HOMEWORK SET 3

ΓΑΛΑΝΗΣ ΑΧΙΛΛΕΑΣ ΑΛΕΞΑΝΔΡΟΣ ΒΑΣΙΛΕΙΟΣ - 02941 and ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ - 03074

Νευρο-Ασαφής Υπολογιστική 2023-24
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας, Βόλος
{acgalanis, kogalanis}@uth.gr

Περιεχόμενα

1 PROBLEM-01

From the problem statement, we deduct that the network will need to have two inputs and one output to distinguish these 2 classes. Everything that is inside the shaded circles will be class 1 and everything outside will be class 2. For simplicity, we will use only two neurons in the first layer (two basis functions), since this will be sufficient to solve our problem.

The rows of the first-layer weight matrix will craft the centers for the two basis functions. By centering a basis function in each of these 2 shaded regions, the first layer weight matrix will be like this:

$$W^1 = \begin{bmatrix} -1 & 1.5 \\ 2 & 2 \end{bmatrix}$$

The choice of the biases in the first layer depends on the width that we want for each basis function. By observing the plot,we deduct that the 1st basis function should be wider than the second. You can verify that by looking at the size of the shaded circles. Therefore, the first bias will be smaller than the second bias. The boundary formed by the first basis function should have a radius of 0.5, while the second basis function boundary should have a radius of 0.25 as seen by the plot. We want the basis functions to drop

significantly from their peaks in these distances. We try a bias of 3 for the first neuron and a bias of 6 for the second neuron:

$$a = e^{-\beta^2} = e^{-(3 \cdot 0.5)^2} = e^{-1.5} = 0.2231$$

$$a = e^{-\beta^2} = e^{-(6 \cdot 0.25)^2} = e^{-1.5} = 0.2231$$

This seems good so we will select first layer bias equal to:

$$b^1 = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

The original basis function response ranges from 0 to 1. We want the output to be negative for inputs outside the decision regions, so we will use a bias of -1 for the second layer:

$$b^2 = \begin{bmatrix} -1 \end{bmatrix}$$

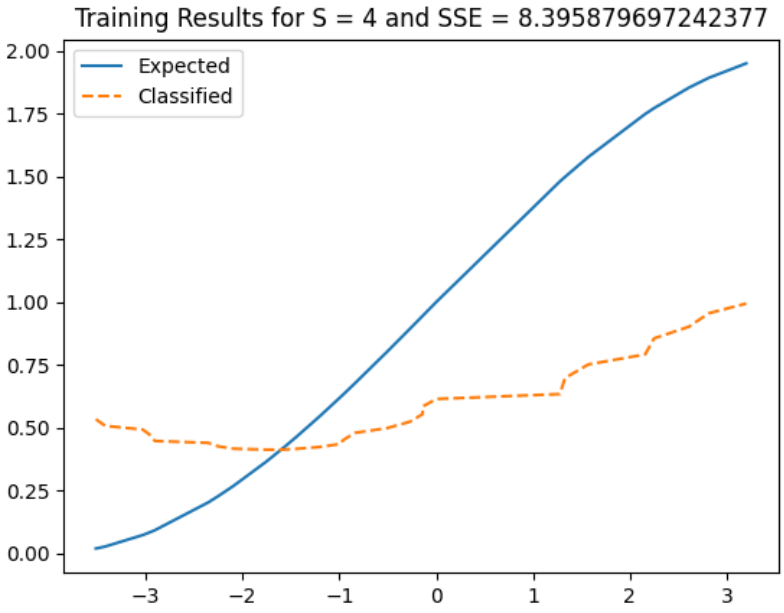
The weights in the second layer scale the height of the hills. We will use a value of 2 for the second layer weights, in order to bring the peaks up to 1:

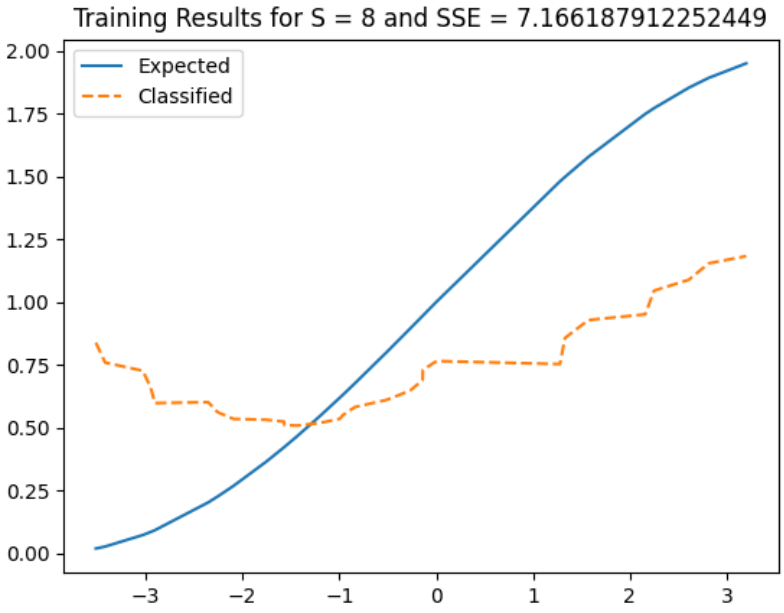
$$W^2 = \begin{bmatrix} 2 \ 2 \end{bmatrix}$$

