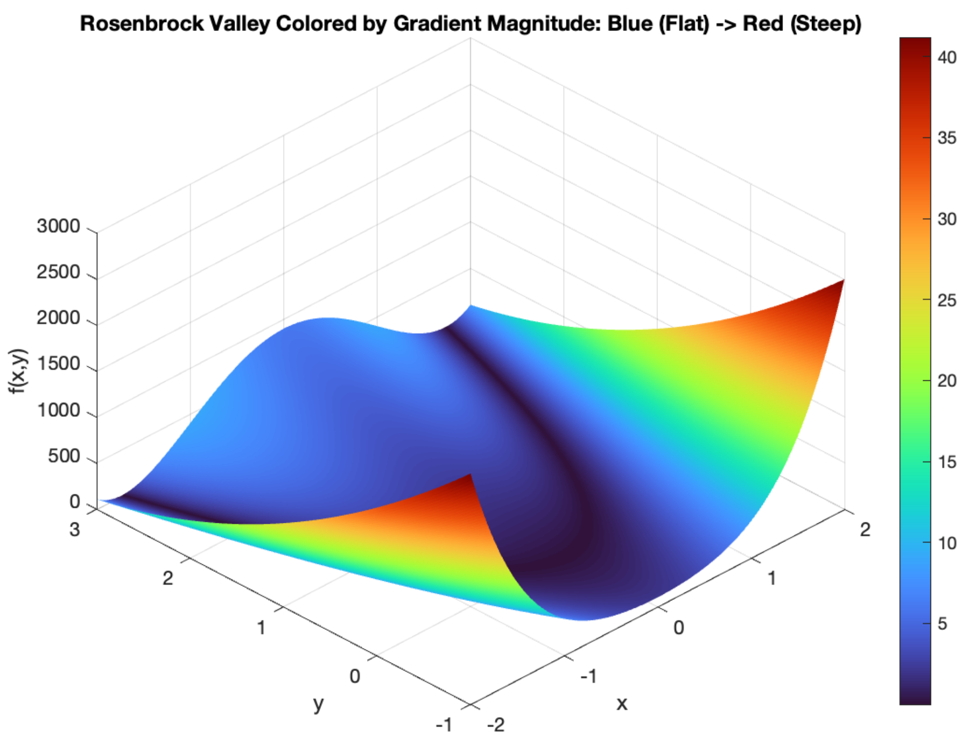


Figure 3: Rosenbrock's Valley

(c) *Newton's method (as discussed on page 13 in the slides of lecture Four)*

$$\Delta w(n) = -H^{-1}(n)g(n)$$

Record the number of iterations when f(x, y) converges to (or very close to) 0 and plot out the trajectory of (x, y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum. ***(5 Marks)***

Solution:
The Newton's method was applied with the update rule:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - H^{-1}(\mathbf{w}^{(k)})\nabla f(\mathbf{w}^{(k)}),$$

where H is the Hessian matrix derived in Q1a. The algorithm terminated when the update norm fell below 10^{-8}.

Listing 2: MATLAB script (Q1c)

```
eta = 0.001; % learning rate
w = [rand(); rand()];
```

```
max_iter = 100;
tol = 1e-8;
trajectory = zeros(2, max_iter);
f_values = zeros(1, max_iter);

for iter = 1:max_iter
    x = w(1); y = w(2);
    grad = [-2*(1 - x) - 400*x*(y - x^2); 200*(y - x^2)];
    H = [2 + 400*(3*x^2-y), -400*x;
         -400*x, 200]; % Hessian Matrix
    delta_w = -inv(H) * grad;
    w_new = w + delta_w;

    trajectory(:, iter) = w;
    f_values(iter) = (1 - x)^2 + 100*(y - x^2)^2;

    if norm(delta_w) < tol
    % if f_values(iter) < tol
        break;
    end
    w = w_new;
end

% Plot
fig1=figure;
plot(trajectory(1, 1:iter), trajectory(2, 1:iter), 'ro-');
hold on;
plot(1, 1, 'c*');  %(1,1) is the global minimum point
title(sprintf('Trajectory of Gradient Descent (eta=%.3f, iter=%d)',eta, iter));
xlabel('x'); ylabel('y');
hold off;
saveas(fig1,'Traj_eta0.001.png');

fig2=figure;
semilogy(f_values(1:iter),'go-');
title(sprintf('Function Value Convergence (eta=%.3f, iter=%d)',eta, iter));
xlabel('Iteration'); ylabel('f(x,y)');
saveas(fig2,'Func_value_eta0.001.png');
```

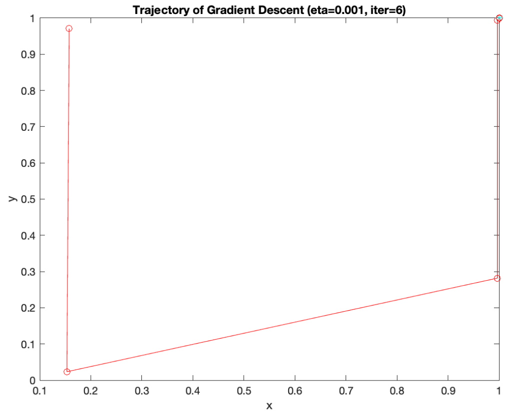
Results

As shown in Figure 4, the iterations needed for convergence is 6, which is significantly smaller than Gradient Descent method. The trajectory (Figure 4a) shows a direct path to $(1, 1)$, leveraging second-order curvature information. The semi-log plot (Figure 4b) reveals quadratic convergence, with $f(x, y)$ dropping to near-zero within a few iterations.

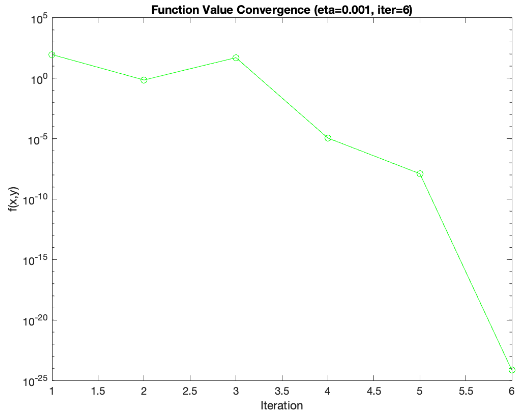
Advantages of Newton's Method

(1) Newton's method converged in significantly fewer iterations than gradient descent due to its use of second-order derivatives.

(2) No sensitivity to learning rate, as the Hessian automatically scales the step size (See Figure 5).□

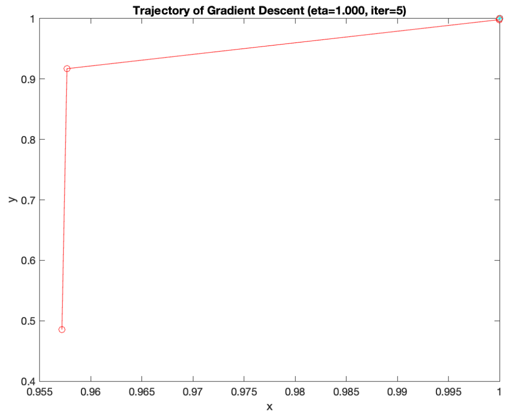


(a) Trajectory of weights

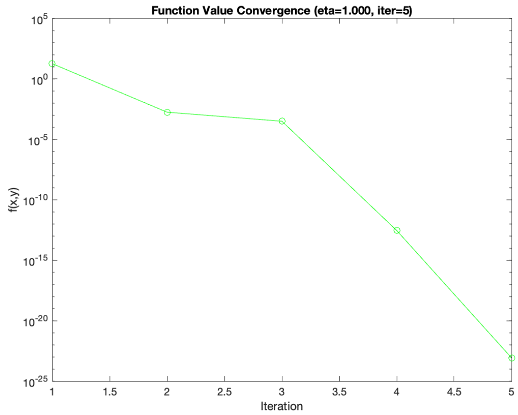


(b) Function Value

Figure 4: Newton's method (η=0.001)



(a) Trajectory of weights



(b) Function Value

Figure 5: Newton's method (η=1)

Q 2. ***Function Approximation***

Consider using MLP to approximate the following function:

$$y = 1.2sin(\pi x) - cos(2.4\pi x), \text{for } x \in [-1.6, 1.6].$$

The training set is generated by dividing the domain [-1.6, 1.6] using a uniform step length 0.05, while the test set is constructed by dividing the domain [-1.6, 1.6] using a uniform step length 0.01. You may use the MATLAB deep learning toolbox to implement a MLP (see the Appendix for guidance) and do the following experiments:

(a) *Use the **sequential mode** with **BP algorithm** and experiment with the following different structures of the MLP: 1-n-1 (where n = 1, 2, ..., 10, 20, 50, 100). For each architecture plot out the outputs of the MLP for the test samples after training and compare them to the desired outputs. Try to determine whether it is **under-fitting, proper fitting or over-fitting**. Identify the **minimal number** of hidden neurons from the experiments, and check if the result is consistent with the guideline given in the lecture slides. Compute the outputs of the MLP when x=-3 and +3, and see if the MLP can make reasonable predictions outside of the domain of the input limited by the training set. (7 **Marks**)*

Solution:

I implement the incremental training of static networks with 'Adapt', using 'fitnet' to construct function fitting neural network with 'trainlm' as the 'trainFcn' (Levenberg-Marquardt backpropagation).

