Problem Set 1: Neuro-Fuzzy Computing

Nikos Mavros
University of Thessaly
Department of Electrical and Computer Engineering

February 17, 2026

Problem 1

We consider the function:

$$f(x, y) = x^4 + y^4 - 4xy + 1.$$

Stationary Points The first partial derivatives are:

$$f_x = 4x^3 - 4y, \quad f_y = 4y^3 - 4x.$$

Setting them equal to zero gives $y = x^3$ and $x = y^3$. Substituting yields $x = x^9 \Rightarrow x(x^8 - 1) = 0 \Rightarrow x = 0, \pm 1$. Hence, the real critical points are:

$$(0, 0), \quad (1, 1), \quad (-1, -1).$$

Second Derivative

$$f_{xx} = 12x^2, \quad f_{yy} = 12y^2, \quad f_{xy} = -4.$$

Hessian:

$$H = \begin{pmatrix} 12x^2 & -4 \\ -4 & 12y^2 \end{pmatrix}, \quad \det(H) = 144x^2y^2 - 16.$$

- At $(0, 0)$: $\det(H) = -16 < 0 \Rightarrow$ **saddle point.**
- At $(1, 1)$ and $(-1, -1)$: $\det(H) = 128 > 0$, $f_{xx} = 12 > 0 \Rightarrow$ **local (and global) minima.**

Function values:

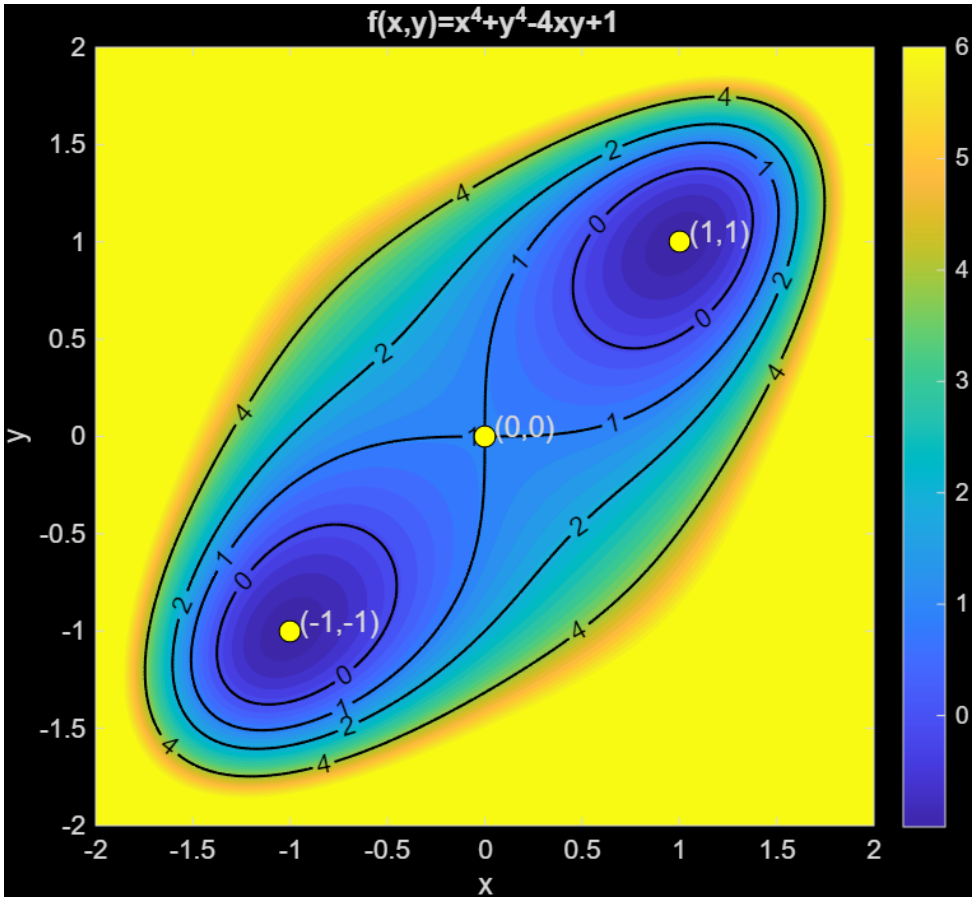
$$f(1, 1) = f(-1, -1) = -1.$$

Conclusion The function has global minima at $(1, 1)$ and $(-1, -1)$ with $f_{\min} = -1$, and a saddle point at $(0, 0)$.

MATLAB Implementation

Listing 1: Contour plot of $f(x, y) = x^4 + y^4 - 4xy + 1$

```
clear; close all; clc

x = linspace(-2,2,400);
y = linspace(-2,2,400);
[X,Y] = meshgrid(x,y);
F = X.^4 + Y.^4 - 4.*X.*Y + 1;

levels = linspace(-2,6,36);
figure('Units','normalized','Position',[0.1 0.1 0.6 0.6])
contourf(X,Y,F,levels,'LineColor','none')
hold on
[CC,hc] = contour(X,Y,F,[-1 0 1 2 4],'LineColor','k','LineWidth',1);
clabel(CC,hc,'FontSize',10,'Color','k')
contour(X,Y,F,[-1 -1],'LineColor','r','LineWidth',2)

crit = [0 0; 1 1; -1 -1];
plot(crit(:,1),crit(:,2),'ko','MarkerFaceColor','y','MarkerSize',8)
text(crit(1,1)+0.05,crit(1,2)+0.05,'(0,0)','FontSize',11)
text(crit(2,1)+0.05,crit(2,2)+0.05,'(1,1)','FontSize',11)
text(crit(3,1)+0.05,crit(3,2)+0.05,'(-1,-1)','FontSize',11)

axis equal
xlabel('x'); ylabel('y')
title('f(x,y)=x^4+y^4-4xy+1')
colorbar
```



Problem 2

We perform two iterations of Gradient Descent starting from the initial point $x_0 = (3, 2)$.

Gradient Calculation The gradient of the function is

$$\nabla f(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 12x - 4y \\ -4x + 8y \end{pmatrix}.$$

Iteration 1: $x_0 = (3, 2)$

$$g_0 = \nabla f(3, 2) = \begin{pmatrix} 28 \\ 4 \end{pmatrix}, \quad \alpha_0 = \frac{g_0^T g_0}{2 g_0^T A g_0} = \frac{5}{54} \approx 0.0926$$

$$x_1 = x_0 - \alpha_0 g_0 = \begin{pmatrix} \frac{11}{27} \\ \frac{44}{27} \end{pmatrix} \approx \begin{pmatrix} 0.4074 \\ 1.6296 \end{pmatrix}, \quad f(x_1) \approx 8.963$$

Iteration 2: $x_1 = (11/27, 44/27)$

$$g_1 = \nabla f\left(\frac{11}{27}, \frac{44}{27}\right) = \begin{pmatrix} -\frac{44}{27} \\ \frac{308}{27} \end{pmatrix}, \quad \alpha_1 = \frac{5}{46} \approx 0.1087$$

$$x_2 = x_1 - \alpha_1 g_1 \approx \begin{pmatrix} 0.5845 \\ 0.3897 \end{pmatrix}, \quad f(x_2) \approx 1.746$$

Summary of Iterations.

