Problem-01

Gradient

Given the separable quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n} \lambda_i x_i^2 \tag{1}$$

We first compute the gradient vector $\nabla F(\mathbf{x})$. The partial derivative with respect to any component x_i is:

$$\frac{\partial F}{\partial x_i} = \frac{1}{2} \cdot 2\lambda_i x_i = \lambda_i x_i \tag{2}$$

Thus, the gradient is:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \lambda_1 x_1 \\ \lambda_2 x_2 \\ \vdots \\ \lambda_n x_n \end{bmatrix} \tag{3}$$

Gradient Descent Update Rule

The standard Gradient Descent update rule with learning rate α is given by:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla F(\mathbf{x}^{(k)}) \tag{4}$$

Substituting the gradient derived above, we can write the update for each individual component i:

$$x_i^{(k+1)} = x_i^{(k)} - \alpha(\lambda_i x_i^{(k)}) \tag{5}$$

$$x_i^{(k+1)} = x_i^{(k)}(1 - \alpha\lambda_i) \tag{6}$$

This recurrence relation shows that the value of each coordinate at step k is:

$$x_i^{(k)} = x_i^{(0)}(1 - \alpha\lambda_i)^k \tag{7}$$

Initial Condition $\mathbf{x}^{(0)}$

We are given the constraints $0 < m = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n = M$ and the starting vector:

$$\mathbf{x}^{(0)} = \left[\frac{1}{m}, 0, \ldots, 0, \frac{1}{M}\right]^T \tag{8}$$

We analyze the trajectory component-wise:

- **Middle components ($1 < i < n$):**
  Since $x_i^{(0)} = 0$, the update rule $x_i^{(k+1)} = 0 \cdot (1 - \alpha\lambda_i)$ implies these components remain **0** for all iterations.

- **First component ($i = 1$):**
  With $\lambda_1 = m$ and starting value $1/m$:

$$x_1^{(k)} = \frac{1}{m}(1 - \alpha m)^k \tag{9}$$

- **Last component ($i = n$):**
  With $\lambda_n = M$ and starting value $1/M$:

$$x_n^{(k)} = \frac{1}{M}(1 - \alpha M)^k \tag{10}$$

Minimum

Since $\lambda_i > 0$, the function is strictly convex and the global minimum occurs at $\nabla F(\mathbf{x}) = \mathbf{0}$, which implies $\mathbf{x}^* = \mathbf{0}$.

Assuming a step size α is chosen to ensure convergence (specifically $\alpha < \frac{2}{M}$), the terms $(1 - \alpha\lambda_i)^k$ vanish as $k \to \infty$. Thus, the algorithm converges to the origin:

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{11}$$

The minimum function value is:

$$F(\mathbf{x}^*) = F(\mathbf{0}) = 0 \tag{12}$$

Problem-02

Newton's Method

Newton's method update rule for optimization is defined as:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{13}$$

First, we determine the first and second derivatives of $f(x)$:

$$\begin{aligned} f(x) &= x^4 - 1 \\ f'(x) &= 4x^3 \\ f''(x) &= 12x^2 \end{aligned}$$

Substituting these derivatives into the Newton's update formula:

$$\begin{aligned} x_{n+1} &= x_n - \frac{4x_n^3}{12x_n^2} \\ x_{n+1} &= x_n - \frac{1}{3}x_n \\ x_{n+1} &= \frac{2}{3}x_n \end{aligned}$$

Condition for Convergence

The recursive relation derived above, $x_{n+1} = \frac{2}{3}x_n$, represents a geometric sequence. The general term for the n-th iteration based on an initial guess x_0 is:

$$x_n = \left(\frac{2}{3}\right)^n x_0$$

The true minimum of $f(x) = x^4 - 1$ is at $x^* = 0$. For the sequence to converge to 0, the common ratio r must satisfy $|r| < 1$. Here, $r = \frac{2}{3}$. Since $\frac{2}{3} < 1$, the sequence converges to 0 as $n \to \infty$ regardless of the starting point.

Conclusion: The method is globally convergent for any finite initial guess $x_0 \neq 0$.

Rate of Convergence

Newton's method typically exhibits quadratic convergence (order 2). However, this holds only if the second derivative at the minimum is non-zero ($f''(x^*) \neq 0$). In our case, at the optimal point $x^* = 0$, the Hessian is:

$$f''(0) = 12(0)^2 = 0$$

Since the second derivative vanishes at the solution, the convergence speed degrades. We can observe the error relationship directly. Let the error at step n be $e_n = |x_n - x^*| = |x_n|$.

$$e_{n+1} = |x_{n+1}| = \left|\frac{2}{3}x_n\right| = \frac{2}{3}|x_n| = \frac{2}{3}e_n$$

Since e_{n+1} is proportional to e_n to the first power ($e_{n+1} \approx C \cdot e_n^1$), the rate of convergence is **Linear**.

Problem-03

Mathematical Formulation

To apply the Conjugate Gradient method, we first express the function in the standard quadratic form:

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c$$

Calculating the gradient $\nabla f(x)$ allows us to identify the Hessian matrix A:

$$\nabla f(x) = \begin{bmatrix} 2x_1 \\ 10x_2 \\ 6x_3 \end{bmatrix}$$

Since we are minimizing $f(x)$, we solve for $\nabla f(x) = 0$, which corresponds to the linear system $Ax = b$. By inspection of coefficients:

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The analytical minimum is clearly at the origin $x^* = [0, 0, 0]^T$. The Conjugate Gradient algorithm iteratively refines the starting position x_0 using conjugate directions d_k and step sizes α_k.

MATLAB Implementation

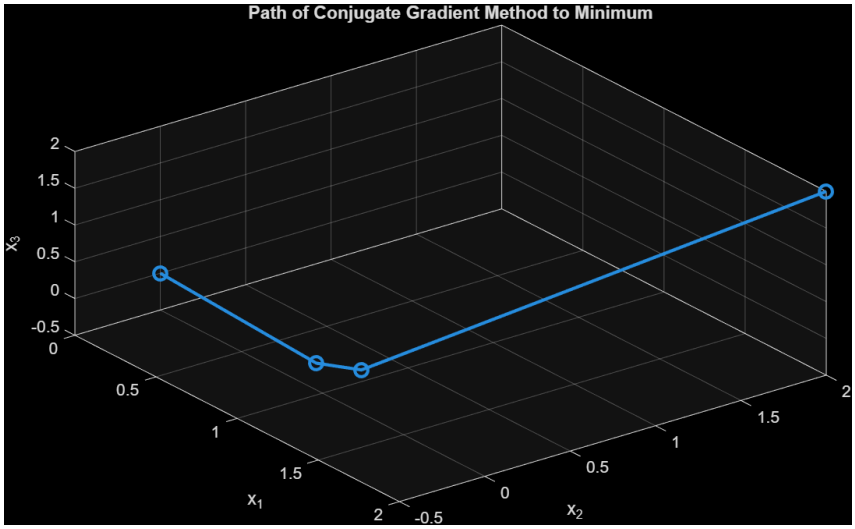
The following code snippets highlight the initialization of the matrix A and the core iterative loop of the Conjugate Gradient method used to solve the system.

```
A = [2 0 0; 0 10 0; 0 0 6];
b = [0; 0; 0];
x = [2; 2; 2];

r = b - A*x;     % Initial residual
d = r;           % Initial direction

for i = 1:3
    alpha = (r' * r) / (d' * A * d);    % Calculate step size
    x = x + alpha * d;

    r_new = r - alpha * A * d;

    % Convergence Check
    if norm(r_new) < 1e-10
        break;
    end

    beta = (r_new' * r_new) / (r' * r); % Fletcher-Reeves update
    d = r_new + beta * d;
    r = r_new;
end
```