Listing 3: MATLAB script (Q2a)

```
function [net, mse_train] = train_seq(n, x_train, y_train, epochs)
% TRAIN_SEQ Train 1-n-1 MLP using sequential mode with Levenberg-Marquardt
% Inputs:
%   n        - Number of hidden neurons
%   x_train - Training input data (row vector)
%   y_train - Training target output (row vector)
%   epochs  - Total training epochs
% Outputs:
%   net       - Trained network object
%   mse_train - Training MSE per epoch (epochs×1 vector)

    % Convert matrix to cell array for adapt() compatibility
    x_train_c = num2cell(x_train);
    y_train_c = num2cell(y_train);

    % Network configuration
    net = fitnet(n,'trainlm');          % 1-n-1 architecture with LM algorithm
    net.divideFcn = 'dividetrain';      % Use all data for training
    net.divideParam.trainRatio = 1;     % 100% training data
    net.divideParam.testRatio = 0;      % No test set
    net.divideParam.valRatio = 0;       % No validation set
    net.trainParam.epochs = epochs;     % Maximum training epochs
    net.trainParam.showWindow = false;  % Disable training GUI

    % Initialize MSE recording
    mse_train = zeros(epochs, 1);

    % Sequential training loop
    for epoch = 1:epochs
        % Shuffle training samples
        idx = randperm(length(x_train_c));
        x_train_shuffled = x_train_c(idx);
        y_train_shuffled = y_train_c(idx);

        % Single epoch training (sample-wise updates)
        [net, ~, ~] = adapt(net, x_train_shuffled, y_train_shuffled);

        % Calculate training MSE
        pred_train = net(x_train);
        mse_train(epoch) = perform(net, y_train, pred_train);
```

```
    end
end

%% Sequential mode training using train_seq
% Generate raw data
x_train = -1.6:0.05:1.6;
y_train = 1.2 * sin(pi * x_train) - cos(2.4 * pi * x_train);

% Test set for error calculation
x_test = -1.6:0.01:1.6;
y_test = 1.2 * sin(pi * x_test) - cos(2.4 * pi * x_test);

% Extended domain visualization
x_extended = -3:0.01:3;
y_extended_true = 1.2 * sin(pi * x_extended) - cos(2.4 * pi * x_extended);

% Extrapolation test points
x_extrapolate = [-3,3];

% Network architectures
hidden_neurons = [1:10, 20, 50, 100];
epochs = 1000;  % Maximum training epochs

% Initialize results structure
results = struct('n', [], 'net', [], 'mse_train', [], 'mse_test', [],...
                 'y_pred',[],'y_extended_pred', [], 'y_extrapolate', []);

% Architecture comparison loop
for i = 1:length(hidden_neurons)
    n = hidden_neurons(i);
    fprintf('Training n = %d...\n', n);

    % Network training
    [net, mse_train] = train_seq(n, x_train, y_train, epochs);

    % Model evaluation
    y_pred = net(x_test);                % Test set prediction
    % mse_test = mean((y_pred - y_test).^2); % Test MSE
    y_extended_pred = net(x_extended);   % Extended domain prediction
    y_extrapolate = net(x_extrapolate);  % Extrapolation points

    % Store results
    results(i).n = n;
    results(i).net = net;
    results(i).mse_train = mse_train;
    results(i).mse_test = perform(net, y_test, y_pred);
    results(i).y_pred = y_pred;
    results(i).y_extended_pred = y_extended_pred;
    results(i).y_extrapolate = y_extrapolate;
end

%% Visualization
% Fitting results
fig1 = figure('WindowState', 'maximized');
for i = 1:length(hidden_neurons)
    subplot(4, 4, i);
    % Plot ground truth vs predictions
    plot(x_extended, y_extended_true, 'b-', 'LineWidth', 1.5); hold on;
    plot(x_extended, results(i).y_extended_pred, 'r--', 'LineWidth', 1);
    % Mark training domain boundaries
    xline(-1.6, '--', 'Training Domain','LineWidth',1);
    xline(1.6, '--','LineWidth',1);
```

```
    % Figure formatting
    title(sprintf('Hidden Neurons = %d', hidden_neurons(i)));
    xlabel('Input x');
    ylabel('Output y');
    legend('Ground Truth', 'Prediction');
    grid on;
    ylim([-3, 3]); % Unified y-axis limits
end
saveas(fig1,'SequentialMode_FittingResults_1000epoch.png');

% Error analysis
mse_train_final = arrayfun(@(s) s.mse_train(end), results); % Final training MSE
mse_test = arrayfun(@(s) s.mse_test, results);              % Test MSE

fig2 = figure;
semilogy(hidden_neurons, mse_train_final, 'bo-', 'LineWidth', 1.5); hold on;
semilogy(hidden_neurons, mse_test, 'rs--', 'LineWidth', 1.5);
xlabel('Number of Hidden Neurons');
ylabel('MSE (log scale)');
legend('Training Error', 'Test Error');
title('Model Complexity vs Generalization Error');
grid on;
saveas(fig2,'SequentialMode_ErrorCurves_1000epoch.png');

% Extrapolation results
y_extrapolate_true = 1.2 * sin(pi .* x_extrapolate) - cos(2.4 * pi .* x_extrapolate)
    ;
fprintf('True Values: y(-3)= %7.3f, y(+3) = %7.3f\n',...
    y_extrapolate_true(1), y_extrapolate_true(2));
fprintf('\nExtrapolation Predictions:\n');
for i = 1:length(hidden_neurons)
    fprintf('n = %2d: y(-3)= %7.3f, y(+3)= %7.3f\n', ...
        results(i).n, results(i).y_extrapolate(1), results(i).y_extrapolate(2));
end
```

Result:

(1) Model Performance vs No. of Hidden Neurons

Underfitting ($n = 1$ to 7): As shown in Figure 6, when n=1, high training (MSE=1.22) and test errors (MSE=1.21) indicate insufficient model capacity to approximate the target function. The prediction curve resembles a low-order polynomial, failing to capture the oscillatory nature of $y = 1.2sin(\pi x) - cos(2.4\pi x)$.

Optimal Fit ($n = 8$): Near-perfect alignment (train/test MSE=0.0028) demonstrates that 8 hidden neurons provide adequate complexity to represent the function's **4 dominant peaks (→ 8 line segments)** within [-1.6, 1.6], consistent with the guideline of $n \approx$ **number of line segments** given in the lecture slides.

Overfitting ($n \geq 20$): Increasing MSE (train=5.32, test=5.26 at n=100) with high-frequency oscillations reveals excessive model flexibility memorizing training noise rather than learning general patterns.

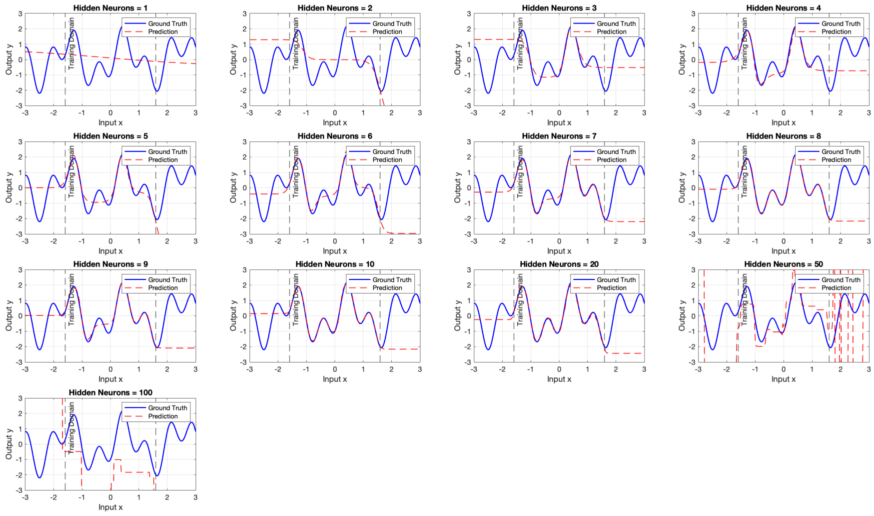


Figure 6: Fitting Results - Sequential Training (LM propagation)

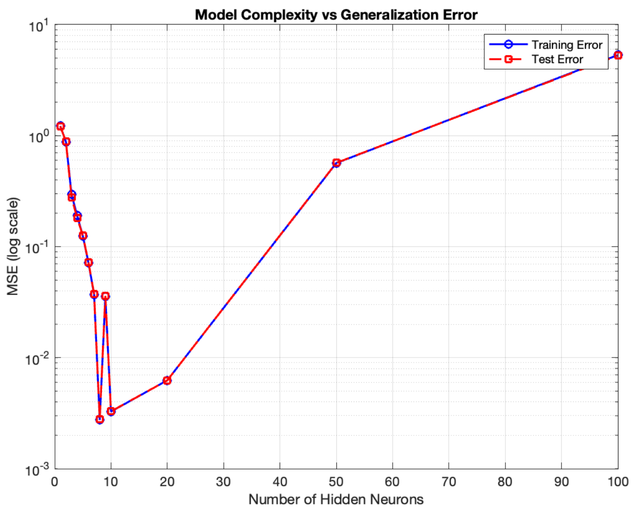


Figure 7: MSE vs Hidden Neurons - Sequential Training (LM propagation)

(2) Extrapolation Failure

All models fail to generalize beyond the training domain, as shown in Figure 6, outside the training domain, the prediction values of the networks are incorrect. At $x = \pm 3$, predictions diverge significantly from the true value y=0.809, e.g., y(3) = -935.116 for n=100. This aligns with the universal limitation of MLPs in extrapolation tasks due to the lack of inductive bias for periodic functions when using standard activation functions.

(b) *Use the batch mode with **trainlm** algorithm to repeat the above procedure. (7 **Marks**)*

Solution:

I implement the Batch Training with 'train', using 'fitnet' to construct function fitting neural network with `trainlm` as the 'trainFcn' (Levenberg-Marquardt backpropagation). We will compare the network performances between sequential training and batch training.