Iteration	x	$f(\mathbf{x})$
0	(3.0000, 2.0000)	46.000
1	(0.4074, 1.6296)	8.963
2	(0.5845, 0.3897)	1.746

Problem 3

1. Logistic Sigmoid Function The logistic sigmoid is defined as:

$$S(x) = \frac{1}{1+e^{-x}}$$

To compute its derivative, we apply the chain rule:

$$S'(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2}$$

From the definition of S, we can isolate e^{-x}:

$$S = \frac{1}{1+e^{-x}} \quad \Rightarrow \quad 1+e^{-x} = \frac{1}{S} \quad \Rightarrow \quad e^{-x} = \frac{1-S}{S}$$

Substituting this into the expression for $S'(x)$:

$$S'(x) = \frac{\frac{1-S}{S}}{\left(\frac{1}{S}\right)^2} = S(1-S)$$

$$\boxed{S' = S(1-S)}$$

2. Hyperbolic Tangent Function

$$S(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Differentiating directly gives:

$$S'(x) = \operatorname{sech}^2(x)$$

Using the identity $\operatorname{sech}^2(x) = 1 - \tanh^2(x)$, we can rewrite the derivative in terms of S:

$$S'(x) = 1 - S^2$$

$$\boxed{S' = 1 - S^2}$$

3. Swish (SiLU) Function

$$S(x) = \frac{x}{1+e^{-x}}$$

Using the quotient rule:

$$S'(x) = \frac{(1+e^{-x}) - x(-e^{-x})}{(1+e^{-x})^2} = \frac{1+e^{-x}+xe^{-x}}{(1+e^{-x})^2}$$

To express $S'(x)$ in terms of S and x, we first rewrite the exponential term from the definition:

$$S = \frac{x}{1+e^{-x}} \quad \Rightarrow \quad 1+e^{-x} = \frac{x}{S} \quad \text{and} \quad e^{-x} = \frac{x-S}{S}$$

Substituting these into the derivative:

$$S'(x) = \frac{1 + \frac{x-S}{S} + x\frac{x-S}{S}}{\left(\frac{x}{S}\right)^2}$$

$$= \frac{S(1 + x - S)}{x}$$

$$\boxed{S' = \frac{S(1 + x - S)}{x}}$$

When $x \to 0$, the derivative approaches $\frac{1}{2}$, ensuring continuity.

4. Composite Function

$$S(x) = x \tanh(x)$$

Applying the product rule:

$$S'(x) = \tanh(x) + x \operatorname{sech}^2(x)$$

Since $S = x \tanh(x)$, we can express the hyperbolic functions in terms of S and x:

$$\tanh(x) = \frac{S}{x}, \qquad \operatorname{sech}^2(x) = 1 - \tanh^2(x) = 1 - \frac{S^2}{x^2}$$

Substituting into the derivative:

$$S'(x) = \frac{S}{x} + x\left(1 - \frac{S^2}{x^2}\right)$$

$$= x + \frac{S - S^2}{x}$$

$$\boxed{S' = x + \frac{S(1 - S)}{x}}, \qquad x \neq 0$$

At $x = 0$, $S(0) = 0$ and $S'(0) = 0$, confirming continuity.

Problem 4

Case 1: Log-Sigmoid Activation

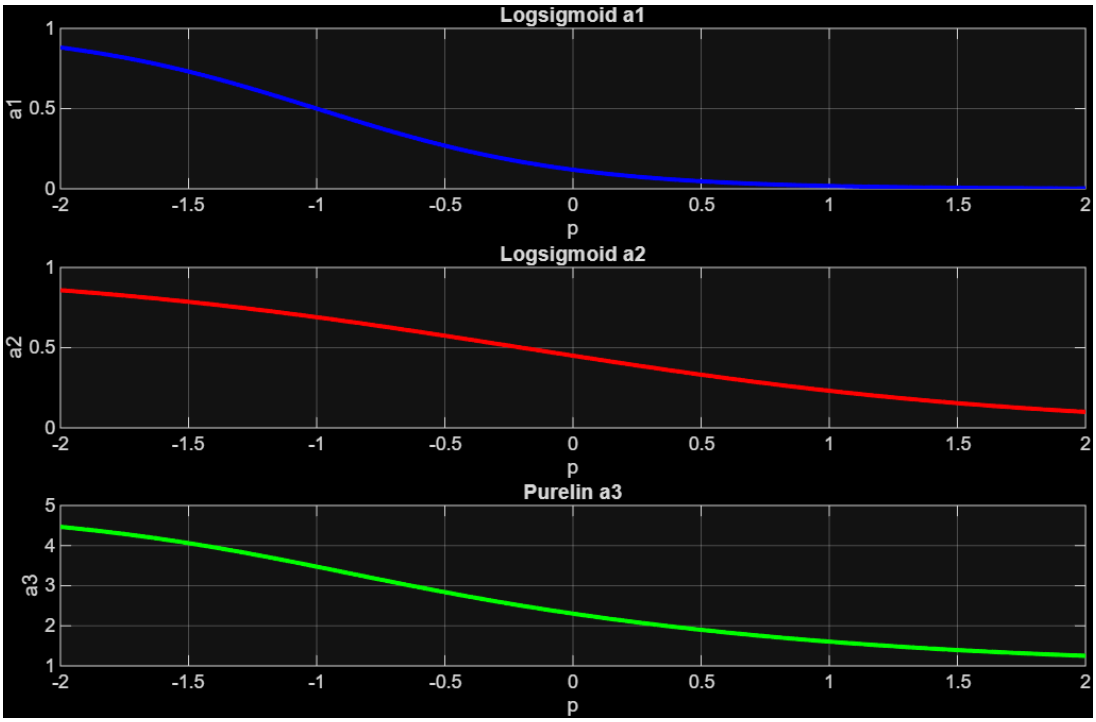
The `logsigmoid` activation function is defined as:

$$a = \frac{1}{1 + e^{-n}}$$

where n is the weighted input of the neuron.

Listing 2: Neural network using logsigmoid writen in MATLAB

```
%% Logsigmoid Activation Implementation
p = linspace(-2,2,100);
w1 = -2; w2 = -1;
b1 = -2; b2 = -0.2;
w22 = 2.5; w11 = 1.5; b3 = 1;

% Hidden layer
n1 = w1*p + b1;
n2 = w2*p + b2;

% Log-sigmoid activation
a1 = 1 ./ (1 + exp(-n1));
a2 = 1 ./ (1 + exp(-n2));

% Output layer (purelin)
n3 = a1*w11 + a2*w22 + b3;
a3 = n3;

% Plot results
figure;
subplot(3,1,1); plot(p,a1,'b','LineWidth',2);
title('Logsigmoid a1'); grid on;
subplot(3,1,2); plot(p,a2,'r','LineWidth',2);
title('Logsigmoid a2'); grid on;
subplot(3,1,3); plot(p,a3,'k','LineWidth',2);
title('Purelin a3'); grid on;
```



Activation outputs for the network using the `logsigmoid` function.

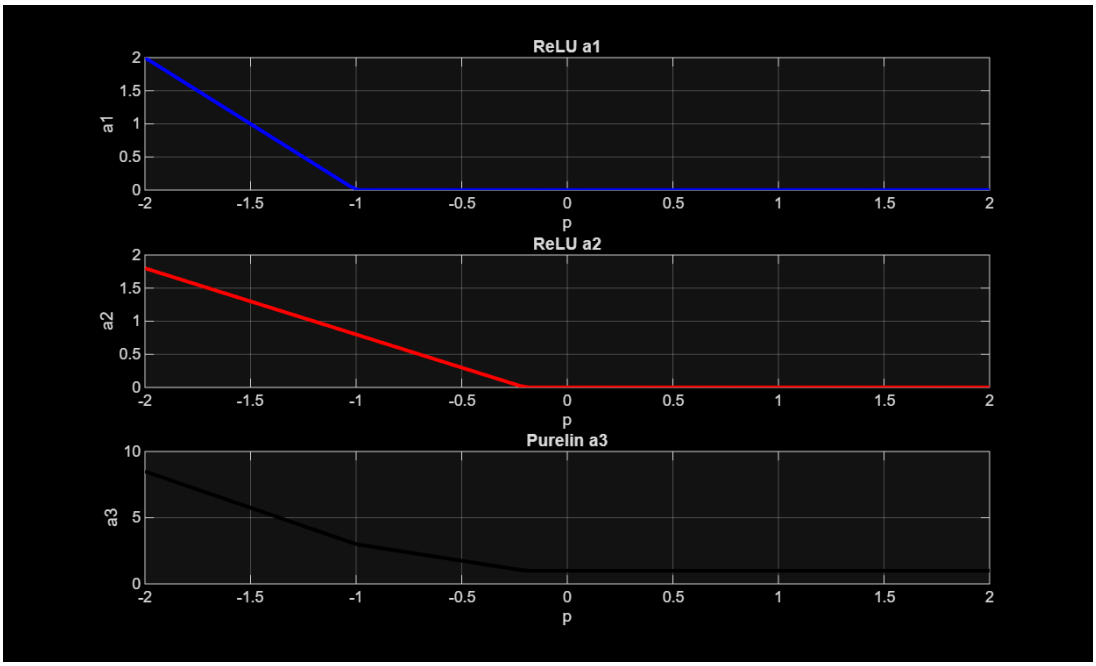
Case 2: ReLU Activation

The ReLU (Rectified Linear Unit) activation function is defined as:

$$a = \max(0, n)$$

Listing 3: Neural network using ReLU activation

```
%% ReLU Activation Implementation
p = linspace(-2,2,100);
w1 = -2; w2 = -1;
b1 = -2; b2 = -0.2;
w22 = 2.5; w11 = 1.5; b3 = 1;

% Hidden layer
n1 = w1*p + b1;
n2 = w2*p + b2;

% ReLU activation
a1 = max(0, n1);
a2 = max(0, n2);

% Output layer (purelin)
n3 = a1*w11 + a2*w22 + b3;
a3 = n3;

% Plot results
figure;
subplot(3,1,1); plot(p,a1,'b','LineWidth',2);
title('ReLU a1'); grid on;
subplot(3,1,2); plot(p,a2,'r','LineWidth',2);
title('ReLU a2'); grid on;
subplot(3,1,3); plot(p,a3,'k','LineWidth',2);
title('Purelin a3'); grid on;
```



Activation outputs for the network using the `ReLU` function.

Comparing plots 1 and 2, we observe that the `logsigmoid` activations are smooth and bounded between 0 and 1, while the `ReLU` activations are piecewise linear and unbounded for positive inputs, producing sharper transitions in the network output.

Problem 5

Network Equations

The hidden layer computes:

$$n_1^1 = w_{1,1}^1 p + b_1^1$$
$$n_2^1 = w_{2,1}^1 p + b_2^1$$

The activation function of the hidden layer is the **Swish** function:

$$\text{Swish}(x) = x \cdot \sigma(x), \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

So the hidden outputs are:

$$a_1^1 = n_1^1 \cdot \sigma(n_1^1)$$
$$a_2^1 = n_2^1 \cdot \sigma(n_2^1)$$

The output neuron computes:

$$n^2 = w_{1,1}^2 a_1^1 + w_{1,2}^2 a_2^1 + b^2$$

with Leaky ReLU activation:

$$a^2 = \max(0, n^2) + 0.1 \cdot \min(0, n^2)$$

MATLAB Implementation

```
clc;clear;close all;

w11_1 = -0.27;
w21_1 = -0.41;
b1_1  = -0.48;
b2_1  = -0.13;

w11_2 = 0.09;
w12_2 = -0.17;
b2    = 0.48;

alpha = 0.10;

p = linspace(-2, 2, 400);
n1_1 = w11_1 .* p + b1_1;
n2_1 = w21_1 .* p + b2_1;

% Swish activation function
sigmoid = @(x) 1 ./ (1 + exp(-x));
a1_1 = n1_1 .* sigmoid(n1_1);
a2_1 = n2_1 .* sigmoid(n2_1);

% Layer 2 net input
n2 = w11_2 .* a1_1 + w12_2 .* a2_1 + b2;

% Leaky ReLU activation
a2 = max(0, n2) + alpha .* min(0, n2);

figure;
subplot(3,2,1);
plot(p, n1_1, 'LineWidth', 1.5);
title('n_1^1 vs p');
grid on;
subplot(3,2,2);
plot(p, a1_1, 'LineWidth', 1.5);
title('a_1^1 vs p');
grid on;
subplot(3,2,3);
plot(p, n2_1, 'LineWidth', 1.5);
title('n_2^1 vs p');
grid on;
subplot(3,2,4);
plot(p, a2_1, 'LineWidth', 1.5);
title('a_2^1 vs p');
grid on;
subplot(3,2,5);
plot(p, n2, 'LineWidth', 1.5);
title('n^2 vs p');
grid on;
subplot(3,2,6);
plot(p, a2, 'LineWidth', 1.5);
title('a^2 vs p');
grid on;

sgtitle('Neural Network Responses for -2 < p < 2');
```

Generated Plot

The figure below shows all six responses of the neural network for $-2 < p < 2$, generated from the MATLAB simulation.

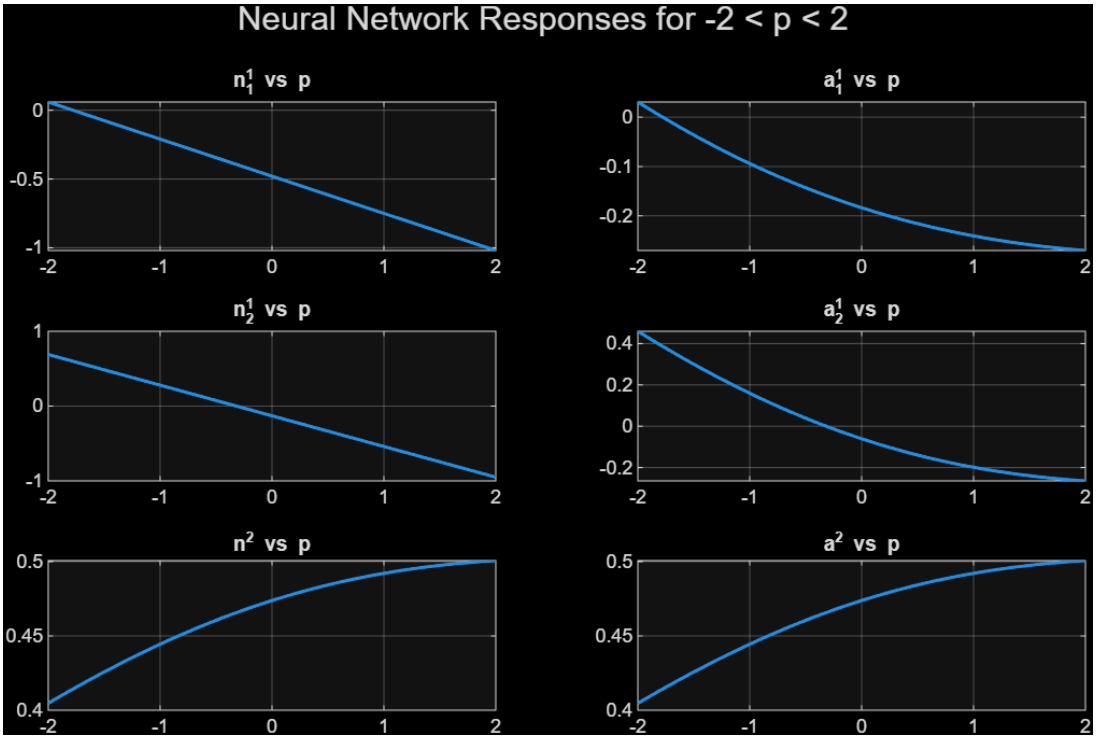


Figure 1: Neural network internal responses and output versus input p.

The plot image was uploaded directly to the Overleaf project file list under the name:

`problem5_nn_plot.png`

Problem 6

A. Target values

We select scalar targets that encode the three classes:

$$t_k = \begin{cases} +1, & p_k \in \text{Class I}, \\ 0, & p_k \in \text{Class II}, \\ -1, & p_k \in \text{Class III}. \end{cases}$$

Explicit target vector:

$$t = [1,\ 1,\ 0,\ 0,\ -1,\ -1,\ -1].$$

B. ADALINE network diagram

A single linear neuron with no bias:

$$y = w_1 x_1 + w_2 x_2.$$

LMS weight update:

$$w(n+1) = w(n) + \eta\, e(n)\, x(n), \qquad e(n) = t(n) - w^\top(n) x(n).$$

C. Mean–square error performance index

The weighted mean-square error is

$$J(w) = \sum_{k=1}^{7} p_k \left(t_k - w^\top x_k\right)^2.$$

Probability-weighted covariance:

$$R_{xx} = \sum_{k=1}^{7} p_k x_k x_k^\top = \begin{bmatrix} 0.825 & -0.125 \\ -0.125 & 1.200 \end{bmatrix}.$$

Cross-correlation vector (corrected):