Listing 1: Core Conjugate Gradient Logic

Results

The algorithm successfully converges to the minimum $x^* = [0, 0, 0]^T$. The trajectory of the descent in 3D space is visualized below.



Problem 4

At this problem we analyze the impact of weight pruning on a Multi-Layer Perceptron (MLP) with $S_1 = 80$ hidden neurons, trained to approximate the function $g(p) = 1 + \sin(p\pi/3)$. We compare three pruning strategies: Smallest Magnitude, Random, and Largest Magnitude.

Before pruning, we train a dense model to convergence. The large hidden layer ($S_1 = 80$) ensures the model is over-parameterized for a simple sine wave, providing fertile ground for pruning.

```
clearvars; close all; clc;

% Data Generation
N = 201; p = linspace(-2, 2, N); t = 1 + sin(p * pi/3);
P = reshape(p, 1, []); T = reshape(t, 1, []);

% Baseline Training (S1=80)
S1 = 80; alpha = 0.05; epochs = 5000; rng(42);
W1 = rand(S1, 1) - 0.5; b1 = rand(S1, 1) - 0.5;
W2 = rand(1, S1) - 0.5; b2 = rand(1, 1) - 0.5;

for ep = 1:epochs
    Z1 = W1 * P + b1; A1 = 1 ./ (1 + exp(-Z1)); Y = W2 * A1 + b2;
    E = Y - T; dE_dy = (2 / N) * E;
    gradW2 = dE_dy * A1.'; gradb2 = sum(dE_dy, 2);
    dE_da1 = (W2.' * dE_dy); dA1_dZ1 = A1 .* (1 - A1);
    dE_dz1 = dE_da1 .* dA1_dZ1;
    gradW1 = dE_dz1 * P.'; gradb1 = sum(dE_dz1, 2);
    W2 = W2 - alpha * gradW2; b2 = b2 - alpha * gradb2;
    W1 = W1 - alpha * gradW1; b1 = b1 - alpha * gradb1;
end

% Save Golden Weights
W1_orig = W1; b1_orig = b1; W2_orig = W2; b2_orig = b2;
all_w_orig = [W1_orig(:); W2_orig(:)];
```

Listing 2: Baseline Training and Data Setup

1 Task A: Smallest Magnitude Pruning

1.1 Methodology

We identify the threshold value for the absolute magnitude $|w|$ corresponding to the x-th percentile. All weights below this threshold are set to zero.

```
prune_levels = 5:2:25; mse_A = [];
for p_val = prune_levels
    threshold = prctile(abs(all_w_orig), p_val);
    W1_p = W1_orig .* (abs(W1_orig) > threshold);
    W2_p = W2_orig .* (abs(W2_orig) > threshold);
    Y_p = W2_p * (1 ./ (1 + exp(-(W1_p * P + b1_orig)))) + b2_orig;
    mse_A = [mse_A, mean((Y_p - T).^2)];
end
```

Listing 3: Task A Implementation

1.2 Theoretical Analysis

Magnitude pruning assumes that small weights contribute primarily to "fine-tuning" or noise fitting. In an over-parameterized model ($S_1 = 80$), many neurons are either redundant or dead. Removing the smallest 25% of weights usually results in a negligible increase in MSE, demonstrating the high level of redundancy in the network.

2 Task B: Random Pruning

2.1 Methodology

A random subset of weight indices is selected and zeroed, regardless of their magnitude. This serves as a control group.

```
mse_B = [];
total_w = numel(all_w_orig);
for p_val = prune_levels
    num_p = round((p_val/100) * total_w);
    idx = randperm(total_w, num_p);
    temp_w = all_w_orig; temp_w(idx) = 0;
    W1_r = reshape(temp_w(1:numel(W1_orig)), size(W1_orig));
    W2_r = reshape(temp_w(numel(W1_orig)+1:end), size(W2_orig));
    Y_r = W2_r * (1 ./ (1 + exp(-(W1_r * P + b1_orig)))) + b2_orig;
    mse_B = [mse_B, mean((Y_r - T).^2)];
end
```

Listing 4: Task B Implementation

2.2 Comparison and Generalization

Random pruning is significantly more destructive than magnitude pruning. By randomly zeroing weights, we risk removing "critical" high-magnitude connections. **Generalization:** Pruning acts as a regularizer. While training error might increase slightly, pruning small weights helps the model ignore high-frequency noise in the training set, potentially improving performance on unseen "clean" data.

3 Task C: Largest Magnitude Pruning & Super-weights

3.1 Methodology

We prune the "top" weights (1.25% to 5%). These are the weights with the highest absolute values.

```
levels_C = 1.25:1.25:5; mse_C = [];
for p_val = levels_C
    threshold = prctile(abs(all_w_orig), 100 - p_val);
    W1_c = W1_orig .* (abs(W1_orig) < threshold);
    W2_c = W2_orig .* (abs(W2_orig) < threshold);
    Y_c = W2_c * (1 ./ (1 + exp(-(W1_c * P + b1_orig)))) + b2_orig;
    mse_C = [mse_C, mean((Y_c - T).^2)];
end
```

Listing 5: Task C Implementation

3.2 Super-weight Analysis

As observed in Large Language Models (LLMs), specific "outlier" weights often carry the bulk of the network's knowledge. In this experiment, pruning just 5% of the largest weights causes the MSE to explode. This confirms the presence of **Super-weights**: a small subset of connections that define the fundamental geometry of the approximation function.

Results

The experiments demonstrate that:

- **Redundancy:** Magnitude pruning (Task A) is safe because the model is over-parameterized.
- **Sensitivity:** Task C proves that weight importance is not uniform. The largest weights are critical "Super-weights."
- **Regularization:** Pruning small weights can serve as a generalization tool, effectively "denoising" the model's logic.

Problem-05

Adadelta Implementation with

Adadelta adapts the learning rates based on a window of accumulated past gradients. We apply the learning rate α to the update step.