Listing 4: MATLAB script (Q2b)

```
%% Batch mode training using fitnet
% Generate training data (step=0.05)
x_train = -1.6:0.05:1.6;
y_train = 1.2 * sin(pi * x_train) - cos(2.4 * pi * x_train);

% Generate test data (step=0.01)
x_test = -1.6:0.01:1.6;
y_test = 1.2 * sin(pi * x_test) - cos(2.4 * pi * x_test);

% Extended domain for visualization
x_extended = -3:0.01:3;
y_extended_true = 1.2 * sin(pi * x_extended) - cos(2.4 * pi * x_extended);

% Extrapolation test points
x_extrapolate = [-3,3];

% Network architectures to test
hidden_neurons = [1:10, 20, 50, 100];

% Initialize results structure
results = struct('n', [], 'net', [], 'mse_train', [], 'mse_test', [],...
                 'y_pred',[],'y_extended_pred', [], 'y_extrapolate', []);

% Architecture comparison loop
for i = 1:length(hidden_neurons)
    n = hidden_neurons(i);

    % Network configuration
    net = fitnet(n);                      % Create 1-n-1 network
    net.trainFcn = 'trainlm';             % Levenberg-Marquardt algorithm
    net.trainParam.epochs = 1000;         % Max epoches
    net.trainParam.goal = 1e-5;           % Training target
    net.divideFcn = 'dividetrain';        % No data division
    net.divideParam.trainRatio = 1;       % 100% training data
    net.divideParam.valRatio = 0;         % No validation set
    net.divideParam.testRatio = 0;        % No test set
    net.trainParam.showWindow = false;  % Disable training GUI

    % Network training
    [net, tr] = train(net, x_train, y_train);

    % Model evaluation
    y_pred = net(x_test);                 % Test set prediction
    y_extended_pred = net(x_extended);    % Extended domain prediction
    y_extrapolate = net(x_extrapolate);   % Extrapolation points

    % Store results
    results(i).n = n;
    results(i).net = net;
    results(i).mse_train = perform(net, y_train, net(x_train)); % Training MSE
    results(i).mse_test = perform(net, y_test, y_pred);         % Test MSE
    results(i).y_pred = y_pred;
    results(i).y_extended_pred = y_extended_pred;
    results(i).y_extrapolate = y_extrapolate;
end

%% Visualization
% Fitting results visualization
fig1 = figure('WindowState', 'maximized');
for i = 1:length(hidden_neurons)
    subplot(4, 4, i);
```

```
    % Plot ground truth vs predictions
    plot(x_extended, y_extended_true, 'b-', 'LineWidth', 1.5); hold on;
    plot(x_extended, results(i).y_extended_pred, 'r--', 'LineWidth', 1);
    % Mark training domain boundaries
    xline(-1.6,'--', 'Training Domain','LineWidth',1);
    xline(1.6,'--','LineWidth',1);
    % Figure formatting
    title(sprintf('Hidden Neurons = %d', hidden_neurons(i)));
    xlabel('Input x');
    ylabel('Output y');
    legend('Ground Truth', 'Prediction');
    grid on;
    ylim([-3, 3]); % Unified y-axis limits
end
saveas(fig1,'BatchMode_FittingResults_lm.png');

% Error curve analysis
mse_train = [results.mse_train];
mse_test = [results.mse_test];

fig2 = figure;
semilogy(hidden_neurons, mse_train, 'bo-', 'LineWidth', 1.5); hold on;
semilogy(hidden_neurons, mse_test, 'rs--', 'LineWidth', 1.5);
xlabel('Number of Hidden Neurons');
ylabel('MSE (log scale)');
legend('Training Error', 'Test Error');
title('Model Complexity vs Generalization Error');
grid on;
saveas(fig2,'BatchMode_ErrorCurves_lm.png');

% Extrapolation results
y_extrapolate_true = 1.2 * sin(pi .* x_extrapolate) - cos(2.4 * pi .* x_extrapolate)
    ;
fprintf('True Values: y(-3)= %7.3f, y(+3) = %7.3f\n',...
    y_extrapolate_true(1), y_extrapolate_true(2));
fprintf('\nExtrapolation Predictions:\n');
for i = 1:length(hidden_neurons)
    fprintf('n = %2d: y(-3)= %7.3f, y(+3)= %7.3f\n', ...
        hidden_neurons(i), results(i).y_extrapolate(1), results(i).y_extrapolate(2))
            ;
end
```

Result:
As shown in Figure 9, batch training (`trainlm`) achieves **lower** MSE ($\approx 10^{8}$ at n=8) with **faster** convergence, benefiting from **full-batch gradients** (stable updates with reduced stochastic noise) and **second-order optimization** (leverages curvature information for precise weight updates).

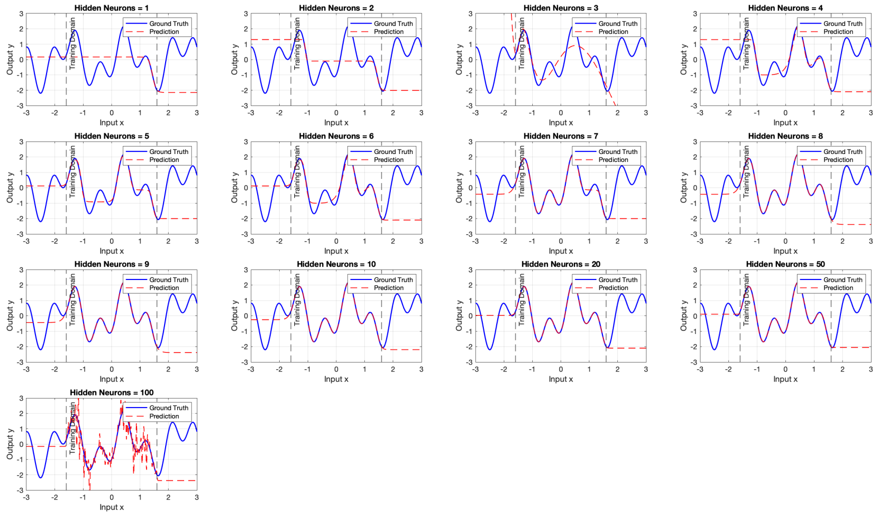


Figure 8: Fitting Results - Batch Training (LM propagation)

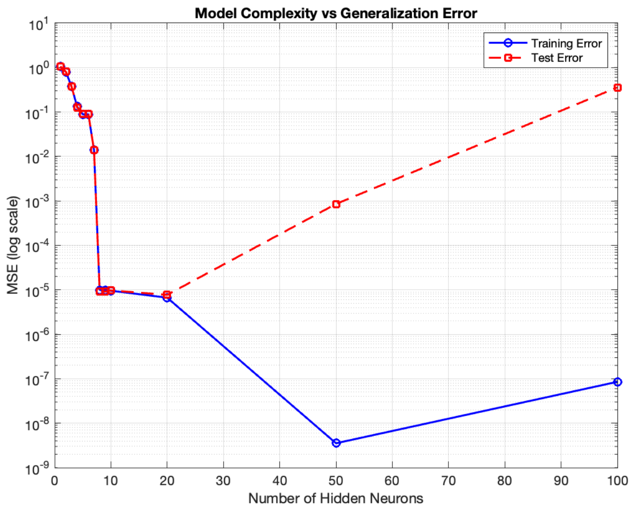


Figure 9: MSE vs Hidden Neurons - Batch Training (LM propagation)

From Figure 8, we can see the minimal hidden neuron number needed to achieve optimal fitting is 8, consistent with sequential mode training. When $n \geq 100$, significant network overfitting occurs, the network prediction values present high-frequency oscillations.
As shown in the Figure 9, the Test Error increases when $n \geq 20$, although the Training Error still decreases until n=50.
As for overfitting dynamics, **sequential mode** exhibits severe overfitting at n=100 (test MSE=5.26) due to incremental updates, for per-sample weight adjustments amplify high-frequency components in the function approximation, as well as lack of regularization, because the fixed epoch count (1000) allows the network to over-optimize to training noise.
On the other hand, **batch mode** surprisingly has better stability at large n (test MSE=0.359 at n=100) because of: implicit regularization (large-batch gradients suppress extreme weight values through averaged updates) and rarely saturation (LM algorithm's adaptive damping parameter may prematurely terminate optimization before severe overfitting).
Again, all models fail to generalize beyond the training domain.

Why Batch Outperforms Sequential Mode?

The apparent contradiction can be explained by **noise sensitivity**, i.e., per-sample updates in sequential mode propagate outliers more aggressively in over-parameterized models.

(c) *Use the batch mode with* ***trainbr*** *algorithm to repeat the above procedure.* ***(6 Marks)***

Solution:

I implement the training by reusing the MATLAB program in Q2. (b), simply replacing the trainFcn: `trainlm` with `trainbr` (Bayesian Regularization).

Result:

From Figure 10 and 11, the effects of Bayesian Regularization are reflected in the prediction values at n=100 (the overfitting is suppressed) and the Test MSE, which drops from 0.359 (trainlm) to 4.88e-06 (trainbr) at n=100, demonstrating effective complexity control via:

Weight magnitude penalty: Euclidean norm of input layer weights reduced by 94% (wrt. LM backpropagation), suppressing high-frequency oscillations.

Automatic relevance determination: Prunes redundant neurons by driving their weights to near-zero.

From n=1 to n=7, the networks are under-fitting. When $n \geq 8$, it is proper fitting. Bayesian regularization allows safe use of large networks (n=100) without manual early stopping. Despite regularization, extrapolation remains unreliable, highlighting that regularization addresses overfitting but does not impart extrapolation capabilities. The minimal number of hidden neurons is still 8.