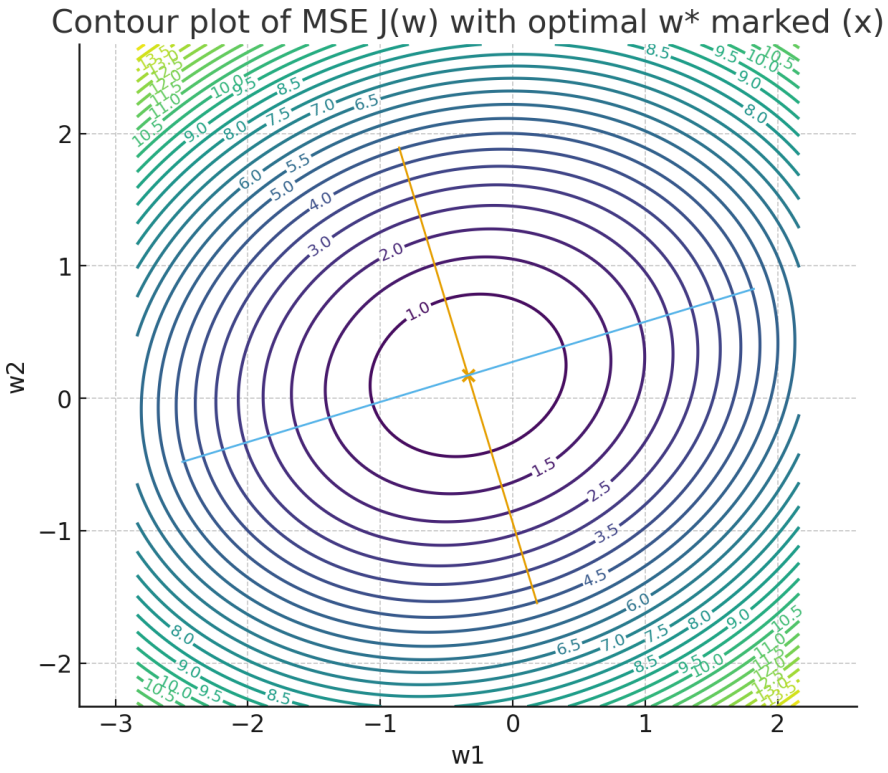
$$r_{xt} = \sum_{k=1}^{7} p_k x_k t_k = \begin{bmatrix} -0.05 \\ 0.75 \end{bmatrix}.$$

Thus,

$$J(w) = w^\top R_{xx} w - 2\, r_{xt}^\top w + J_{\min},$$

a quadratic bowl whose minimum occurs at

$$w_* = R_{xx}^{-1} r_{xt} = \begin{bmatrix} 0.034638 \\ 0.628608 \end{bmatrix}.$$



MSE contour plot centered at the optimal weight vector w_*.

D. Optimal decision boundary

With no bias, the decision boundary is

$$w_*^\top x = 0 \quad \Rightarrow \quad x_2 = -\frac{w_1}{w_2}\, x_1 = -0.05513\, x_1.$$

Evaluating $y_k = w_*^\top p_k$ confirms that Class III patterns lie below the line, Class I/-Class II patterns lie above or near it. Exact class separation is not achievable because p_1 is at the origin, but the least-squares solution provides the best linear fit.

E. Maximum stable learning rate

The LMS stability condition is

$$0 < \eta < \frac{2}{\lambda_{\max}(R_{xx})}.$$

Eigenvalues of R_{xx}:

$$\lambda_1 \approx 1.23785, \qquad \lambda_2 \approx 0.78715.$$

Therefore,

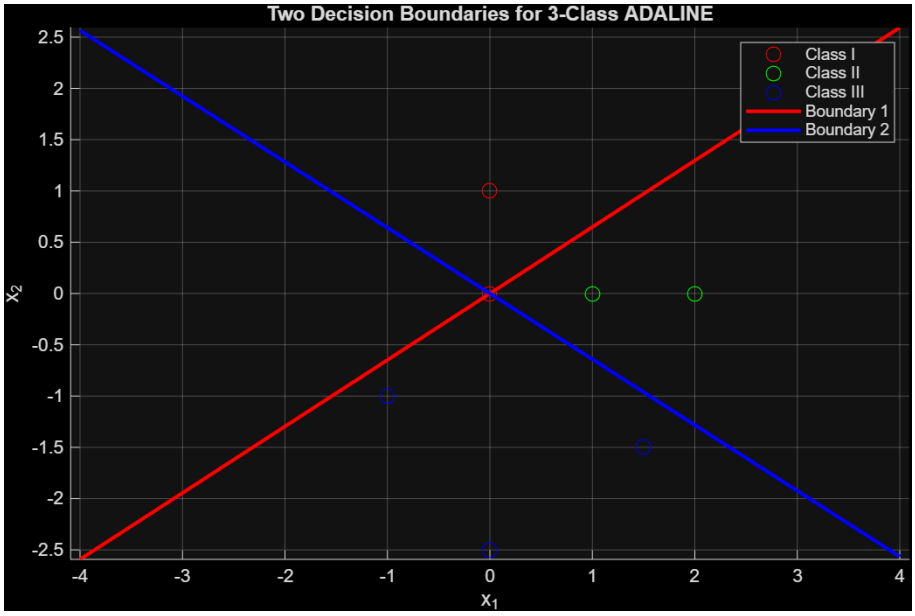
$$\eta_{\max} = \frac{2}{1.23785} \approx 1.6157.$$

Effect of flipping the targets. Replacing $t \to -t$ flips

$$r_{xt} \to -r_{xt}, \qquad w_* \to -w_*,$$

but does *not* change R_{xx}. Hence, the maximum stable learning rate remains

$$\eta_{\max} \text{ unchanged.}$$



Decision boundary corresponding to the least-squares solution w_* (no bias).

MATLAB Code

```
clear; clc; close all;

P = [ 0   0; 0   1;1   0;2  0;-1  -1;0  -2.5;1.5 -1.5 ];

N = size(P,1);
T1 = [-1; -1; -1; -1; -1; -1; -1];
T1(1:2) = 1;
T2 = -1 * ones(N,1);
T2(5:7) = 1;

w1 = (P' * P) \ (P' * T1);
w2 = (P' * P) \ (P' * T2);
disp('Weights for ADALINE 1 (Class I vs Rest):');
disp(w1);
disp('Weights for ADALINE 2 (Class III vs Rest):');
disp(w2);

figure; hold on;
plot(P(1:2,1), P(1:2,2), 'ro', 'MarkerSize', 8);  % Class I
plot(P(3:4,1), P(3:4,2), 'go', 'MarkerSize', 8);  % Class II
plot(P(5:7,1), P(5:7,2), 'bo', 'MarkerSize', 8);  % Class III

x = linspace(-4,4,200);
x_line1 = -(w1(1)/w1(2)) * x;
plot(x, x_line1, 'r', 'LineWidth', 2);

x_line2 = -(w2(1)/w2(2)) * x;
plot(x, x_line2, 'b', 'LineWidth', 2);

title('Two Decision Boundaries for 3-Class ADALINE');
xlabel('x_1'); ylabel('x_2');
axis equal; grid on;
legend('Class I','Class II','Class III','Boundary 1','Boundary 2');
hold off;

w1_range = linspace(w1(1)-5, w1(1)+5, 200);
w2_range = linspace(w1(2)-5, w1(2)+5, 200);
[W1_mesh, W2_mesh] = meshgrid(w1_range, w2_range);

MSE1 = zeros(size(W1_mesh));

for i = 1:numel(W1_mesh)
    w = [W1_mesh(i); W2_mesh(i)];
    y = P * w;
    err = T1 - y;
    MSE1(i) = mean(err.^2);
end

figure;
contour(W1_mesh, W2_mesh, MSE1, 30);
hold on;
plot(w1(1), w1(2), 'rx','MarkerSize',12,'LineWidth',2);
title('MSE Contour Plot  ADALINE 1');
xlabel('w_1'); ylabel('w_2');
grid on;
```

Problem 7

In this exercise we implement the classical Widrow Hoff learning algorithm for training an ADALINE to classify six binary 4×4 patterns representing the letters T, G and F in both their original and shifted forms. Each pattern is first converted into a 16–element binary vector, using the convention that each blue square is encoded as $+1$ and each white square as -1. The vector is formed in column-major order where, we traverse each column from top to bottom before proceeding to the next column.