The gradient of the normal function is:

$$\nabla F(\mathbf{w}) = \begin{bmatrix} 0.2w_1 \\ 4w_2 \end{bmatrix} \tag{14}$$

The core update step used in the code is:

```
% Accumulate Gradient (Eg2)
Eg2 = rho * Eg2 + (1 - rho) * g.^2;

% Compute Step (dx) using RMS of previous updates and current gradient
dx  = -(sqrt(Edx2 + eps) ./ sqrt(Eg2 + eps)) .* g;

% Update Weights with learning rate alpha
w   = w + alpha * dx;

% Accumulate Updates (Edx2)
Edx2 = rho * Edx2 + (1 - rho) * dx.^2;
```

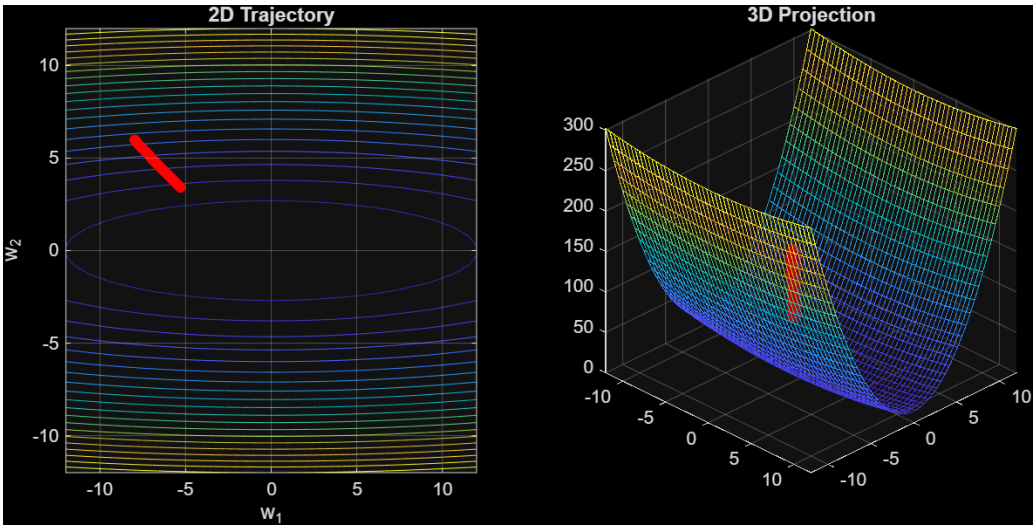


Figure 1: Trajectory with $\alpha = 0.4$. The path starts correctly but moves slowly due to the low learning rate and Adadelta's "cold start" property.

Adadelta with $\alpha = 3$

We change the learning rate to $\alpha = 3$. This effectively scales the Adadelta step, allowing the optimizer to cover more ground and converge faster toward the minimum at $(0, 0)$.

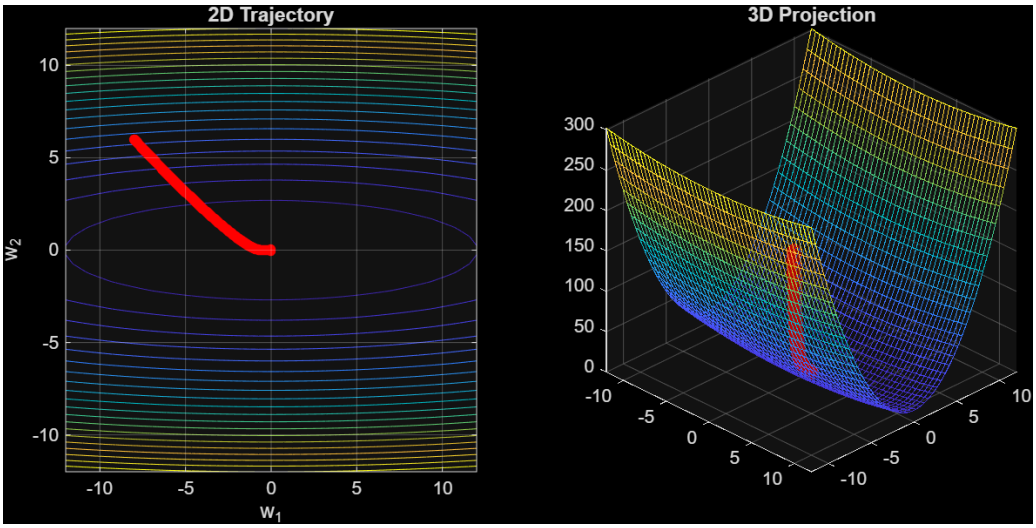


Figure 2: Trajectory with $\alpha = 3.0$. The larger learning rate overcomes the initial dampening, resulting in a complete trajectory to the minimum.

Rotated Function

We apply the optimizer to the rotated function:

$$F(\mathbf{w}) = 0.1(w_1 + w_2)^2 + 2(w_1 - w_2)^2$$

For the implementation, we expanded the gradient terms analytically to simplify the

computation:

$$\frac{\partial F}{\partial w_1} = 4.2w_1 - 3.8w_2$$
$$\frac{\partial F}{\partial w_2} = -3.8w_1 + 4.2w_2$$

```
% Rotated Gradient Implementation
G = @(w) [4.2*w(1) - 3.8*w(2); ...
          -3.8*w(1) + 4.2*w(2)];
```

Listing 6: Expanded Gradient for Rotated Function

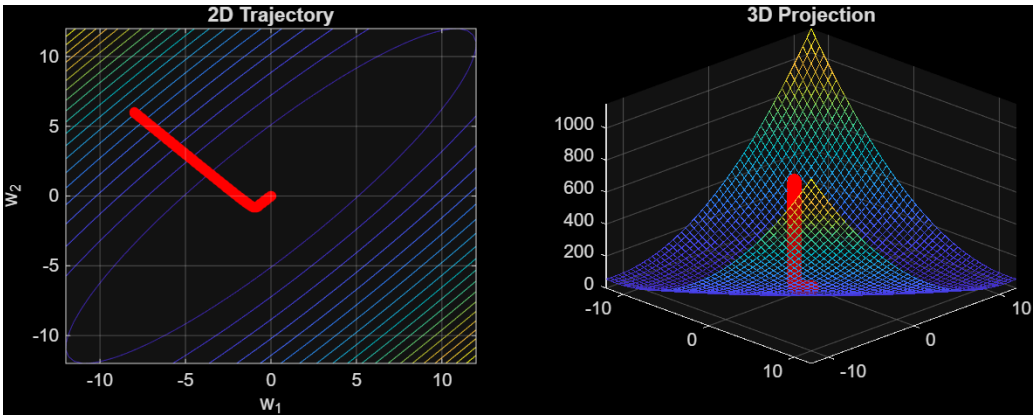


Figure 3: Trajectory on the rotated function. The correlation between parameters causes the path to curve significantly as Adadelta adjusts each coordinate scale independently.