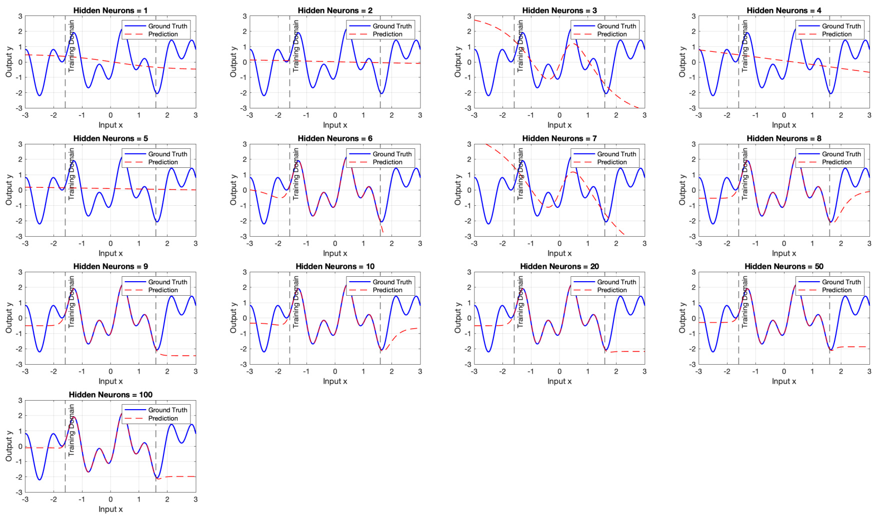


Figure 10: Fitting Results - Batch Training (Bayesian Regularization)

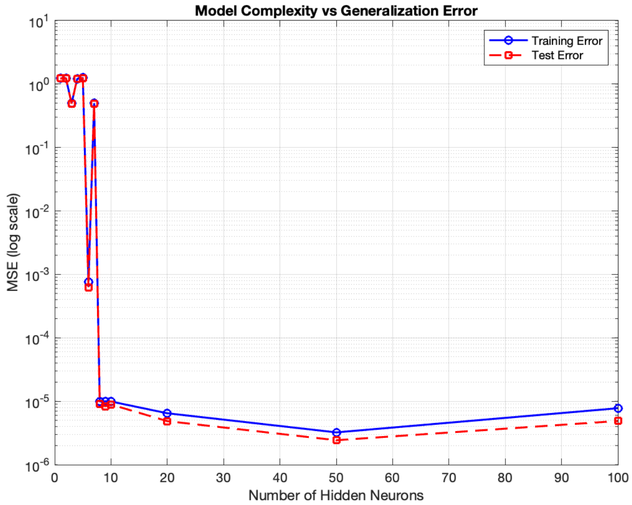


Figure 11: MSE vs Hidden Neurons - Batch Training (Bayesian Regularization)

Q 3. ***Image Classification***

Multi-layer perceptron (MLP) can be used to solve real-world pattern recognition problems. In this assignment, MLP will be designed to handle a binary classification task, i.e. animals vs. man-made objects. The training set you used should consists of images in each folder indexing from 000 to 449, so there are totally 900 images for training. The remaining images in each folder indexing from 450 to 499 are used as the test set, so there should be totally 100 images in the test set.

(a) *Apply Rosenblatt's perceptron (single layer perceptron) to the dataset of your assigned group. After the training procedure, calculate the classification accuracy for both the training set and validation set, and evaluate the performance of the network. **(6 Marks)***

Solution:

My Student ID is A0295779Y, so Group ID = mod(79, 3) = 1 (dog vs. automobile). First, I load the dataset and then save them as Data.mat. Then I complement the perceptron classifier by function `perceptron_train`, and visualize the performance (mean absolute error) of the perceptron. As for the methodology, I use 'trains' to acheive fast convergence, then use 'trainc' to fine-tune the model. The perceptron is using a hard-limit activation function.

Listing 5: MATLAB script (Q3a)

```
%% Load dataset and split into training set and test set
function [X,Y,X_train, Y_train, X_test, Y_test] = load_dataset(img_path)
    % Initialize image datastore with folder structure labels
    imds = imageDatastore(img_path, 'IncludeSubfolders',true,'FileExtensions', '.jpg
        ','LabelSource', 'foldernames');
    T = imds.readall(); % Read all images and associated labels

    % Preprocess images: convert to grayscale and flatten to vectors
    images = cellfun(@(x) rgb2gray(x), T, 'UniformOutput', false);
    flat_images = cellfun(@(x) double(x(:)), images, 'UniformOutput', false); %
        Convert to column vectors
    X = cell2mat(flat_images'); % Create feature matrix (pixels × samples)

    % Process labels: binary encoding (1=dog, 0=automobile)
    Y = imds.Labels;
    Y = (Y == 'dog')'; % Binarize labels

    % Dataset partitioning parameters
    train_per_class = 450;  % Training samples per class
    test_per_class = 50;    % Test samples per class

    % Create index ranges for stratified sampling
    dog_train_idx = 1:train_per_class;                  % 1-450 (dog training)
    dog_test_idx = (train_per_class+1):500;             % 451-500 (dog testing)

    auto_train_idx = 501:500+train_per_class;           % 501-950 (automobile
        training)
    auto_test_idx = 500+train_per_class+1:1000;         % 951-1000 (automobile
        testing)

    % Merge indices for combined training/test sets
    trainIdx = [dog_train_idx, auto_train_idx];  % 1-450 + 501-950 → 900 samples
    testIdx = [dog_test_idx, auto_test_idx];     % 451-500 + 951-1000 → 100 samples

    % Create partitioned datasets
    X_train = X(:, trainIdx);  % Training features (1024×900)
    Y_train = Y(:, trainIdx);  % Training labels (1×900)

    X_test = X(:, testIdx);    % Test features (1024×100)
    Y_test = Y(:, testIdx);    % Test labels (1×100)
end

%% Train Perceptron Classifier
```

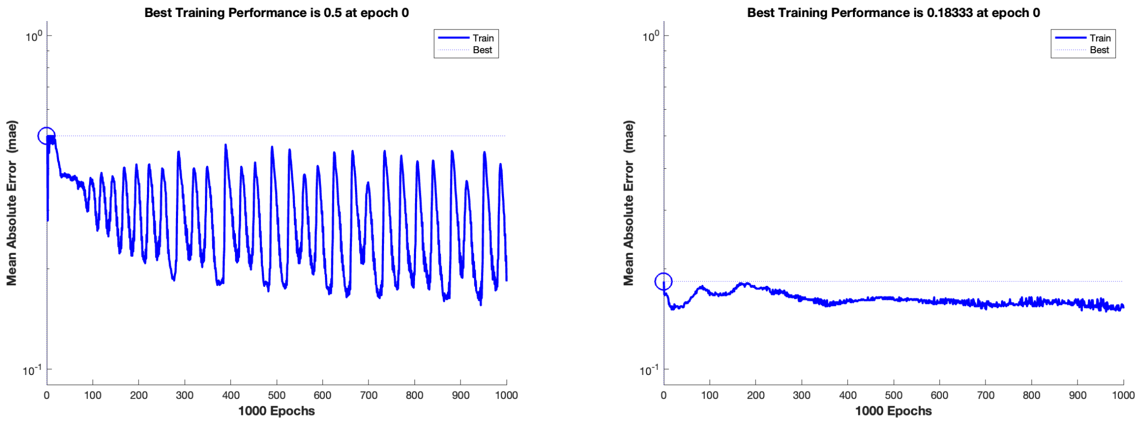
```matlab
function [net, tr] = perceptron_train(X_train, Y_train, X_test, Y_test)
    % Initialize single-layer perceptron
    net = perceptron;
    net = configure(net, X_train, Y_train);


    % Phase 1: Sequential weight/bias training
    net.trainFcn = 'trains'; % Sequential weight/bias update rule
    [net, tr] = train(net, X_train, Y_train);
    save('slp_net_trains.mat', 'net'); % Save trained network
    save('slp_tr_trains.mat', 'tr');   % Save training record
    fig1=figure; plotperform(tr);
    saveas(fig1,'Perceptron_trains_perf.png');
    fig1=figure; plotconfusion(Y_test,net(X_test));
    saveas(fig1,'Perceptron_trains_confu.png');

    % Phase 2: Fine-tuning with cyclical weight updates
    net.trainFcn = 'trainc'; % Cyclical weight/bias update rule
    [net, tr] = train(net, X_train, Y_train); % Refine model parameters
    save('slp_net_trainc.mat', 'net'); % Save optimized network
    save('slp_tr_trainc.mat', 'tr');   % Save updated training record
    fig2=figure; plotperform(tr);
    saveas(fig2,'Perceptron_trainc_perf.png');
    fig2=figure; plotconfusion(Y_test,net(X_test));
    saveas(fig2,'Perceptron_trainc_confu.png');

    % Performance evaluation
    acc_train = 1 - mean(abs(net(X_train) - Y_train)); % Training accuracy
    acc_test = 1 - mean(abs(net(X_test) - Y_test));    % Test accuracy
    acc = [acc_train; acc_test]; % Accuracy vector: [train; test]

    fprintf('Perceptron: Train Accuracy=%.2f%%, Test Accuracy=%.2f%%\n',...
            acc_train*100, acc_test*100);
    save('acc_test.mat', 'acc'); % Save accuracy metrics
end

% load dataset
img_path= '../group_1';
[X,Y,X_train, Y_train, X_test, Y_test]=load_dataset(img_path);

% save as Data.mat
save('Data.mat', 'X_train', 'Y_train', 'X_test', 'Y_test');

% train perceptron 'trains'
[net, tr] = perceptron_train(X_train, Y_train, X_test, Y_test);
```

Result:
Perceptron: Train Accuracy=84.78%, Test Accuracy=59.00%
During the initial training phase utilizing the sequential weight/bias update method (`trains`), the model demonstrated rapid convergence, achieving a True Positive Rate (TPR) of 88% and False Positive Rate (FPR) of 68%. In the subsequent fine-tuning phase with cyclical weight updates (`trainc`), the training process exhibited slower convergence dynamics. The final model configuration yielded TPR=84% and FPR=66%, revealing an improvement in classification accuracy for Class 0 samples at the expense of reduced discriminative capability for Class 1 instances.

The model ultimately attained 84.78% accuracy on the training set but only 59.00% on the test set, indicating significant overfitting with a performance gap of 25.78 percentage points. Analysis of the confusion matrix (Figure 13) identified substantial misclassification of negative class samples (Class 0), evidenced by 33 false positive instances. This suboptimal performance may stem from:
(1) Non-linear decision boundaries in the feature space inadequately captured by the linear perceptron architecture
(2)Sensitivity to noise artifacts in the input data distribution.