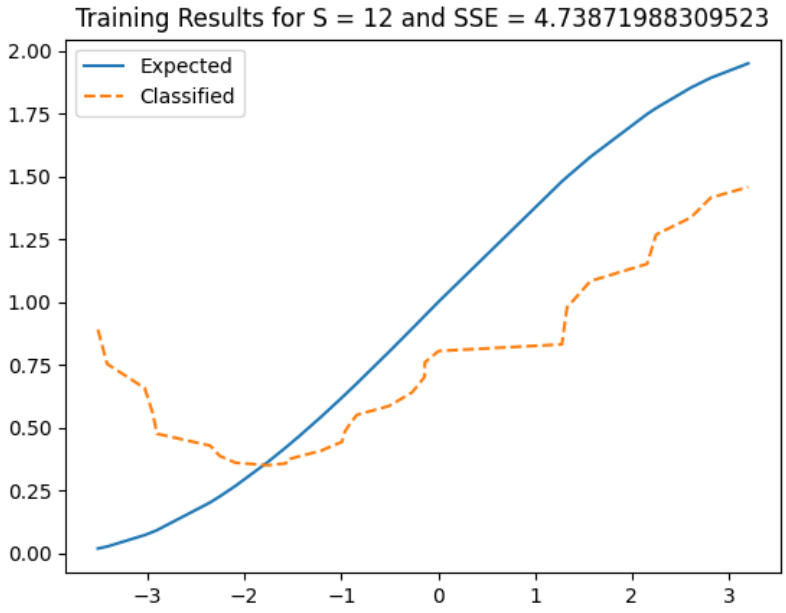
2 PROBLEM-02

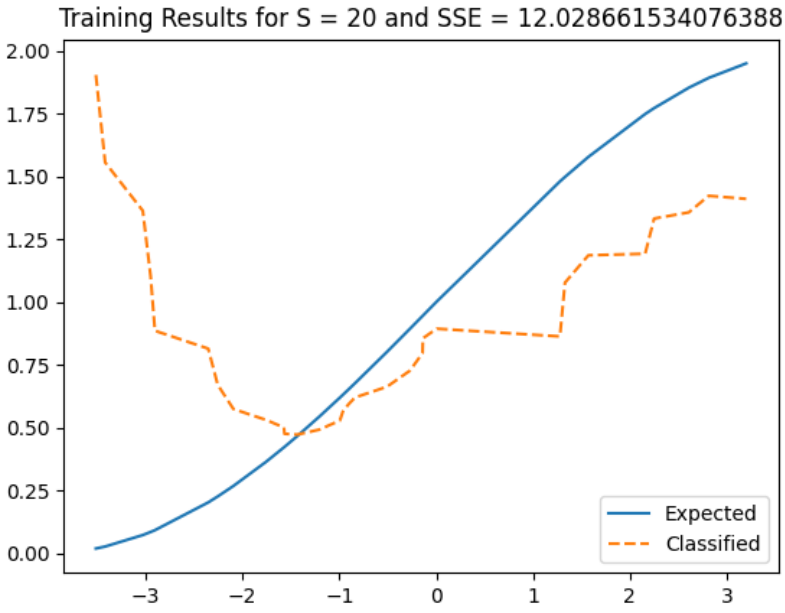
– For a = 0.01 :

4

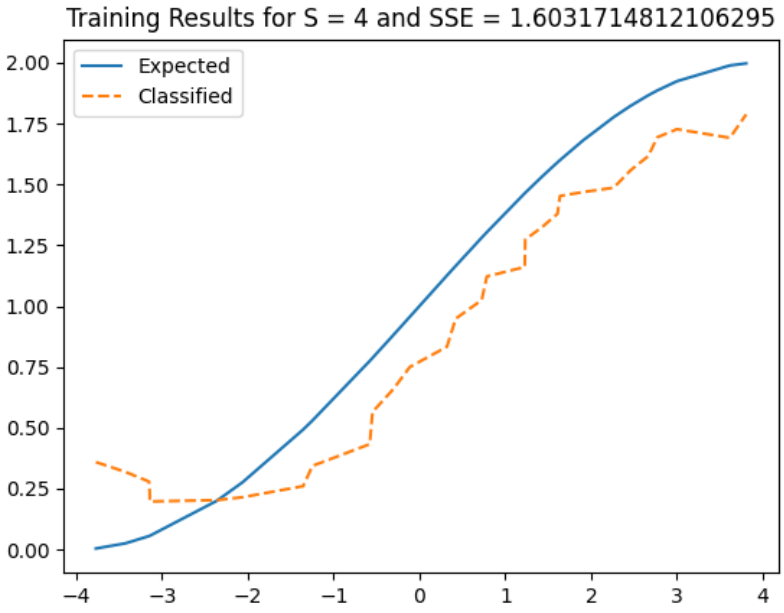

Training Results for S = 4 and SSE = 8.395879697242377
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3