1. Targets The target outputs assigned to each class are:

$$T \mapsto +60, \qquad G \mapsto 0, \qquad F \mapsto -60,$$

and these same target values are used for the shifted versions of the letters. The six resulting input vectors are collected in a matrix

$$P = [p_1\ p_2\ p_3\ p_4\ p_5\ p_6] \in \mathbb{R}^{16\times 6},$$

with the corresponding target vector

$$T = [60,\ 0,\ -60,\ 60,\ 0,\ -60].$$

2. ADALINE Model The ADALINE consists of a linear neuron with output

$$a = Wp + b,$$

where $W \in \mathbb{R}^{1\times 16}$ is the weight vector and b is the bias. Because the activation function is linear, learning is governed entirely by the Widrow Hoff update rule.

3. Widrow Hoff Learning Rule The error for a given pattern is computed as

$$e = t - a = t - (Wp + b),$$

and the online weight update rule is

$$W \leftarrow W + a\, e\, p^\top, \qquad b \leftarrow b + a\, e,$$

where $a\ = 0.03$ is the learning rate.

At each training step we randomly select one of the six patterns and update the weights immediately after computing its error.

4. Error Metric To evaluate learning progress, we compute at every training step the total sum–squared error (SSE) over all patterns:

$$\text{SSE} = \sum_{i=1}^{6} (t_i - a_i)^2.$$

This value is recorded during training and plotted as a function of the number of pattern presentations. Following Widrow and Hoff, the error is typically displayed on a logarithmic scale to illustrate the approximately exponential decrease in SSE over time. You can clearly see the evolution of SSE in the plot below:

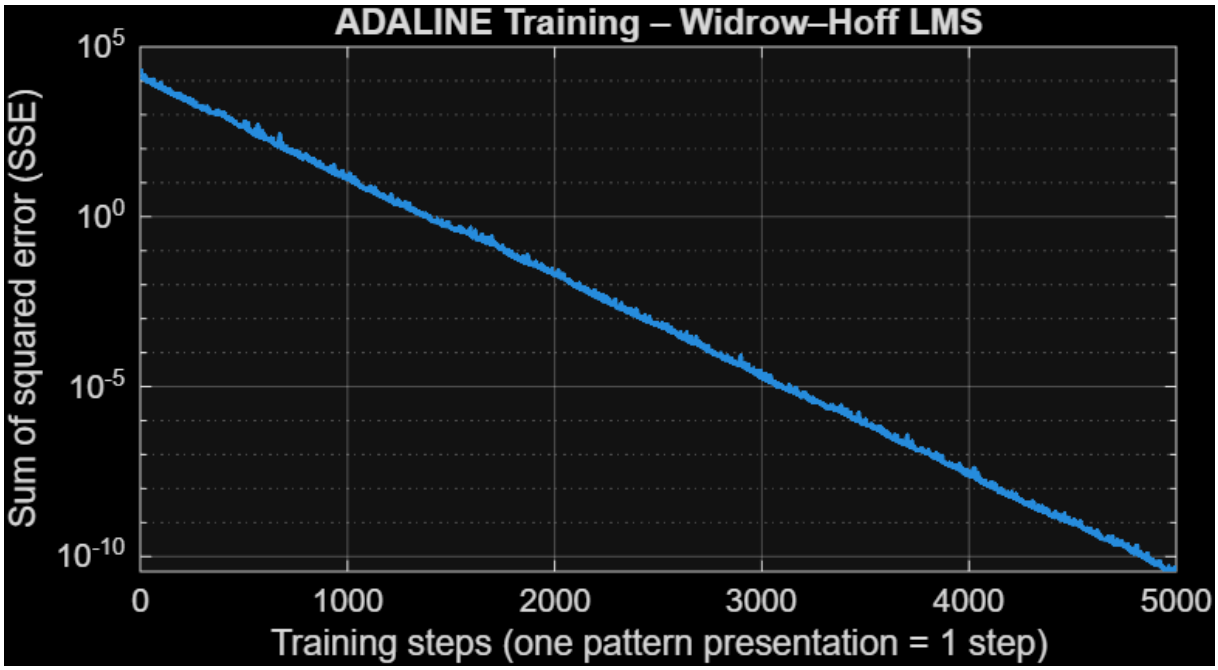


Figure 2: SSE Plot - Widrow Hoff Method

5. Expected Behaviour As training progresses, the ADALINE should converge so that its outputs approach their desired numerical targets:

$$a_{\mathrm{T}} \to +60, \qquad a_{\mathrm{G}} \to 0, \qquad a_{\mathrm{F}} \to -60.$$

Both the original and shifted versions of each letter should converge to the same target value, demonstrating the network's ability to generalize across spatial shifts of the input pattern.

The resulting SSE plot should qualitatively resemble the classical learning curve reported by Widrow and Hoff, showing a smooth monotonic decrease of the total error as the LMS algorithm iteratively adapts the weights.

6. Code: Here is the code implementation we ran to train the neural network with the previous characteristics.

```
p1 = [1,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0];   % T original
p2 = [1,1,1,1,1,0,1,1,1,0,1,1,0,0,0,0];   % G original
p3 = [1,1,1,1,1,1,0,0,1,0,0,0,0,0,0,0];   % F original
p4 = [0,0,0,0,1,0,0,0,1,1,1,1,1,0,0,0];   % T shifted
p5 = [0,0,0,0,1,1,1,1,1,1,0,1,1,1,0,1,1];   % G shifted
p6 = [0,0,0,0,1,1,1,1,1,1,1,0,0,1,0,0,0];   % F shifted

P = [p1' p2' p3' p4' p5' p6'];   % 16*6 input matrix

%% Target values
T = [60 0 -60 60 0 -60];   % 1*6 vector

%% Initialize ADALINE
eta = 0.03;                 % learning rate
W   = zeros(1,16);          % weight row vector
b   = 0;                    % bias

```

```
max_steps = 5000;           % number of training steps
SSE = zeros(max_steps,1);   % record sum-squared error

%% Training loop (online Widrow-Hoff)
for step = 1:max_steps

    % pick a random pattern index
    k = randi(6);
    p = P(:,k);
    t = T(k);

    % forward pass (linear neuron)
    a = W*p + b;

    % error
    e = t - a;

    % weight update (LMS rule)
    W = W + eta * e * p';
    b = b + eta * e;

    % compute SSE across ALL patterns
    A_all = W*P + b;          % 1*6 outputs
    errors = T - A_all;
    SSE(step) = sum(errors.^2);

end

%% Plot SSE vs training steps
figure;
plot(1:max_steps, SSE, 'LineWidth', 1.5);
set(gca, 'YScale','log');     % log scale like Widrow-Hoff Fig 5
xlabel('Training steps (one pattern presentation = 1 step)');
ylabel('Sum of squared error (SSE)');
title('ADALINE Training - Widrow-Hoff LMS');
grid on;

%% Print final outputs
final_outputs = W*P + b
```

Problem 8

A. Contour Plot of the MSE Performance Index

Given the two reference patterns

$$p_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix},\ t_1 = -1, \qquad p_2 = \begin{bmatrix} -2 \\ 1 \end{bmatrix},\ t_2 = 1,$$

and equal probabilities, the correlation matrix and the input–target correlation vector are

$$R = \tfrac{1}{2}(p_1 p_1^T + p_2 p_2^T) = \begin{bmatrix} 2.5 & 0 \\ 0 & 2.5 \end{bmatrix}, \qquad r_{pt} = \tfrac{1}{2}(p_1 t_1 + p_2 t_2) = \begin{bmatrix} -1.5 \\ -0.5 \end{bmatrix}.$$