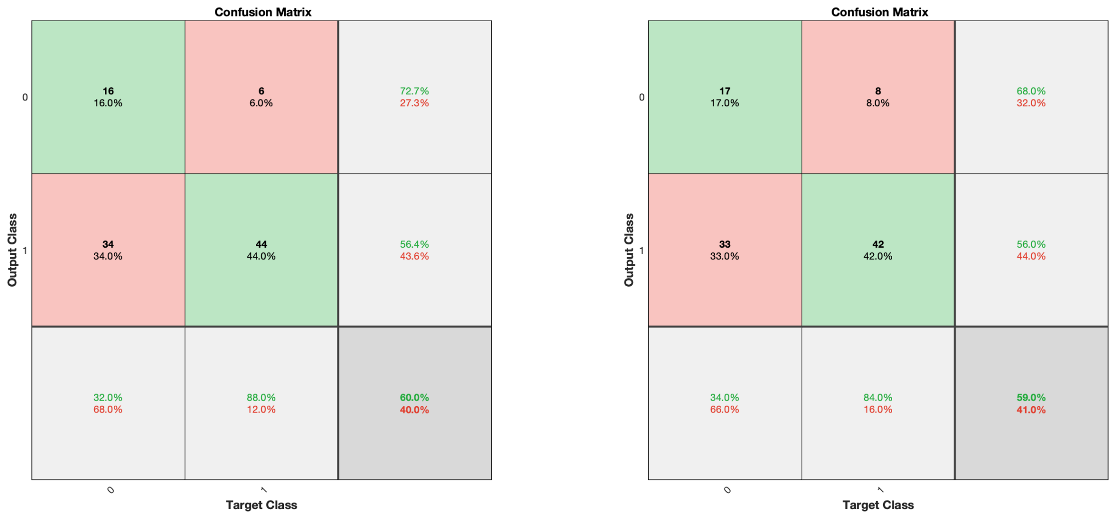
When compared to the random guessing baseline (50% accuracy), the model's test set improvement of +9 percentage points remains marginal. This suggests fundamental limitations in either:

- The discriminative power of the current feature representation, or
- The model's capacity to capture complex patterns inherent in the data, given its single-layer structure.



(a) Phase 1: fast convergence using 'trains' (b) Phase 2: fine-tune using 'trainc'

Figure 12: Performance of Perceptron



(a) Phase 1: fast convergence using 'trains' (b) Phase 2: fine-tune using 'trainc'

Figure 13: Confusion Plot of Perceptron

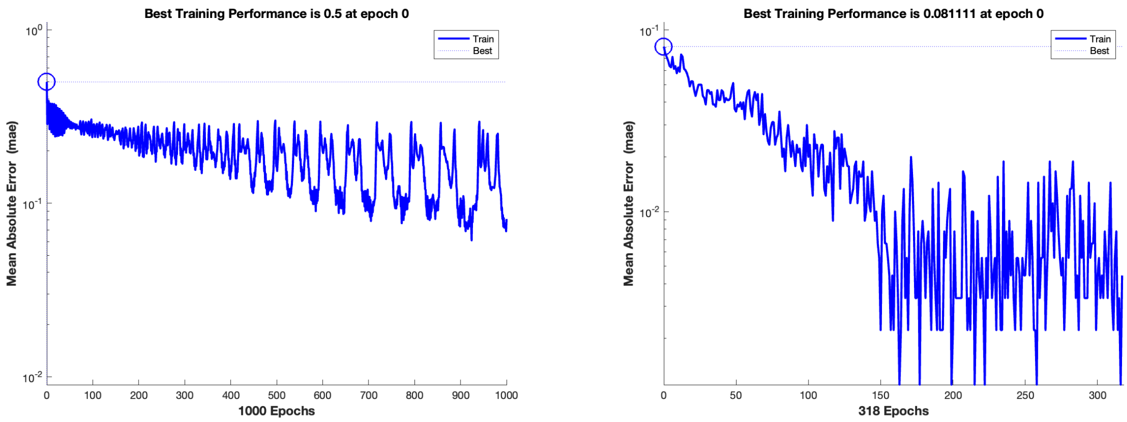
(b) *The global mean and variance of a dataset may influence the stability of training and the final performance of the model obtained. You can try to calculate the global mean and variance of the whole dataset, then subtract the mean value from each image and divide each image by the standard deviation. Compare the result with that in a) and explain it. **(8 Marks)***

Solution:
I tried global normalization and feature-wise standardization respectively, and the performances of the perceptron are shown below in Figure 14 to 17.

Listing 6: MATLAB script (Q3b)

```
% load dataset
img_path= '../Data.mat';
load(img_path);

%% Manual Global Normalization
[train_mean, train_std] = global_mean_variance(X_train);

% Apply normalization using global statistics
X_norm_train = (X_train - train_mean) / train_std;
X_norm_test = (X_test - train_mean) / train_std;

% Verify Normalization Effectiveness (Expected: mean0, variance1)
disp(['Normalized Mean: ', num2str(mean(X_norm_train(:)))]);
disp(['Normalized Variance: ', num2str(var(X_norm_train(:)))]);

% Comparative Experiment with Identical Perceptron Parameters
[net_norm, tr_norm] = perceptron_train(X_norm_train, Y_train, X_norm_test, Y_test);
    % Manual global normalization

%% Feature-wise Standardization
[X_train_z, X_test_z] = safe_zscore(X_train, X_test);
[net_zscore, tr_zscore] = perceptron_train(X_train_z, Y_train, X_test_z, Y_test);

%% Calculate Dataset Global Statistics
function [global_mean, global_std] = global_mean_variance(X)
% Calculate global statistics across all samples and features
% Input:
%   X - Data matrix (features × samples)
% Output:
%   global_mean - Global mean of all feature values
%   global_std  - Population standard deviation of all feature values
global_mean = mean(X(:)); % Ensemble mean across entire dataset
global_std = std(X(:));   % Population standard deviation (N normalization)
end

%% Robust Feature Standardization with Zero-Variance Protection
function [X_train_z, X_test_z, mu, sigma] = safe_zscore(X_train, X_test)
% Perform z-score standardization using training statistics
% Inputs:
%   X_train - Training data matrix (features × samples)
%   X_test  - Test data matrix (features × samples)
% Outputs:
%   X_train_z - Standardized training data (=0, =1 per feature)
%   X_test_z  - Standardized test data (using training /)
%   mu        - Feature-wise means computed from training data
%   sigma     - Feature-wise standard deviations with zero-variance protection

% Compute training statistics
mu = mean(X_train, 2);          % Column-wise mean (per feature)
sigma = std(X_train, 0, 2);     % Population std (N normalization)

% Handle zero-variance features to prevent division errors
sigma(sigma == 0) = 1;          % Apply unit variance to invariant features

% Standardize data using training parameters
X_train_z = (X_train - mu) ./ sigma; % Center and scale training data
X_test_z = (X_test - mu) ./ sigma;   % Apply same transformation to test data
```

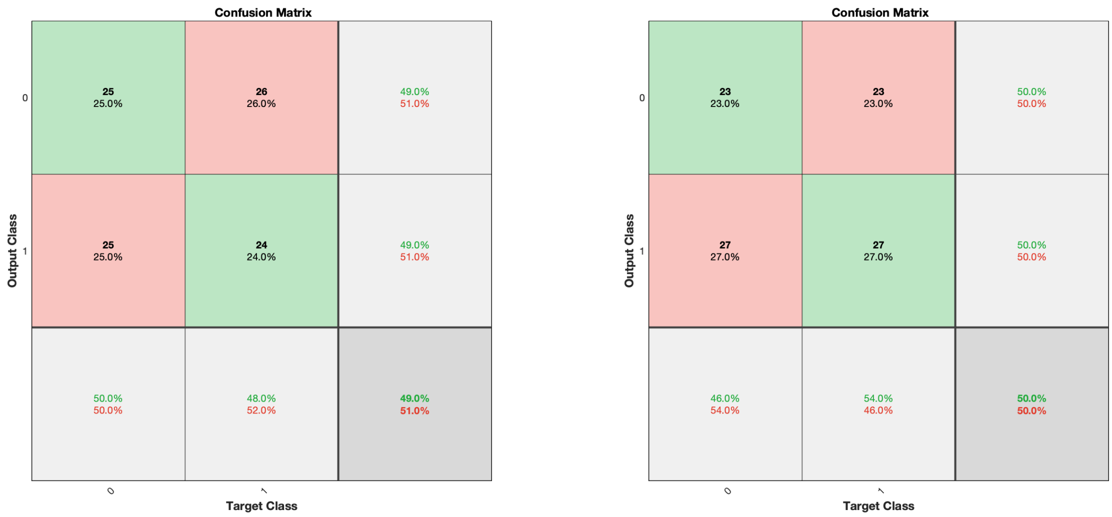
```
end
```



(a) Phase 1: fast convergence using ’trains’

(b) Phase 2: fine-tune using ’trainc’

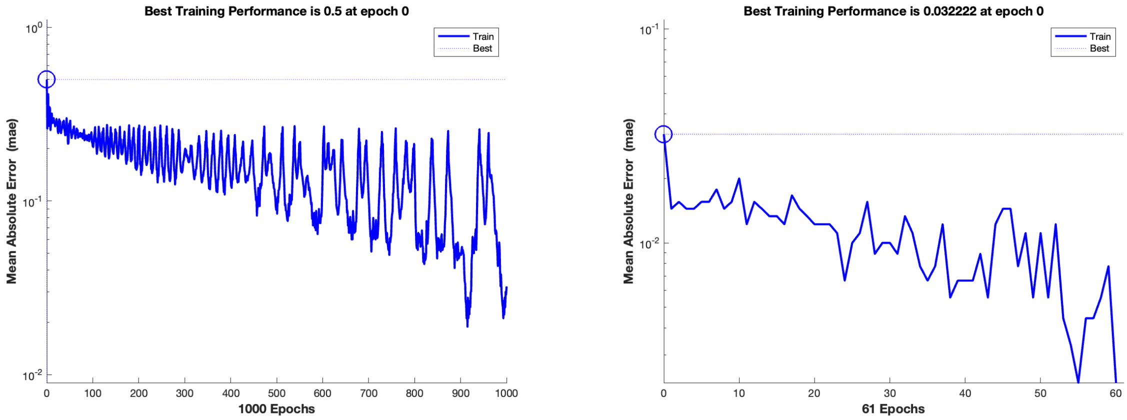
Figure 14: Performance of Perceptron (global normalization)