Analysis of Behavior:

- **Effect of α:** With $\alpha = 0.4$, the progress is slow because the accumulated updates ($E[\Delta x^2]$) start at zero. Increasing α to 3.0 compensates for this, allowing full convergence.
- **Rotation:** In the rotated case, the variables w_1 and w_2 are highly correlated. Since Adadelta uses a diagonal approximation (scaling each parameter independently), it cannot perfectly "uncouple" the dimensions, leading to a curved trajectory compared to the direct path in the axis-aligned case.

Problem-06

Given the quadratic function:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 4 & -4 \end{bmatrix} \mathbf{x}$$

We identify the Hessian matrix $\mathbf{A}$ and the vector $\mathbf{d}$:

$$\mathbf{A} = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} 4 \\ -4 \end{bmatrix}$$

The gradient is given by $\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}$. Given parameters: Learning rate $\alpha = 1$, Momentum $\gamma = 0.75$, Initial condition $\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

1. **Perform two iterations of steepest descent with momentum.**

   The update rules are:

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} - \alpha \nabla F(\mathbf{x}_k)$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_k$$

   (Assuming $\mathbf{v}_{-1} = \mathbf{0}$).

   Iteration 1 ($k = 0$):

$$\nabla F(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 + \mathbf{d} = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 4 \\ -4 \end{bmatrix} = \begin{bmatrix} 4 \\ -4 \end{bmatrix}$$
$$\mathbf{v}_0 = 0.75(\mathbf{0}) - 1 \begin{bmatrix} 4 \\ -4 \end{bmatrix} = \begin{bmatrix} -4 \\ 4 \end{bmatrix}$$
$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{v}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -4 \\ 4 \end{bmatrix} = \begin{bmatrix} -4 \\ 4 \end{bmatrix}$$

   Iteration 2 ($k = 1$):

$$\nabla F(\mathbf{x}_1) = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} -4 \\ 4 \end{bmatrix} + \begin{bmatrix} 4 \\ -4 \end{bmatrix}$$
$$= \begin{bmatrix} -12 - 4 \\ 4 + 12 \end{bmatrix} + \begin{bmatrix} 4 \\ -4 \end{bmatrix} = \begin{bmatrix} -16 \\ 16 \end{bmatrix} + \begin{bmatrix} 4 \\ -4 \end{bmatrix} = \begin{bmatrix} -12 \\ 12 \end{bmatrix}$$
$$\mathbf{v}_1 = 0.75\mathbf{v}_0 - \alpha \nabla F(\mathbf{x}_1)$$
$$= 0.75 \begin{bmatrix} -4 \\ 4 \end{bmatrix} - 1 \begin{bmatrix} -12 \\ 12 \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \end{bmatrix} - \begin{bmatrix} -12 \\ 12 \end{bmatrix} = \begin{bmatrix} 9 \\ -9 \end{bmatrix}$$
$$\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{v}_1 = \begin{bmatrix} -4 \\ 4 \end{bmatrix} + \begin{bmatrix} 9 \\ -9 \end{bmatrix} = \begin{bmatrix} 5 \\ -5 \end{bmatrix}$$

2. **Stability with momentum.**

   First, we calculate the eigenvalues of the Hessian matrix $\mathbf{A}$. The characteristic equation is $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$:

$$\det \begin{bmatrix} 3 - \lambda & -1 \\ -1 & 3 - \lambda \end{bmatrix} = (3 - \lambda)^2 - 1 = 0$$
$$(3 - \lambda) = \pm 1 \implies \lambda_1 = 2, \quad \lambda_2 = 4 \implies \lambda_{\max} = 4$$

   The stability condition for momentum is $\alpha < \frac{2(1+\gamma)}{\lambda_{\max}}$. Substituting the values:

$$\alpha < \frac{2(1 + 0.75)}{4} = \frac{3.5}{4} = 0.875$$

   Since our learning rate $\alpha = 1$ is greater than 0.875, the algorithm is **unstable**.

3. **Stability without momentum ($\gamma = 0$).**

   If momentum is zero, the algorithm reverts to standard steepest descent. The stability condition is $\alpha < \frac{2}{\lambda_{\max}}$.

$$\alpha < \frac{2}{4} = 0.5$$

   Since $\alpha = 1 > 0.5$, the algorithm would still be **unstable** (even more so than with momentum).

Problem-07

1. **Is the max-pooling commutative with ReLU, i.e., maxPool[ReLU(x)] = ReLU[maxPool(x)]?**

   Answer: Yes.

   Max-pooling and ReLU are commutative. This property holds because the ReLU function, defined as $f(x) = \max(0, x)$, is a *monotonically non-decreasing* function. For any monotonically non-decreasing function $g(\cdot)$ and a set of inputs $S = \{x_1, x_2, \ldots, x_n\}$, the maximum of the transformed elements is equal to the transformation of the maximum element:

   $$\max(g(x_1), \ldots, g(x_n)) = g(\max(x_1, \ldots, x_n))$$

   Specifically for ReLU and a pooling window $\mathbf{x}$:

   $$\max(0, \max(\mathbf{x})) = \max(\{\max(0, x_i) \mid x_i \in \mathbf{x}\})$$

   Thus, the output is identical regardless of the order of operations.

2. **If yes, why should we choose to first apply max-pooling and then ReLU to a set of "pixels"?**

   Answer: Computational Efficiency.

   While the mathematical result is identical, the order of operations significantly impacts computational cost.

   - **Reduction of FLOPs:** Max-pooling performs downsampling, which reduces the spatial dimensions of the feature map. For instance, a standard 2×2 pooling layer reduces the number of elements (pixels) by a factor of 4.
   - **Optimization:** By applying max-pooling *first*, we discard a large portion of the data before applying the activation function. This means the ReLU operation is performed on significantly fewer elements (e.g., only 25% of the original input size for 2×2 pooling), saving processing time and memory bandwidth.