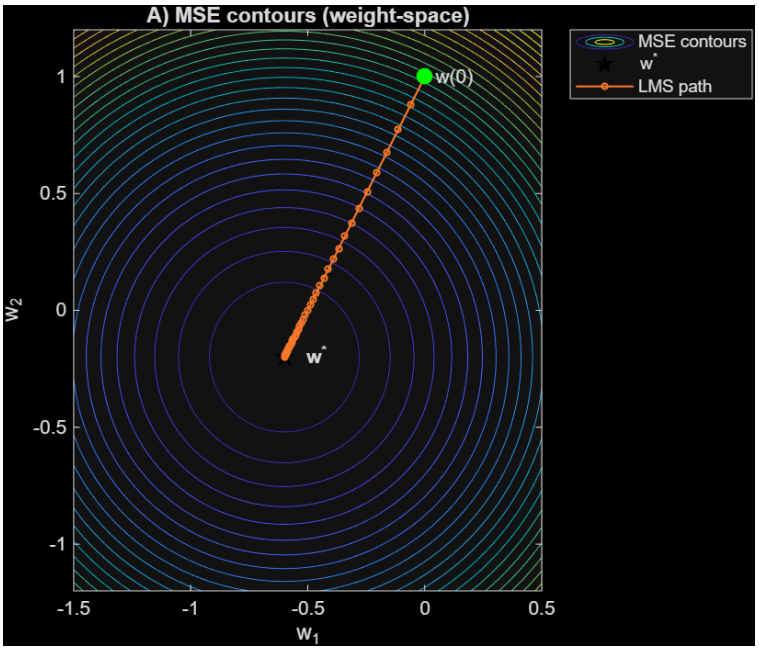
The optimal ADALINE weight vector is

$$w^* = R^{-1} r_{pt} = \begin{bmatrix} -0.6 \\ -0.2 \end{bmatrix}.$$

The mean-square error is

$$J(w) = \frac{1}{2}\Big[(t_1 - w^T p_1)^2 + (t_2 - w^T p_2)^2\Big] = w^T R w - 2 w^T r_{pt} + 1.$$

Since $R = 2.5I$, all level sets of $J(w)$ are concentric circles centered at w^*.



Contour plot of the MSE performance index and optimal weight w^*.

B. Optimal Decision Boundary

For an ADALINE network without bias, the decision boundary satisfies

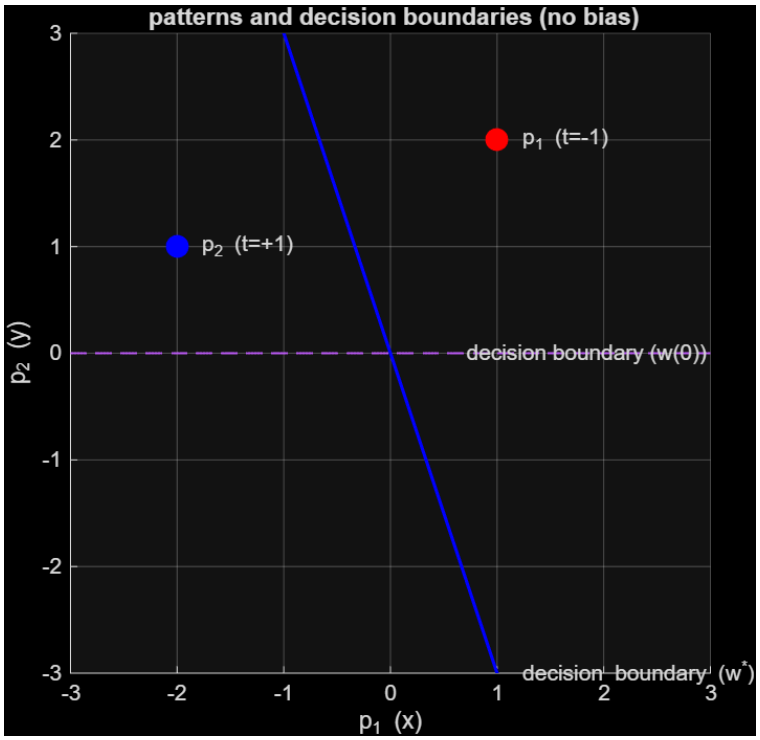
$$w^T p = 0.$$

Using the optimal weights,

$$-0.6x - 0.2y = 0 \quad \Rightarrow \quad y = -3x.$$

The initial weights $w(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ give the decision boundary

$$y = 0.$$



Decision boundaries
corresponding to w^* and $w(0)$.

C. LMS Trajectory on the MSE Surface

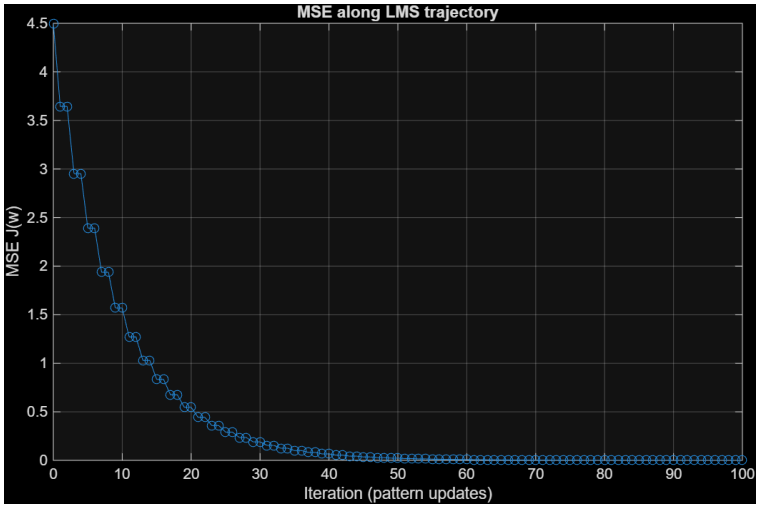
The LMS update rule is

$$w(k+1) = w(k) + \mu\, p_i\Big(t_i - w(k)^T p_i\Big),$$

applied with a small learning rate and cyclic presentation of the patterns. Starting at

$$w(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

the weight vector moves along the negative gradient of the MSE surface and converges toward w^*.



LMS learning trajectory superimposed on the MSE contour plot.

Listing 4: MATLAB code

```
clear; close all; clc;

% Data
p1 = [1;2]; t1 = -1;
p2 = [-2;1]; t2 = 1;

R = 0.5*(p1*p1' + p2*p2');     % = 2.5*I
r_pt = 0.5*(p1*t1 + p2*t2);    % = [-1.5; -0.5]
w_star = R \ r_pt;             % = [-0.6; -0.2]

%%  Contour plot of MSE
w1 = linspace(-1.5,0.5,300);
w2 = linspace(-1.2,1.2,300);
[W1,W2] = meshgrid(w1,w2);
J = zeros(size(W1));
for i = 1:numel(W1)
    w = [W1(i); W2(i)];
    e1 = t1 - w' * p1;
    e2 = t2 - w' * p2;
    J(i) = 0.5*(e1^2 + e2^2);
end

figure(1);
contour(W1,W2,J,30); axis equal; hold on
plot(w_star(1), w_star(2), 'kp','MarkerFaceColor','k','MarkerSize',10)
text(w_star(1), w_star(2),'  w^*','FontWeight','bold')
xlabel('w_1'); ylabel('w_2');
title('MSE contours (weight-space)')

%% LMS trajectory
w = [0;1];                  % initial weight W(0)
mu = 0.02;                  % very small learning rate (adjustable)
steps = 100;                % number of pattern updates
trajectory = zeros(2,steps+1);
trajectory(:,1) = w;
for k = 1:steps
    idx = mod(k-1,2)+1;
    if idx==1, p=p1; t=t1; else p=p2; t=t2; end
    e = t - w'*p;
    w = w + mu * p * e;
    trajectory(:,k+1) = w;
end
plot(trajectory(1,:), trajectory(2,:), '-o', 'MarkerSize',3, 'LineWidth',1);
plot(trajectory(1,1), trajectory(2,1), 'go','MarkerFaceColor','g','MarkerSize',8)
text(trajectory(1,1), trajectory(2,1),'  w(0)')
legend('MSE contours','w^*','LMS path','Location','northeastoutside')
hold off
```

```
%% Input-space: patterns and decision boundaries
figure(2); hold on; grid on; axis equal
plot(p1(1),p1(2),'ro','MarkerFaceColor','r','MarkerSize',10); text(p1(1),p1(2),'  p_1 (t=-1)')
plot(p2(1),p2(2),'bo','MarkerFaceColor','b','MarkerSize',10);

text(p2(1),p2(2),'  p_2 (t=+1)')
x = linspace(-3,3,200);
% boundary for w* : y = -3x
y_star = -3*x;
plot(x,y_star,'k-','LineWidth',1.5)
text(1, -3*1, '  decision boundary (w^*)')

% initial boundary w(0) = [0;1] -> y = 0
plot(x, zeros(size(x)), '--','LineWidth',1)
text(0.6, 0, '  decision boundary (w(0))')

xlabel('p_1 (x)'); ylabel('p_2 (y)');
title('Input-space: patterns and decision boundaries (no bias)')
xlim([-3 3]); ylim([-3 3]); hold off

%% Optional: plot J along iterations
J_traj = zeros(1,size(trajectory,2));
for i=1:size(trajectory,2)
    wtmp = trajectory(:,i);
    e1 = t1 - wtmp' * p1;
    e2 = t2 - wtmp' * p2;
    J_traj(i) = 0.5*(e1^2 + e2^2);
end
figure(3);
plot(0:steps, J_traj, '-o'); xlabel('Iteration (pattern updates)'); ylabel('MSE J(w)')
    ;
title('MSE along LMS trajectory');
grid on;
```