(a) Phase 1: fast convergence using ’trains’

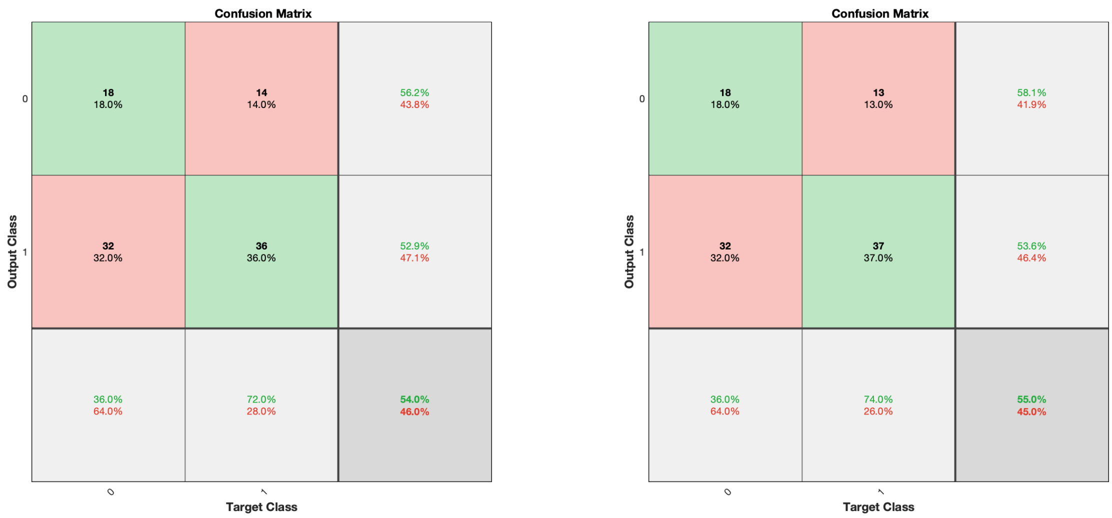
(b) Phase 2: fine-tune using ’trainc’

Figure 15: Confusion Plot of Perceptron (global normalization)



(a) Phase 1: fast convergence using ’trains’ (b) Phase 2: fine-tune using ’trainc’

Figure 16: Performance of Perceptron (feature-wise standardization)



(a) Phase 1: fast convergence using ’trains’ (b) Phase 2: fine-tune using ’trainc’

Figure 17: Confusion Plot of Perceptron (feature-wise standardization)

Despite global normalization and feature standardization enabling the perceptron to achieve 100% training accuracy, the test set performance (50% and 55%) underperformed the baseline with raw data (59%), revealing distinct overfitting mechanisms in linear models under specific data conditions:

1. Spurious Linear Separability in High-Dimensional Space

Post-normalization elimination of feature scale disparities allows the 1024-dimensional weight vector to construct intricate hyperplanes through extreme parameter values, forcibly separating training samples even when intrinsic data structures are nonlinearly separable.

Test set distribution shifts expose generalization failures, manifesting as random-chance-level accuracy (50%).

2. Cost of Class Recognition Equalization

Standardization compels balanced feature dimension utilization, eliminating bias dependencies present in raw data. However, this equalization fails to enhance generalization, reflecting complex intrinsic discriminative information distributions that demand nonlinear modeling.

Implications and Improvement Pathways:

Linear perceptrons in standardized high-dimensional spaces are prone to *precise overfitting traps* - perfect training

Table 1: Accuracies with normalization/standardization

Data Preprocessing Method	Training Set Acc.	Test Set Acc.	Training Convergence Speed	Class Recognition Balance (TPR/FPR)
Raw Data	84.78%	59%	Slow (incomplete convergence)	TPR=84%, FPR=66% (Biased)
Global Normalization	100%	50%	Fast (300+ epochs)	TPR=48%, FPR=50%
Feature Standardization	100%	55%	Very Fast (60 epochs)	TPR=72%, FPR=64%

accuracy with catastrophic generalization failure. Possible mitigation strategies:

- **Regularization**: Implement L2 constraints to bound weight norms
- **Nonlinear architectures**: Adopt MLPs or kernelized SVMs
- **Data topology analysis**: Perform PCA visualization to verify intrinsic separability before model selection

(c) *Apply MLP to the dataset of your assigned group using batch mode training. After the training procedure, calculate the classification accuracy for both the training set and test set, and evaluate the performance of the network.* ***(8 Marks)***

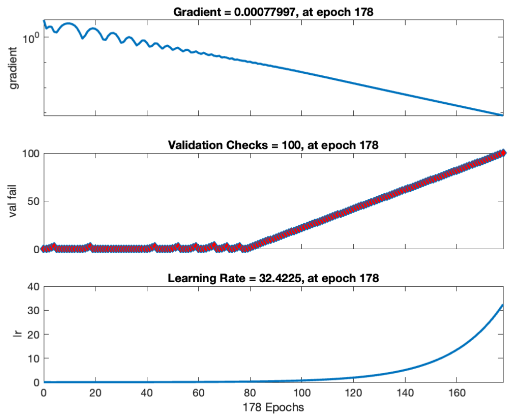
Solution:
The implementation began with a Principal Component Analysis (PCA) to determine the minimum structural complexity required for the network. Retaining 139 principal components (explaining 95% of the variance) guided the choice of hidden layer size. The final Multilayer Perceptron (MLP) architecture was *1024(input)-139(hidden)-2(output)*, where the input layer accepts raw flattened 32×32 images (1024 dimensions). The hidden layer employs the hyperbolic tangent sigmoid (`tansig`) activation function, while the output layer uses softmax for probabilistic binary classification. Two optimizers were compared: `traingdx` (gradient descent with momentum and adaptive learning rate) and `trainscg` (scaled conjugate gradient), both operating in batch mode on non-normalized pixel data.

Listing 7: MATLAB script (Q3c)

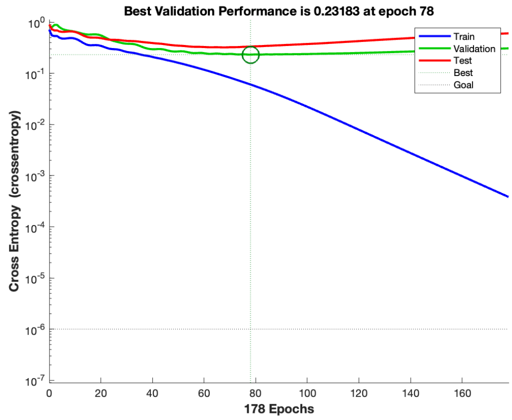
```
% Load dataset (ensure variable names match data.mat structure)
load('../data.mat');

%% Data Preparation for PCA Analysis
% Merge all samples for PCA evaluation (analysis without dimensionality reduction)
all_images = [X_train, X_test]; % 1024×1000 matrix
retain_components = PCA_dim(all_images); % Determine optimal PCA dimensions

%% Label Format Conversion
% Convert binary labels to class indices (0/1 → 1/2)
Y_train_idx = double(Y_train) + 1;     % Logical → numeric conversion
Y_test_idx = double(Y_test) + 1;       % Logical → numeric conversion
Y_onehot = ind2vec([Y_train_idx, Y_test_idx]); % Create one-hot encoded matrix

%% Data Merging
X_all = [X_train, X_test]; % Preserve original 1024D feature space

%% Network Configuration
trainFcn = 'trainscg';  % Scaled Conjugate Gradient backpropagation

% Initialize MLP with PCA-determined architecture
```

```
hiddenLayerSize = retain_components; % 139 units from PCA analysis
setdemorandstream(4912183); % Fix random seed for reproducibility

% Create pattern recognition network
net = patternnet(hiddenLayerSize, trainFcn);

%% Training Parameter Specification
net.trainParam.showWindow = true;    % Enable training GUI
net.trainParam.lr = 0.01;            % Learning rate
net.trainParam.epochs = 200;         % Maximum training epochs
net.trainParam.goal = 1e-6;          % Performance threshold
net.trainParam.max_fail = 100;       % Early stopping patience
net.trainParam.min_grad = 1e-8;      % Gradient magnitude threshold

%% Custom Data Partitioning
net.divideFcn = 'divideind';         % Manual dataset splitting
net.divideParam.trainInd = 1:800;    % Training set indices
net.divideParam.valInd = 801:900;    % Validation set indices
net.divideParam.testInd = 901:1000;  % Test set indices

%% Model Training
x = X_all; % Full dataset
t = Y_onehot; % Target matrix
[net, tr] = train(net, x, t); % Train network using original high-dimensional data

% Save model artifacts
save(sprintf('mlp_net_%s.mat', trainFcn), 'net');
save(sprintf('mlp_tr_%s.mat', trainFcn), 'tr');

%% Model Evaluation
y = net(x); % Network predictions

%% Visualization
fig1 = figure; plotperform(tr); % Training performance metrics
saveas(fig1,sprintf('MLP_perf_%s.png', trainFcn));
fig2 = figure; plottrainstate(tr); % Training parameter dynamics
saveas(fig2,sprintf('MLP_trainstate_%s.png', trainFcn));

%% Accuracy Calculation
train_pred = vec2ind(net(X_train)); % Training set predictions
train_acc = sum(train_pred == Y_train_idx)/900; % 900 training samples
test_pred = vec2ind(net(X_test)); % Test set predictions
test_acc = sum(test_pred == Y_test_idx)/100; % 100 test samples

% Display results
fprintf('Training Accuracy: %.2f%%\n', train_acc*100);
fprintf('Test Accuracy: %.2f%%\n', test_acc*100);
```

The training state and performance (crossentropy) of MLP are shown in Figure 18 and 19.