Problem-08

Parameter breakdown:

1. **LAYER-1 (3D Convolution):**
   - Input channels: 3
   - Output channels: 16
   - Kernel size: $5 \times 5 \times 5$
   - Weights: $3 \times 5^3 \times 16 = 6000$
2. **Batch Normalization after LAYER-1:**
   - Parameters per channel: 2 (γ and β)

- Total: $2 \times 16 = 32$

3. **LAYER-2 (3D Convolution):**
   - Input channels: 16
   - Output channels: 32
   - Kernel size: $5 \times 5 \times 5$
   - Weights: $16 \times 5^3 \times 32 = 64\,000$
4. **Batch Normalization after LAYER-2:**
   - Total: $2 \times 32 = 64$
5. **Flattening:** Output from LAYER-2 has 32 channels of size $1 \times 1 \times 1$, yielding 32 features.
6. **LAYER-4 (Dense layer):**
   - Input features: 32
   - Output units: 200
   - Weights: $32 \times 200 = 6400$
   - Biases: 200
7. **LAYER-5 (Dense layer):**
   - Input features: 200
   - Output units: 64
   - Weights: $200 \times 64 = 12\,800$
   - Biases: 64
8. **LAYER-6 (Output layer):**
   - Input features: 64
   - Output units: 1
   - Weights: $64 \times 1 = 64$
   - Bias: 1

Total parameters:

$$\begin{aligned} &6000 + 32 + 64\,000 + 64 + 6400 + 200 + 12\,800 + 64 + 64 + 1 \\ &= 89\,625. \end{aligned}$$

Therefore, the network has $\boxed{89625}$ trainable parameters.

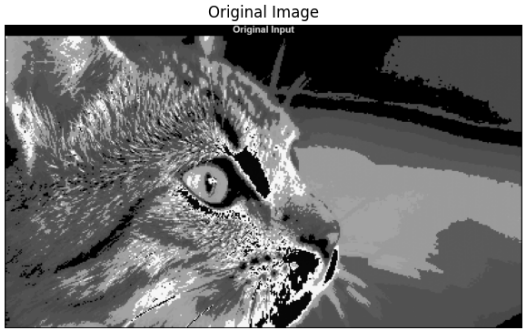
Problem-09

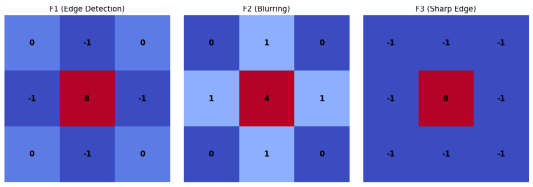
A

We are given three 3×3 filters: $F1$, $F2$, and $F3$. The convolution operation $C = I \otimes F$ slides these filters over the input image I using a stride of one and 'valid' padding.

Filter Visualization

The filters and the original image are visualized below:





(Part A.a & A.b)

The results of the convolutions are shown in Figure 4. Below is the theoretical analysis of each filter's function:

- **Filter F1 (Edge Detection / Laplacian Approximation):**

$$F1 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

  Function: This is a high-pass filter. It calculates the difference between the center pixel and its vertical/horizontal neighbors. **Effect:** It highlights edges (regions of rapid intensity change). Since the sum of weights is $8 - 4 = 4$, the output retains some brightness but emphasizes boundaries.

- **Filter F2 (Blurring / Low-Pass):**

$$F2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

  Function: This is a smoothing filter. All weights are positive, performing a weighted average of the neighborhood. **Effect:** It reduces noise and fine detail, resulting in a blurred image. Since the sum of weights is 8, the output image brightness increases significantly unless normalized.

- **Filter F3 (Sharpening / Strong Edge Detection):**

$$F3 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Function: This is a standard discrete Laplacian kernel including diagonal neighbors. **Effect:** It is a very aggressive edge detector. The sum of weights is $8-8=0$. This removes the DC component (average brightness), resulting in a dark image where only the edges appear as bright lines (as seen in the third panel of Figure 4).

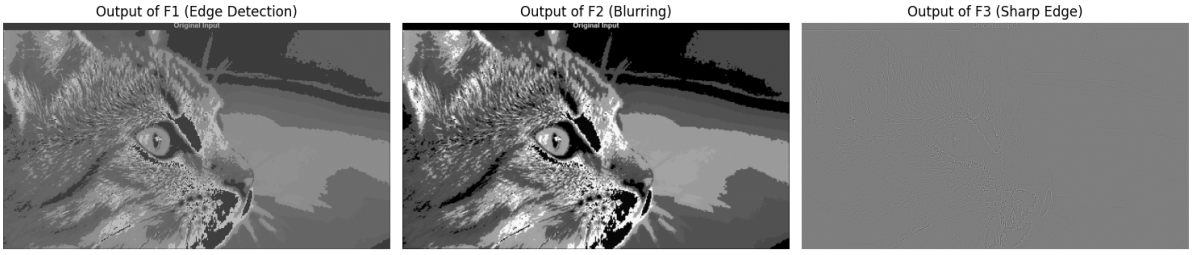


Figure 4: Convolution Results: C1 (F1), C2 (F2), and C3 (F3)

B. & C.

It is possible to compute the convolution $C = I \otimes F$ without sliding windows by transforming the operation into a matrix-vector multiplication:

$$\mathbf{y} = H\mathbf{x}$$

Where:

- $\mathbf{x}$ is the flattened input image vector (size $N^2 \times 1$).
- H is a **Doubly Block Toeplitz** transformation matrix (size roughly $N^2 \times N^2$).
- $\mathbf{y}$ is the flattened output image vector.

Implementation Logic

The construction of matrix H is sparse and follows a diagonal band structure. The pseudo-code for generating this matrix is provided below:

Algorithm 1 Pseudo-code for Toeplitz Matrix Construction

```
Input: Image_Shape (H_in, W_in), Kernel (K)
Output: Transformation Matrix H

Calculate Output_Shape (H_out, W_out)
Initialize H as zeros of size (H_out * W_out, H_in * W_in)

For i from 0 to H_out:
    For j from 0 to W_out:
        # Determine the row index in H (corresponds to one output pixel)
        Row_Index = i * W_out + j

        # Map kernel weights to input pixel columns
        For ki from 0 to Kernel_Height:
            For kj from 0 to Kernel_Width:
                # Calculate corresponding input pixel coordinates
                Input_Row = i + ki
                Input_Col = j + kj

                # Determine column index in H (flattened input index)
                Col_Index = Input_Row * W_in + Input_Col

                # Assign weight
                H[Row_Index, Col_Index] = Kernel[ki, kj]
```

The resulting structure of matrix H for a small example is visualized in Figure 5. Note the diagonal bands, which are characteristic of convolution operations.

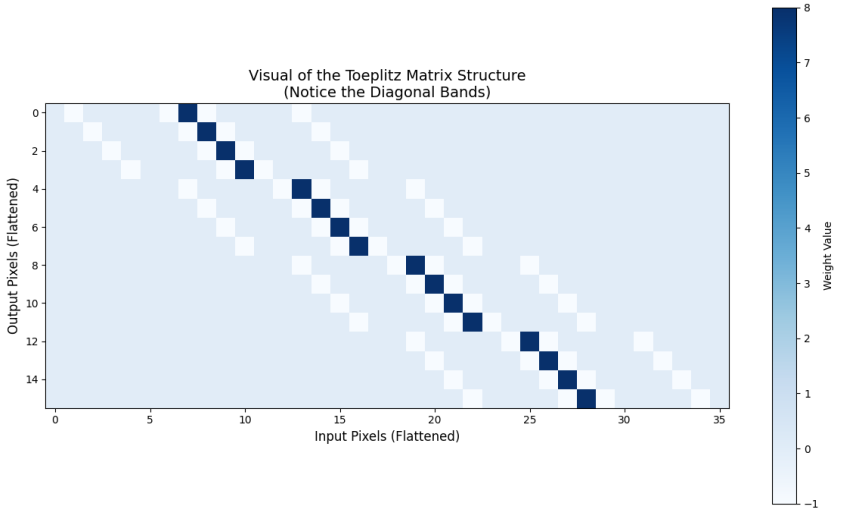


Figure 5: Visual of the Toeplitz Matrix Structure showing diagonal bands.