Problem 9

explanation The provided script trains a single-hidden-layer neural network (1 input $\to S_1$ sigmoid hidden units $\to$ 1 linear output) to approximate

$$g(p) = 1 + \sin\left(\frac{\pi}{3}p\right),$$

on $N = 201$ evenly spaced points $p \in [-2, 2]$. Training uses full-batch gradient descent with MSE loss:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2, \qquad d_i = \frac{2}{N}(y_i - t_i).$$

Vectorized forward pass (all samples at once):

$$Z_1 = W_1 P + b_1, \quad A_1 = \sigma(Z_1), \quad Y = W_2 A_1 + b_2,$$

with shapes $P : 1 \times N$, $W_1 : S_1 \times 1$, $A_1 : S_1 \times N$, $W_2 : 1 \times S_1$.

Backprop (compact):

$$\nabla_{W_2} = d\, A_1^\top, \quad \nabla_{b_2} = \sum d, \quad \nabla_{z_1} = (W_2^\top d) \odot (A_1 \odot (1 - A_1)),$$

$$\nabla_{W_1} = \nabla_{z_1} P^\top, \quad \nabla_{b_1} = \sum \nabla_{z_1},$$

and standard gradient-descent updates $W \leftarrow W - \alpha \nabla_W$.

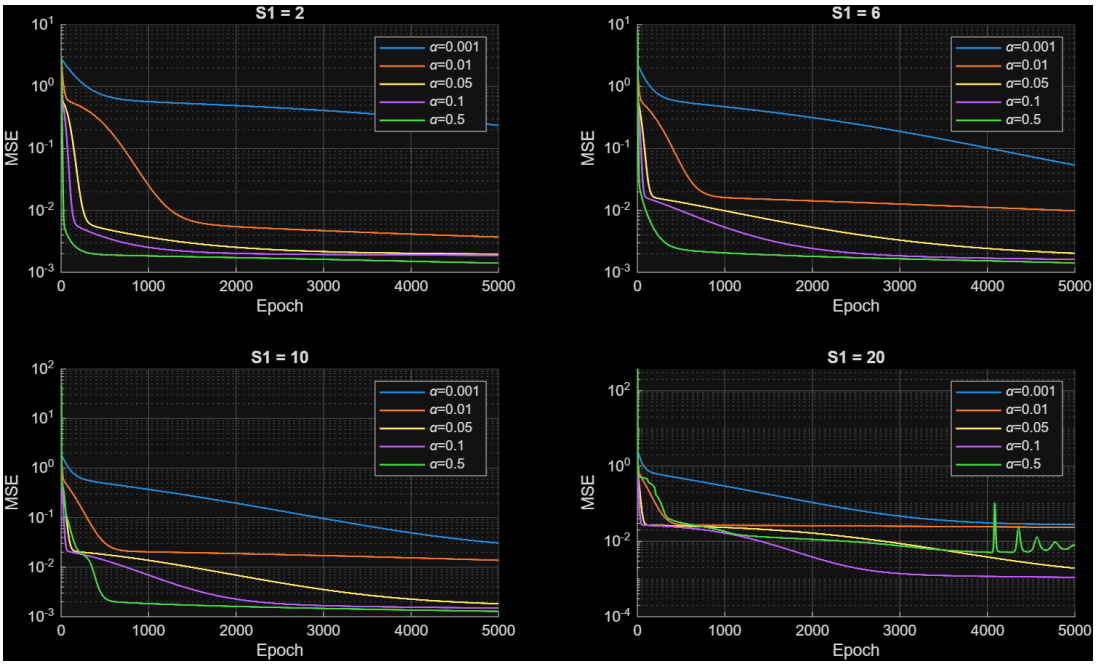
The script sweeps hidden sizes S_1, learning rates α and random seeds, stores results in a structured container, detects NaN/divergence, and plots mean MSE curves (log-scale) and final approximations.

```
1
2
3  % backprop_1S1_fixed_keys_v2.m
4  % Corrected version: sanitizes dynamic struct field names and provides a
5  % compatible NaN-ignoring mean for MATLAB installations without nanmean.
6
7  clearvars; close all; clc;
8
9  %% Data
10 N = 201;
11 p = linspace(-2,2,N);
12 t = 1 + sin(p * pi/3);
13
14 P = reshape(p,1,[]);
15 T = reshape(t,1,[]);
16
17 %% Experiment settings
18 S1_list = [2, 6, 10, 20];
19 alpha_list = [0.001, 0.01, 0.05, 0.1, 0.5];
20 seeds = [1, 7, 42];
21 epochs = 5000;
22
23 results = struct();
24
25 % Helper to create a valid MATLAB struct field name
26 makeField = @(S1,alpha,seed) sprintf('S1_%02d_alpha_%s_seed_%d', ...
27     S1, regexprep(sprintf('%.6g',alpha),'[^0-9A-Za-z]','p'), seed);
28
```