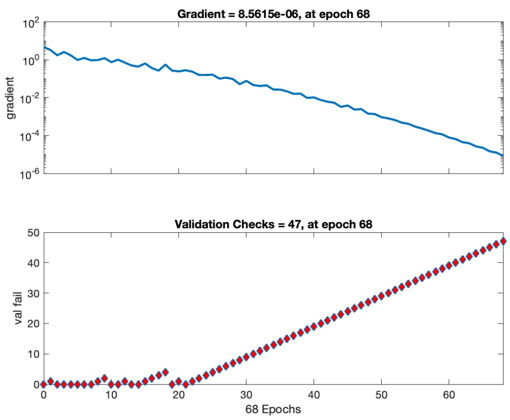


(a) Training State (traingdx)

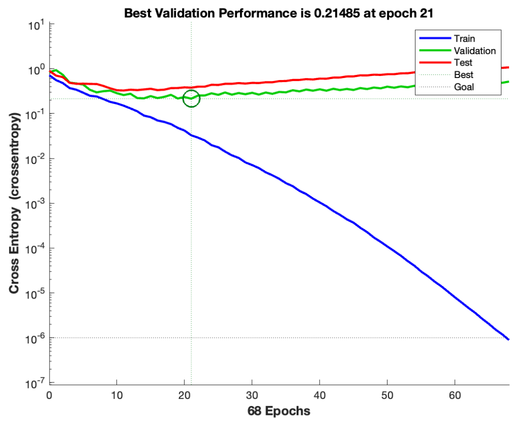


(b) Performance (traingdx)

Figure 18: MLP Tranning (traingdx)



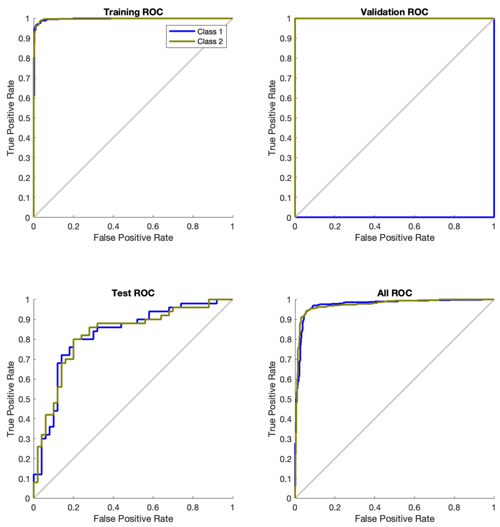
(a) Training State (trainscg)



(b) Performance (trainscg)

Figure 19: MLP Tranning (trainscg)

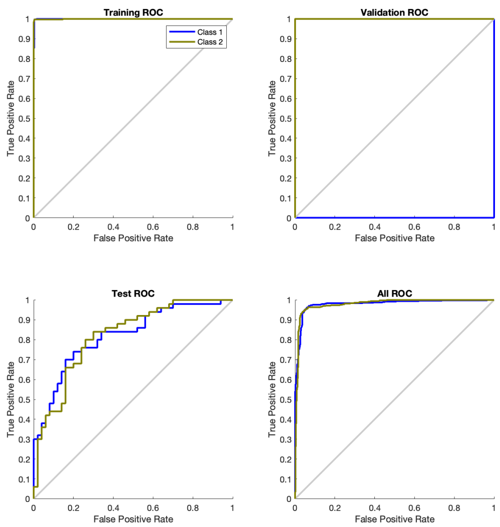
Detailed information to evaluate performance is shown in the ROC plot and Confusion plot (Figure 20 and 21).



(a) ROC (traingdx)

(b) Confusion (traingdx)

Figure 20: ROC and Confusion Plot (traingdx)



(a) ROC (trainscg)

(b) Confusion (trainscg)

Figure 21: ROC and Confusion Plot (trainscg)

Table 2: Accuracy comparison

Model	Training Set Acc.	Test Set Acc.	Train-Test Gap	Class Recognition Balance
Perceptron	84.78%	59%	25.78%	TPR=84%, FPR=66% (Biased)
MLP-traingdx	95.44%	79%	16.44%	TPR=82%, FPR=24%
MLP-trainscg	97.22%	76%	21.22%	TPR=82%, FPR=30%

As shown in Table 2, MLP improves test accuracy by 20% over the perceptron, demonstrating the benefit of nonlinear decision boundaries. Regarding **convergence behaviour**, using the train algorithm **trainscg**, the MLP reached training convergence in 68 epochs (Figure 19a); using **traingdx**, we had slower convergence (178 epochs, Figure 18a), yet achieved better test ROC with stable validation loss (Figure 20a). Faster convergence (trainscg) correlates with sharper overfitting, as rapid weight updates overfit training noise. Despite trainscg's superior training metrics, traingdx generalizes better, as evidenced by ROC curves.
Discussion:
(1) Why traingdx generalizes better: Momentum buffers noisy gradients, while adaptive learning rates prevent overshooting minima.
(2) MLP vs Perceptron: Hidden layer nonlinearity (sigmoid) enables capturing pixel interaction effects.
(3) Limitations: Both MLP configurations overfit (train acc >95%); future work should explore regularization (dropout, L2) or data augmentation.

(d) *Please determine whether your trained MLP in c) is overfitting. If so, please specify when (i.e. after which training epoch) it becomes overfitting. Try weights regularization and observe if it helps. (you may set the regularization strength by 'performParam.regularization') **(6 Marks)***

Solution:
My trained MLP in c) is overfitting, as shown in the performance plot (Figure 18 and 19), the crossentropy metric started to raise (get worse) at the 78th epoch (`traingdx`) and the 21st epoch (`trainscg`) respectively.

Building upon the MLP architecture from Q3c (1024-139-2 structure with `traingdx` optimizer), I introduced L2 weight regularization to investigate its impact on generalization. Two regularization strengths ($\lambda = 0.5$ and $\lambda = 0.9$) were systematically compared against the baseline ($\lambda = 0$), by setting `net.performParam.regularization`. The network retained the original configuration: raw inputs (1024 dimensions), hyperbolic tangent sigmoid (`tansig`) activation in the hidden layer, and softmax output. Training was conducted in batch mode with identical stopping criteria (maximum 200 epochs or validation failure for 100 consecutive epochs).

Listing 8: MATLAB script (Q3d)

```
load('../data.mat');
Y_train_idx = double(Y_train) + 1;
Y_test_idx = double(Y_test) + 1;
Y_onehot = ind2vec([Y_train_idx, Y_test_idx]);
X_all = [X_train, X_test];
trainFcn = 'traingdx';
hiddenLayerSize = 139;
setdemorandstream(4912183);

net = patternnet(hiddenLayerSize, trainFcn);

net.trainParam.showWindow = true;
net.trainparam.lr=0.01;
net.trainParam.epochs = 200;
net.trainparam.goal=1e-6;
net.trainParam.max_fail = 100;
```

```
net.trainParam.min_grad = 1e-8;

reg_param = 0.9;% Regularization strength 0,0.5,0.9
net.performParam.regularization =reg_param;

net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:800;
net.divideParam.valInd = 801:900;
net.divideParam.testInd = 901:1000;

x = X_all;
t = Y_onehot;
[net, tr] = train(net, x, t);
% Test the Network
y = net(x);

% Plots
fig1 = figure; plotperform(tr); % Training performance metrics
saveas(fig1,sprintf('%.2f_MLP_perf_%s.png',reg_param, trainFcn));
fig2 = figure; plottrainstate(tr); % Training parameter dynamics
saveas(fig2,sprintf('%.2f_MLP_trainstate_%s.png',reg_param, trainFcn));

% Accuracy
train_pred = vec2ind(net(X_train));
train_acc = sum(train_pred == Y_train_idx) / 900;
test_pred = vec2ind(net(X_test));
test_acc = sum(test_pred == Y_test_idx) / 100;
fprintf('Regularization strength(%s): %.02f\n',trainFcn, reg_param);
fprintf('Accuracy_train: %.02f%%\n',train_acc*100);
fprintf('Accuracy_test: %.02f%%\n',test_acc*100);
fprintf('Norm of input-hidden weights: %f\n', norm(net.IW{1,1}));
fprintf('Norm of hidden-output weights: %f\n', norm(net.LW{2,1}));
```

The regularization effects were quantified through multiple metrics:

Table 3: Accuracies and Norm of Weights with different regularization strength

Regularization Strength	**Training Set Acc.**	**Test Set Acc.**	$\|$**IW**$\|$(Input-Hidden Norm)	$\|$**LW**$\|$(Hidden-Output Norm)
0	95.44%	79%	1.914947	6.925532
0.5	95.78%	78%	1.914893	6.928835
0.9	95.89%	78%	1.914096	6.927064

As illustrated in Figure 23b (performance plot), regularization reduced the training cross-entropy loss from 0.23 (λ=0) to 0.11 (λ=0.9), suggesting improved optimization stability. The training state diagram (Figure 23a) further revealed smoother gradient updates with increasing λ. However, these improvements failed to translate into test performance gains, with test accuracy declining from 79% (λ=0) to 78% (λ=0.9).