D.

We recorded the wall-clock time for both the standard convolution (Part A) and the matrix multiplication approach (Part C). Due to the extreme memory requirements of the matrix approach ($O(N^4)$ space for a dense matrix), the matrix method was tested on a downscaled version of the image and projected.

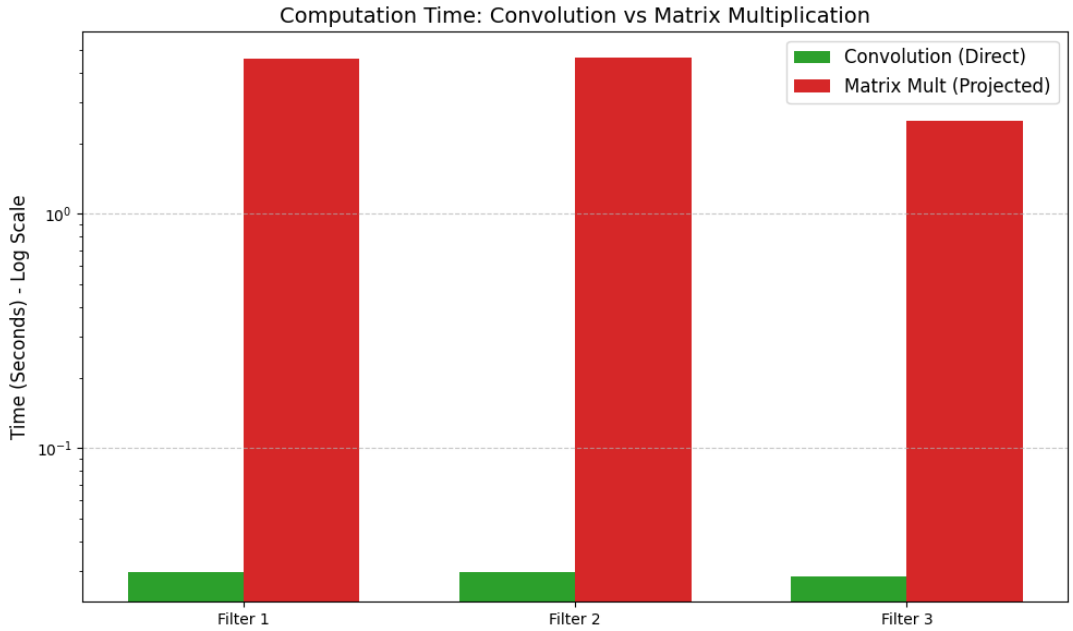


Figure 6: Computation Time: Direct Convolution vs. Matrix Multiplication (Log Scale).

Observations

1. **Speed:** As shown in Figure 6 (note the logarithmic scale), the **Direct Convolution is orders of magnitude faster**. The matrix multiplication approach requires constructing a massive matrix and performing a generic dot product, whereas the sliding window approach exploits the locality of the data and avoids processing zero-weights.

2. **Memory Demand:** The Matrix Multiplication approach is significantly **more memory demanding**. For a standard 512×512 image, the transformation matrix H would contain roughly $(512^2)^2 \approx 68$ billion entries. Even if stored sparsely, the overhead is massive compared to the minimal memory footprint of the sliding window kernel (3×3).

Conclusion: While mathematically elegant, the matrix multiplication method is computationally impractical for standard image processing tasks compared to optimized convolution implementations.

Problem 10

1.

Let h_1 and h_2 be two convolution kernels of sizes k_1 and k_2, respectively. For an input signal x, applying convolution with h_1 followed by h_2 yields:

$$y = (x * h_1) * h_2.$$

Since convolution is associative and commutative (for discrete, finite-length signals under appropriate boundary conditions), we have:

$$y = x * (h_1 * h_2).$$

Thus, the composition of the two convolutions is equivalent to a single convolution with kernel $h = h_1 * h_2$.

2.

Assuming 1D convolutions with stride 1 and no padding (valid convolution), the size of the equivalent kernel h is:

$$k = k_1 + k_2 - 1.$$

For 2D square kernels of sizes $k_1 \times k_1$ and $k_2 \times k_2$, the equivalent kernel size is:

$$k \times k \quad \text{with} \quad k = k_1 + k_2 - 1.$$

If the kernels are not square, the dimensions add similarly in each spatial direction.

3.

No, the converse is not true. Not every convolution kernel of size k can be decomposed into the convolution of two smaller kernels of sizes k_1 and k_2.

- **Algebraic perspective**: In 1D, this requires factoring the polynomial represented by the kernel into polynomials of lower degrees. This is not always possible over the reals (e.g., irreducible polynomials).
- **Spatial perspective**: In 2D, factorization is even more restricted; most kernels cannot be expressed as the convolution of two smaller kernels.

Thus, while any composition of two convolutions yields a single convolution, reverse decomposition is not always possible.

Problem 11

By inspecting the provided plot, we identify three key characteristics of the target function $f(x)$:

- **Odd Symmetry:** The function satisfies $f(-x) = -f(x)$. For an RBF network, this implies the bias b is zero, and the weights are antisymmetric ($w_2 = -w_1$).
- **Local Extrema:** The function exhibits a maximum of 0.5 at $x = -1.5$ and a minimum of -0.5 at $x = 1.5$.
- **Signal Localization:** The function amplitude is negligible at $x \in \{-1.0, -0.5, 0, 0.5, 1.0\}$. This indicates that the Gaussian basis functions must have a narrow bandwidth (σ) to prevent interference in these zero-magnitude regions.

Network Parameterization

The Radial Basis Function network is modeled as:

$$f(x) = \sum_{i=1}^{2} w_i \exp\left(-\frac{(x - c_i)^2}{2\sigma_i^2}\right) + b$$

Centers (c_i): We align the centers with the locations of the local extrema:

$$c_1 = -1.5, \quad c_2 = 1.5$$

Bias (b) and Weights (w_i): Due to the odd symmetry and the requirement that $f(0) = 0$, we set $b = 0$. Given the negligible overlap between basis functions, the peak amplitude is determined solely by the weight at that center.