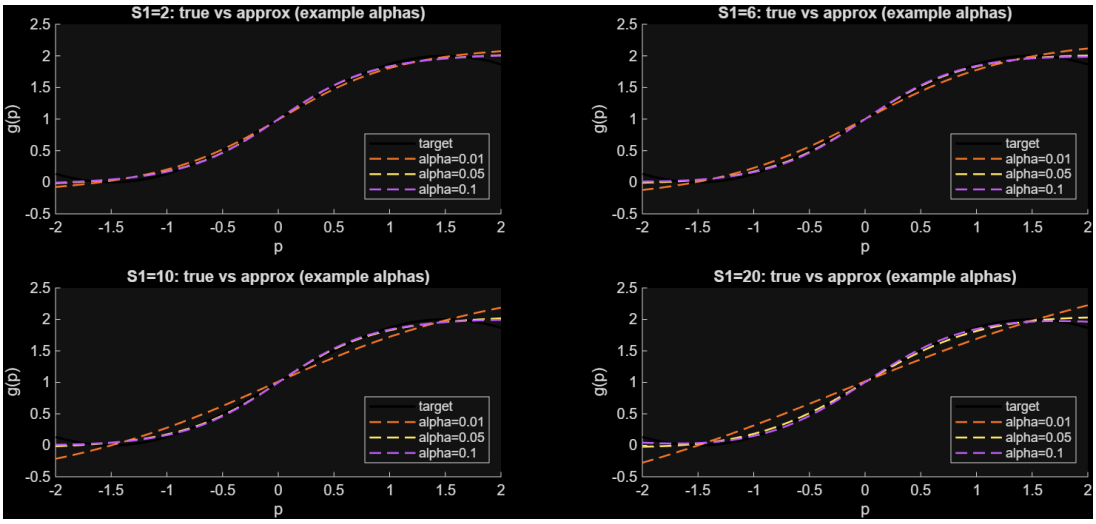
```
1
2  % Compatible NaN-ignoring mean function (use nanmean if available)
3  if exist('nanmean','file') == 2
4      safe_nanmean = @(x,dim) nanmean(x,dim);
5  else
6      safe_nanmean = @(x,dim) mean(x,dim,'omitnan');
7  end
8
9  %% Training loop
10 for sIdx = 1:numel(S1_list)
11     S1 = S1_list(sIdx);
12     for aIdx = 1:numel(alpha_list)
13         alpha = alpha_list(aIdx);
14         for seedIdx = 1:numel(seeds)
15             seed = seeds(seedIdx);
16             rng(seed);
17
18             W1 = rand(S1,1) - 0.5;
19             b1 = rand(S1,1) - 0.5;
20             W2 = rand(1,S1) - 0.5;
21             b2 = rand(1,1) - 0.5;
22
23             mse_hist = zeros(1,epochs);
24
25             for ep = 1:epochs
26                 Z1 = W1 * P + b1;
27                 A1 = 1 ./ (1 + exp(-Z1));
28                 Y  = W2 * A1 + b2;
29
30                 E = Y - T;
31                 mse = mean(E.^2);
32                 mse_hist(ep) = mse;
33
34                 dE_dy = (2 / N) * E;
35
36                 gradW2 = dE_dy * A1.';
37                 gradb2 = sum(dE_dy, 2);
38
39                 dE_da1 = (W2.' * dE_dy);
40                 dA1_dZ1 = A1 .* (1 - A1);
41                 dE_dz1 = dE_da1 .* dA1_dZ1;
42
43                 gradW1 = dE_dz1 * P.';
44                 gradb1 = sum(dE_dz1, 2);
45
46                 W2 = W2 - alpha * gradW2;
47                 b2 = b2 - alpha * gradb2;
48                 W1 = W1 - alpha * gradW1;
49                 b1 = b1 - alpha * gradb1;
50
51                 if any(isnan([W1(:);W2(:);b1(:);b2(:)]))
52                     mse_hist(ep+1:end) = NaN;
53                     break;
54                 end
55             end
56
```

```
            key = makeField(S1, alpha, seed);
            results.(key).W1 = W1;
            results.(key).b1 = b1;
            results.(key).W2 = W2;
            results.(key).b2 = b2;
            results.(key).mse_hist = mse_hist;

            Z1 = W1 * P + b1;
            A1 = 1 ./ (1 + exp(-Z1));
            Y  = W2 * A1 + b2;
            results.(key).Y = Y;
        end
    end
end

%% Visualization: MSE convergence for each S1 and alpha (averaged across seeds)
figure('Name','MSE convergence (avg over seeds)','Units','normalized','Position',[0.05
     0.05 0.9 0.8]);
for sIdx = 1:numel(S1_list)
    S1 = S1_list(sIdx);
    subplot(2,2,sIdx);
    hold on;
    title(sprintf('S1 = %d', S1));
    xlabel('Epoch'); ylabel('MSE');
    for aIdx = 1:numel(alpha_list)
        alpha = alpha_list(aIdx);
        stacked = zeros(numel(seeds), epochs);
        for seedIdx = 1:numel(seeds)
            seed = seeds(seedIdx);
            key = makeField(S1, alpha, seed);
            if isfield(results, key)
                stacked(seedIdx,:) = results.(key).mse_hist;
            else
                stacked(seedIdx,:) = NaN(1,epochs);
            end
        end
        mean_mse = safe_nanmean(stacked,1);
        plot(1:epochs, mean_mse, 'LineWidth', 1.2);
    end
    legend(arrayfun(@(a) sprintf('\\alpha=%.3g',a), alpha_list, 'UniformOutput',false)
    , 'Location','northeast');
    set(gca,'YScale','log'); grid on;
    hold off;
end

%% Visualization: final approximations for selected configs
figure('Name','Function approximation examples','Units','normalized','Position',[0.05
    0.05 0.9 0.6]);
example_alphas = [0.01, 0.05, 0.1];
for sIdx = 1:numel(S1_list)
    S1 = S1_list(sIdx);
    subplot(2,2,sIdx);
    hold on;
    plot(P, T, 'k-', 'LineWidth', 1.5); % true function
    for aIdx = 1:numel(example_alphas)
        alpha = example_alphas(aIdx);

```

```
        key = makeField(S1, alpha, seeds(1)); % pick seed 1 as example
        if isfield(results, key)
            Y = results.(key).Y;
            plot(P, Y, '--', 'LineWidth', 1.2);
        end
    end
    title(sprintf('S1=%d: true vs approx (example alphas)', S1));
    xlabel('p'); ylabel('g(p)');
    legend(['target', arrayfun(@(a) sprintf('alpha=%.3g',a), example_alphas, '
    UniformOutput',false)], 'Location','best');
    hold off;
end

%% Summarize final MSEs (print table)
fprintf('\nSummary of final MSEs (rows: S1, columns: alpha; values averaged over seeds
    )\n');
for sIdx = 1:numel(S1_list)
    S1 = S1_list(sIdx);
    row = zeros(1,numel(alpha_list));
    for aIdx = 1:numel(alpha_list)
        alpha = alpha_list(aIdx);
        vals = zeros(1,numel(seeds));
        for seedIdx = 1:numel(seeds)
            seed = seeds(seedIdx);
            key = makeField(S1, alpha, seed);
            if isfield(results,key)
                lastIdx = find(~isnan(results.(key).mse_hist),1,'last');
                if isempty(lastIdx)
                    vals(seedIdx) = NaN;
                else
                    vals(seedIdx) = results.(key).mse_hist(lastIdx);
                end
            else
                vals(seedIdx) = NaN;
            end
        end
        row(aIdx) = safe_nanmean(vals,2);
    end
    fprintf('S1=%2d : ', S1);
    fprintf('  %.3e', row);
    fprintf('\n');
end
```



Average MSE vs epoch (averaged across seeds) for each S_1 and α.



Example function approximations (true vs learned) for selected α values.

Problem 10

We consider the modified performance function

$$E = \sum_i (t_i - a_i)^4 \; + \; \sum_{m=1}^{M} \Big(\|W^m\|_F^2 + \|b^m\|_2^2 \Big),$$

where $e = t - a$ denotes the output error. The forward propagation equations of the network remain unchanged:

$$a^0 = p, \qquad a^{m+1} = f^{m+1}(W^{m+1}a^m + b^{m+1}), \qquad a = a^M.$$

Output-layer sensitivity. For the new error term

$$E_{\text{err}} = \sum_i (t_i - a_i)^4,$$

the derivative with respect to the output a is

$$\frac{\partial E_{\text{err}}}{\partial a} = -4\,(t-a)^{\odot 3}.$$

Thus, using the activation derivative matrix $F'^M(n^M)$, the output-layer sensitivity becomes

$$s^M = F'^M(n^M)\,\frac{\partial E_{\text{err}}}{\partial a} = -4\,F'^M(n^M)\,(t-a)^{\odot 3}.$$

Hidden-layer sensitivities. The backward recursion is unchanged in form:

$$s^m = F'^m(n^m)\,(W^{m+1})^T s^{m+1}, \qquad m = M-1, \ldots, 1.$$

Gradients with respect to weights and biases. The gradient contribution from the error term remains

$$\frac{\partial E_{\text{err}}}{\partial W^m} = s^m (a^{m-1})^T, \qquad \frac{\partial E_{\text{err}}}{\partial b^m} = s^m.$$

The regularization term contributes

$$\frac{\partial}{\partial W^m}\|W^m\|_F^2 = 2W^m, \qquad \frac{\partial}{\partial b^m}\|b^m\|_2^2 = 2b^m.$$

Hence the full gradients are

$$\frac{\partial E}{\partial W^m} = s^m (a^{m-1})^T + 2W^m, \qquad \frac{\partial E}{\partial b^m} = s^m + 2b^m.$$

Updated parameter rules. Using the steepest descent update with learning rate α,

$$W^m(k+1) = W^m(k) - \alpha\Big[s^m (a^{m-1})^T + 2W^m(k)\Big],$$

$$b^m(k+1) = b^m(k) - \alpha\Big[s^m + 2b^m(k)\Big].$$