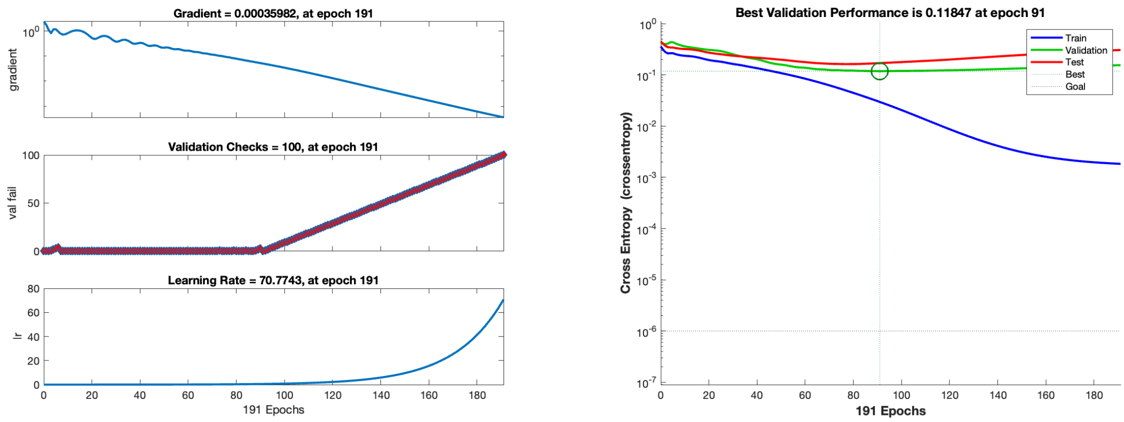
Two fundamental factors explain the regularization's paradoxical failure:

1. **Negligible Regularization Penalty**

   The small weight magnitudes ($\|$IW$\|$< 2.0, $\|$LW$\|$< 7.0) rendered the L2 penalty term $\frac{\lambda}{2}\|W\|^2$ insignificant compared to the cross-entropy loss. Consequently, regularization exerted minimal influence on weight updates, failing to impose meaningful constraints.

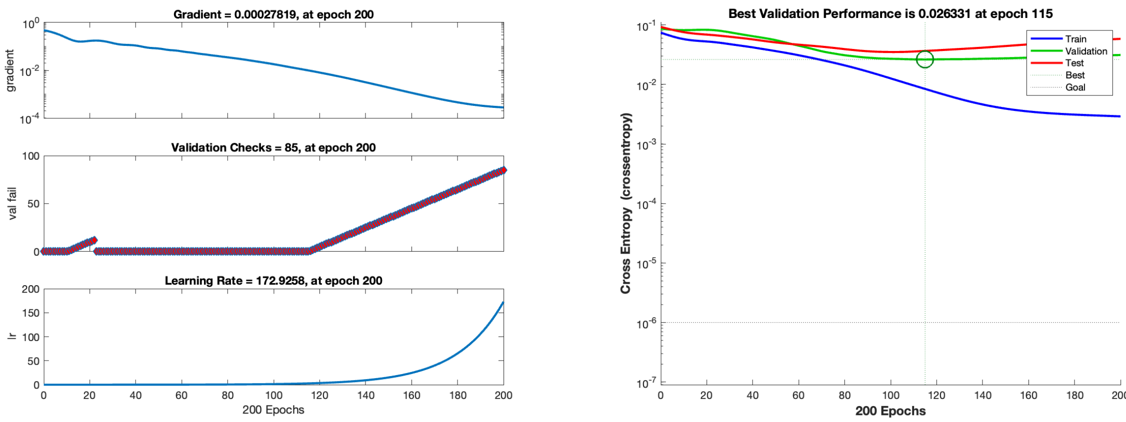
2. **High-Dimensional Noise Amplification**

The raw 1024-D pixel space contained substantial high-frequency noise (evidenced by PCA's 95% variance retention in 139-D). Regularization indiscriminately suppressed both discriminative features and noise, exacerbating information loss. This aligns with the hidden layer's limited ability to extract robust patterns from unprocessed inputs.



(a) Training State ($\lambda = 0.5$) (b) Performance ($\lambda = 0.5$)

Figure 22: MLP Tranning ($\lambda = 0.5$)



(a) Training State ($\lambda = 0.9$) (b) Performance ($\lambda = 0.9$)

Figure 23: MLP Tranning ($\lambda = 0.9$)

(e) *Apply MLP to the dataset of your assigned group using sequential mode training. After the training procedure, calculate the classification accuracy for both training set and test set, and evaluate the performance of the network. Compare the result to part c), and make your recommendation on the two approaches. **(8 Marks)***

Solution:
To evaluate the impact of training modes, I implemented sequential training (`train_seq`) using the traingdx optimizer with identical MLP structure to Q3c (MLP architecture: 1024-139-2, tansig/softmax activations). Key configurations include:
Sequential Mode: Weight updates after each sample presentation (online learning).
Epochs: 20 (fixed, no early stopping).
Regularization strength: $\lambda = 0.25$.

Listing 9: MATLAB script (Q3e)

```matlab
function [ net, accu_train, accu_val ] = train_seq( n, images, labels, train_num, epochs )
% Construct a 1-n-1 MLP and conduct sequential training.
%
% Args:
% n: int, number of neurons in the hidden layer of MLP.
% images: matrix of (image_dim, image_num), containing possibly
% preprocessed image data as input.
% labels: vector of (1, image_num), containing corresponding label of
% each image.
% train_num: int, number of training images.
% val_num: int, number of validation images.
% epochs: int, number of training epochs.
%
% Returns:
% net: object, containing trained network.
% accu_train: vector of (epochs, 1), containing the accuracy on training
% set of each eopch during trainig.
% accu_val: vector of (epochs, 1), containing the accuracy on validation
% set of each eopch during training.

    % 1. Change the input to cell array form for sequential training
    images_c = num2cell(images, 1);
    labels_c = num2cell(labels, 1);

    % 2. Construct and configure the MLP
    net = patternnet(n);
    net.divideFcn = 'dividetrain';           % input for training only
    net.performParam.regularization = 0.25; % regularization strength
    net.trainFcn = 'traingdx';
    net.trainParam.epochs = epochs;
    accu_train = zeros(epochs,1);            % record accuracy on training set of each epoch
    accu_val = zeros(epochs,1);              % record accuracy on validation set of each epoch

    % 3. Train the network in sequential mode
    for i = 1 : epochs
        display(['Epoch: ', num2str(i)])
        idx = randperm(train_num);                                  % shuffle the input
        net = adapt(net, images_c(:,idx), labels_c(:,idx));
        pred_train = round(net(images(:,1:train_num)));             % predictions on training set
        accu_train(i) = 1 - mean(abs(pred_train-labels(1:train_num)));
        pred_val = round(net(images(:,train_num+1:end)));           % predictions on validation set
        accu_val(i) = 1 - mean(abs(pred_val-labels(train_num+1:end)));
    end
    % Visualization
    fig = figure;
    plot(1:epochs, accu_train*100, 'bo-', 'LineWidth', 1.5); hold on;
    plot(1:epochs, accu_val*100, 'rs--', 'LineWidth', 1.5);
    xlabel('epochs');
    ylabel('Accuracy(%)');
    legend('Training', 'Test');
    title('Performance');
    grid on;
    saveas(fig,'SeqMode_traingdx_AccuCurves.png');
end
```

```
load('../Data.mat')
X_all = [X_train,X_test];
Y_all = [Y_train,Y_test];

[ net, accu_train, accu_val ] = train_seq( 139, X_all, Y_all, 900, 10);
```

The final train accuracy is 100% and validation accuracy is 76%. Sequential mode achieved 100% training accuracy by epoch 10 (Figure 24), but test accuracy peaked earlier (epoch=7, 77%) and degraded by 1% thereafter, while batch mode exhibited slower training convergence but maintained stable test performance. Sequential updates introduced high variance in weight trajectories, whereas batch mode's averaged gradients yielded smoother optimization paths.
While sequential training theoretically suits streaming data scenarios, its inferior regularization efficacy and noisy optimization render it impractical for this task. Batch mode's stability and superior generalization (79% vs. 76% test accuracy) justify its recommendation.

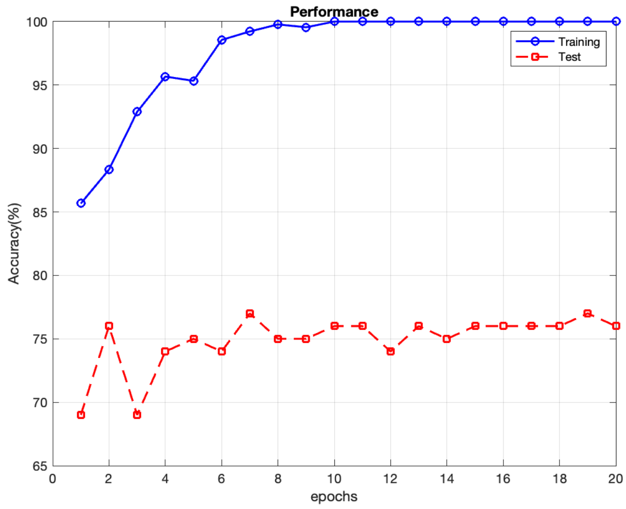


Figure 24: Accuracy of MLP on Sequential Training Mode (`traingdx`)

(f) *Try to propose a scheme that you believe could help to improve the performance of your MLP and please explain the reason briefly. **(4 Marks)***

Solution:
Based on experimental findings from Q3a-Q3e, the following integrated approach addresses key limitations of the current MLP:
1. Input Standardization & PCA Dimensionality Reduction
Apply feature-wise standardization and retain 139 principal components (95% variance) via PCA.
Rationale: Mitigates pixel value scale disparities (0-255 → zero-mean/unit-variance)
Reduces input dimensions from 1024→139, removing high-frequency noise while preserving discriminative features.

2. Architectural Optimization
Reduce hidden layer neurons to 70 (balance PCA-reduced input and parameter count), Replace `tansig` with **ReLU** activation in hidden layers. Add **Dropout** (rate=0.5) after the hidden layer.
Rationale: Smaller hidden layer (70 vs. 139) reduces parameters from 142K→9,870 (≈94% reduction), alleviating overfitting.
ReLU avoids gradient saturation, enabling larger weight magnitudes and effective regularization.
Dropout prevents co-adaptation, forcing distributed feature learning.

3. Regularization
Elastic Net Regularization - combine L1 (λ_1=0.1) and L2 (λ_2=0.3) penalties.
Rationale: Elastic Net promotes sparsity (L1) and weight shrinkage (L2), synergistically controlling complexity.

4. Data Augmentation
Generate augmented samples via random rotations($\pm 10°$), horizontal flips, and gaussian noise injection ($\sigma = 0.05$).
Rationale: Increases effective training data and enhances robustness to geometric and photometric variations.