- At $x = -1.5$, target is $0.5 \implies w_1 = 0.5$.
- At $x = 1.5$, target is $-0.5 \implies w_2 = -0.5$.

Bandwidth (σ): We require the activation to decay to near-zero at a distance of 0.5 from the center (e.g., at $x = -1.0$). We select a width of $\sigma = 0.15$. The resulting spread parameter is:

$$2\sigma^2 = 2(0.15)^2 = 0.045$$

To verify, we calculate the activation at the boundary $x = -1.0$:

$$\phi_1(-1.0) = \exp\left(-\frac{(-1.0 + 1.5)^2}{0.045}\right) = \exp(-5.55) \approx 0.0038$$

This confirms sufficient localization.

Final Model

Substituting the derived parameters, the analytical function is:

$$f(x) = 0.5 \exp\left(-\frac{(x + 1.5)^2}{0.045}\right) - 0.5 \exp\left(-\frac{(x - 1.5)^2}{0.045}\right)$$

Problem 12

This script implements the steepest descent algorithm for a Radial Basis Function network.

```
% Initial Parameters
w1 = 0; b1 = 1; % Hidden layer
w2 = -2; b2 = 1; % Output layer
alpha = 1; % Learning rate

% Training Set
p = [-1, 1];
t = [0, 1];

fprintf('Starting Steepest Descent\n\n');

for iter = 1:2
    fprintf('Iteration %d\n', iter);

    for i = 1:length(p)
        % Hidden layer output - Gaussian RBF
        n1 = (p(i) - w1) * b1;
        a1 = exp(-(n1^2));

        % Output layer output
        a2 = w2 * a1 + b2;

        % Error
        e = t(i) - a2;

        % Sensitivity of output layer
        s2 = -2 * e;

        % Sensitivity of hidden layer, chain rule
        % da1/dn1 = -2 * n1 * exp(-n1^2)
        s1 = s2 * w2 * (-2 * n1 * a1);

        % Output layer updates
        dw2 = s2 * a1;
        db2 = s2;

        % Hidden layer updates
        % Gradient for w1 involves d(n1)/dw1 = -b1
        % Gradient for b1 involves d(n1)/db1 = (p - w1)
        dw1 = s1 * (-b1);
        db1 = s1 * (p(i) - w1);

        w2 = w2 - alpha * dw2;
        b2 = b2 - alpha * db2;
        w1 = w1 - alpha * dw1;
        b1 = b1 - alpha * db1;

        fprintf('Point p=%d: w1=%.4f, b1=%.4f, w2=%.4f, b2=%.4f, Error
    =%.4f\n', ...
                p(i), w1, b1, w2, b2, e);
    end
    fprintf('\n');
end
```

Problem 13

A

$W = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & -1 \end{bmatrix}$ the matrix after transposition T.

*Each row i represents the coordinates of a neuron in the two-dimensional input space. Based on the Future Map part of the graph, neurons 1-4 are arranged in a 2×2 grid.

Initial weights:

- $w_1 = (0, 0)$
- $w_2 = (1, 0)$
- $w_3 = (1, 1)$
- $w_4 = (0, -1)$

From the Future Map grid, we distinguish that:

- Neuron 1 is directly connected to 2 and 3
- Neuron 2 is directly connected to 1 and 4
- Neuron 3 is directly connected to 1 and 4
- Neuron 4 is directly connected to 3 and 2

B

input: $p = [-1 \quad 1]^T$, 1 iteration:

Calculating the distance for each neuron $n_i = -\|w_i - p\|$

- $d_1^2 = (0 - (-1))^2 + (0 - 1)^2 = 1 + 1 = 2 \Rightarrow d_1 = \sqrt{2} \Rightarrow \underline{d_1 = 1.41}$
- $d_2^2 = (1 - (-1))^2 + (0 - 1)^2 = 4 + 1 = 5 \Rightarrow d_2 = \sqrt{5} \Rightarrow \underline{d_2 = 2.24}$
- $d_3^2 = (1 - (-1))^2 + (1 - 1)^2 = 4 + 0 = 4 \Rightarrow d_3 = \sqrt{4} \Rightarrow \underline{d_3 = 2}$
- $d_4^2 = (0 - (-1))^2 + (-1 - 1)^2 = 1 + 4 = 5 \Rightarrow d_4 = \sqrt{5} \Rightarrow \underline{d_4 = 2.24}$

The Winner neuron is the one with the smallest distance from the input p to the weights of the neuron. Therefore $d_1 = 1.41 \rightarrow$ Winner: neuron 1.

With neighborhood radius $R = 1$, in the 2×2 grid of the figure, since the winner is neuron 1, an update will also occur for the neighbors 2 & 3.

Since learning rate $a = 0.5 \rightarrow w(new) = w(old) + 0.5 \left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} - w(old) \right)$

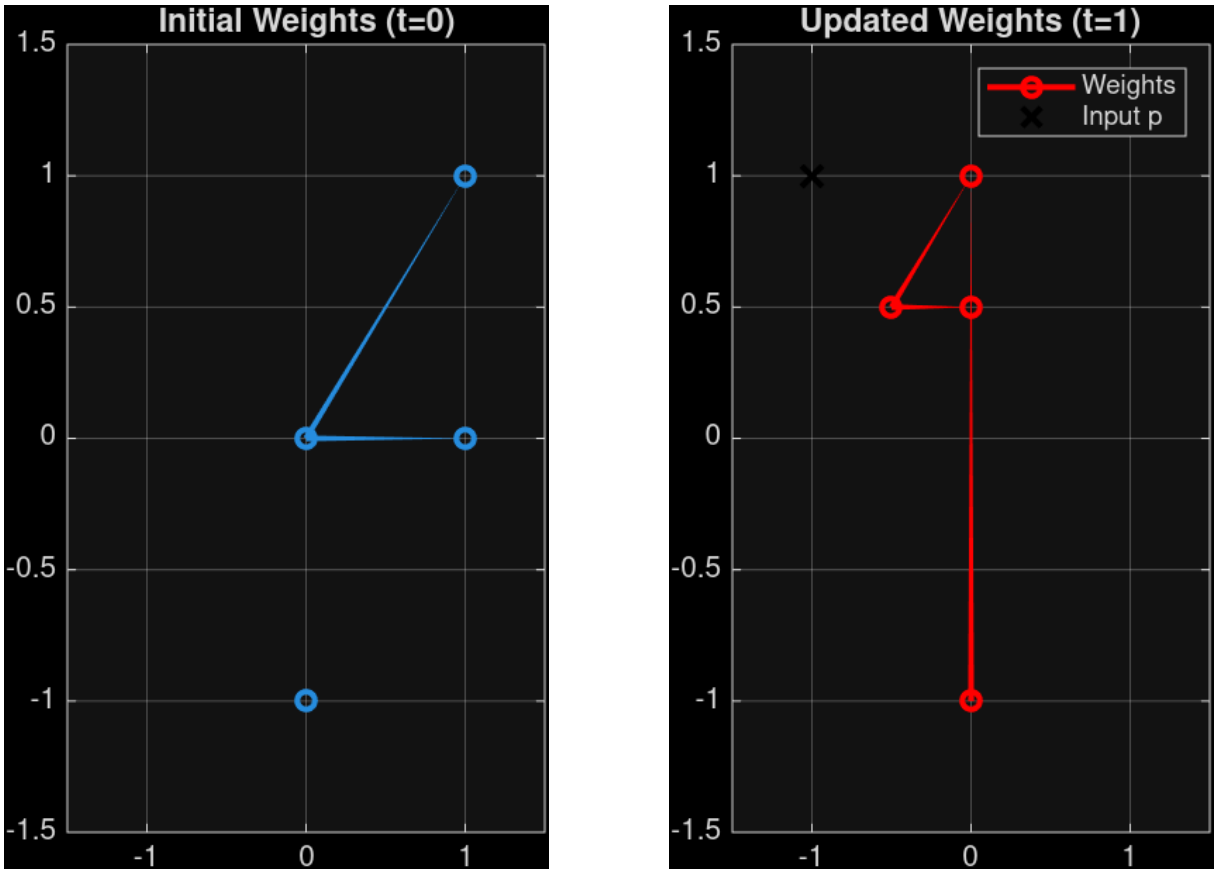
for w_1: $w_1^{new} = \begin{bmatrix}0\\0\end{bmatrix} + 0.5\left(\begin{bmatrix}-1\\1\end{bmatrix} - \begin{bmatrix}0\\0\end{bmatrix}\right) \Rightarrow w_1^{(new)} = \begin{bmatrix}-0.5\\0.5\end{bmatrix}$

for w_2: $w_2^{new} = \begin{bmatrix}1\\0\end{bmatrix} + 0.5\left(\begin{bmatrix}-1\\1\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}1\\0\end{bmatrix} + 0.5\begin{bmatrix}-2\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}-1\\0.5\end{bmatrix} \Rightarrow w_2^{(new)} = \begin{bmatrix}0\\0.5\end{bmatrix}$

for w_3: $w_3^{(new)} = \begin{bmatrix}1\\1\end{bmatrix} + 0.5\left(\begin{bmatrix}-1\\1\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix}\right) = \begin{bmatrix}1\\1\end{bmatrix} + 0.5\begin{bmatrix}-2\\0\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}-1\\0\end{bmatrix} \Rightarrow w_3^{(new)} = \begin{bmatrix}0\\1\end{bmatrix}$

C

$$W_{new} = \begin{bmatrix}w_1\\w_2\\w_3\\w_4\end{bmatrix}^T = \begin{bmatrix}-0.5 & 0.5\\0 & 0.5\\0 & 1\\0 & -1\end{bmatrix}^T \Rightarrow W_{new} = \begin{bmatrix}-0.5 & 0 & 0 & 0\\0.5 & 0.5 & 1 & -1\end{bmatrix}$$



(a) Initial Weights ($t = 0$) (b) Updated Weights ($t = 1$)

Figure 7: Visualization of the SOM weight update process for Problem 13.

They are all pushed towards p by a except for neuron 4 which was neither a winner nor a neighbor.

Problem 14

A

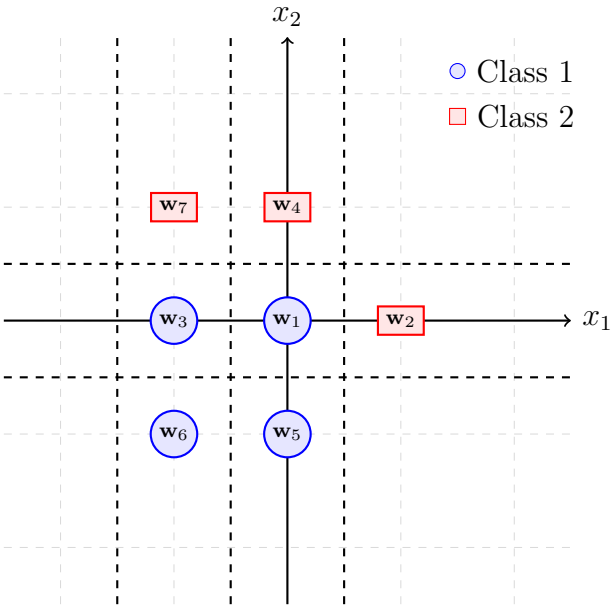
- **Number of Classes:** There are **2 classes**. This is determined by the number of rows in the second-layer weight matrix $\mathbf{W}^2$.
- **Number of Subclasses:** There are **7 subclasses**. This is determined by the number of columns in $\mathbf{W}^2$ (or rows in $(\mathbf{W}^1)^T$), which corresponds to the number of hidden neurons (prototypes).

B & C

The diagram below plots the first-layer weight vectors (subclasses).

- **Class 1** (Blue Circles): Vectors $\mathbf{w}_1, \mathbf{w}_3, \mathbf{w}_5, \mathbf{w}_6$.
- **Class 2** (Red Squares): Vectors $\mathbf{w}_2, \mathbf{w}_4, \mathbf{w}_7$.

The dashed lines represent the Voronoi decision boundaries separating the subclasses.



D

Given:

- Input vector: $\mathbf{p} = [1, 0.5]^T$ belonging to **Class 1**.
- Learning rate: $\alpha = 0.5$.

Winning Neuron

We calculate the squared Euclidean distance between $\mathbf{p}$ and the prototype vectors.

$$d(\mathbf{w}_1, \mathbf{p})^2 = (1-0)^2 + (0.5-0)^2 = 1.25$$
$$d(\mathbf{w}_2, \mathbf{p})^2 = (1-1)^2 + (0.5-0)^2 = \mathbf{0.25}$$

The winning neuron is **Subclass 2** ($\mathbf{w}_2 = [1, 0]^T$).

Classification Correctness

The winner $\mathbf{w}_2$ is associated with **Class 2**, but the input $\mathbf{p}$ is **Class 1**. This is an **incorrect classification**, so we move the weights away.

Update Weights

Formula: $\mathbf{w}_{new} = \mathbf{w}_{old} - \alpha(\mathbf{p} - \mathbf{w}_{old})$

$$\begin{aligned}\mathbf{w}_{new} &= \begin{bmatrix}1\\0\end{bmatrix} - 0.5\left(\begin{bmatrix}1\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right)\\ &= \begin{bmatrix}1\\0\end{bmatrix} - \begin{bmatrix}0\\0.25\end{bmatrix}\\ &= \begin{bmatrix}1\\-0.25\end{bmatrix}\end{aligned}$$

The new weight vector is $\mathbf{w}_2 = [1, -0.25]^T$.