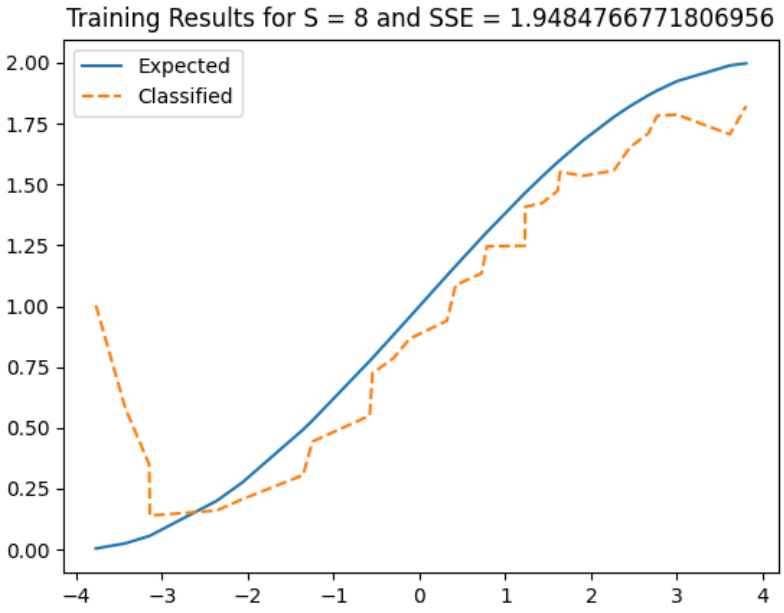

Training Results for S = 8 and SSE = 7.166187912252449
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3

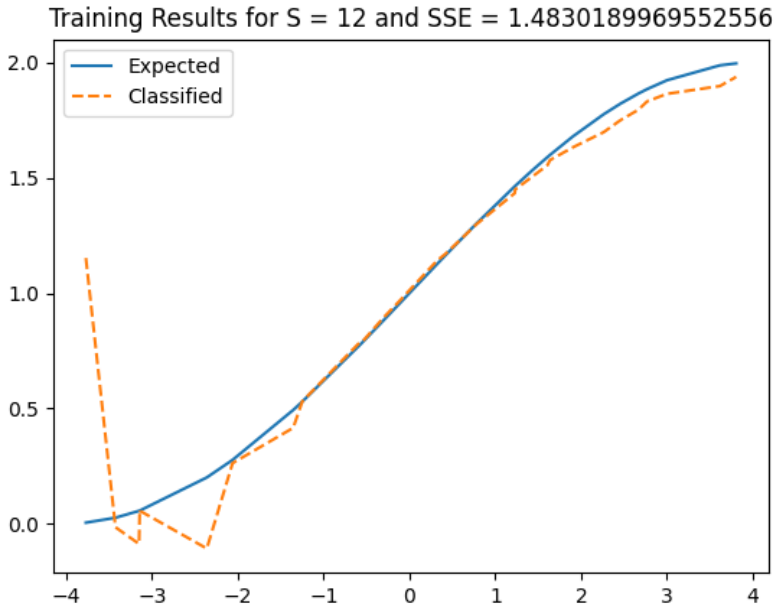

Training Results for S = 12 and SSE = 4.73871988309523
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3

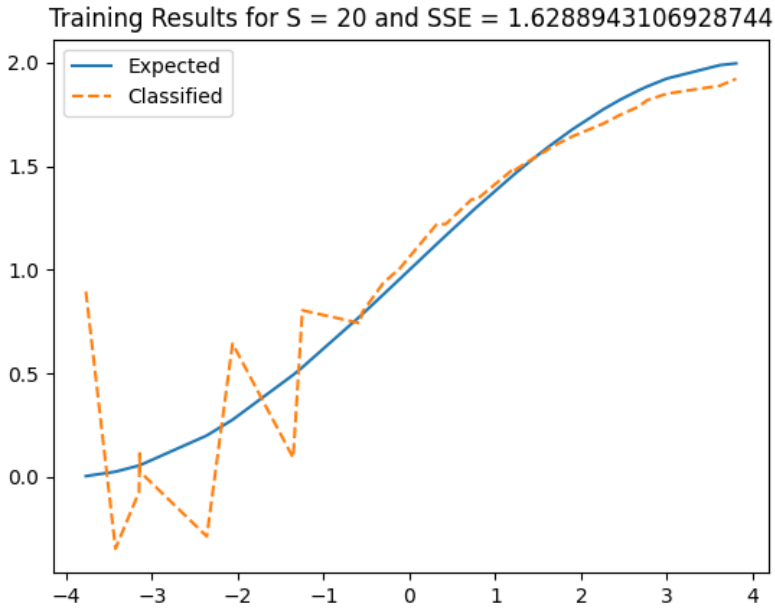

Training Results for S = 20 and SSE = 12.028661534076388
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3
Expected
Classified

– For $a = 0.05$:

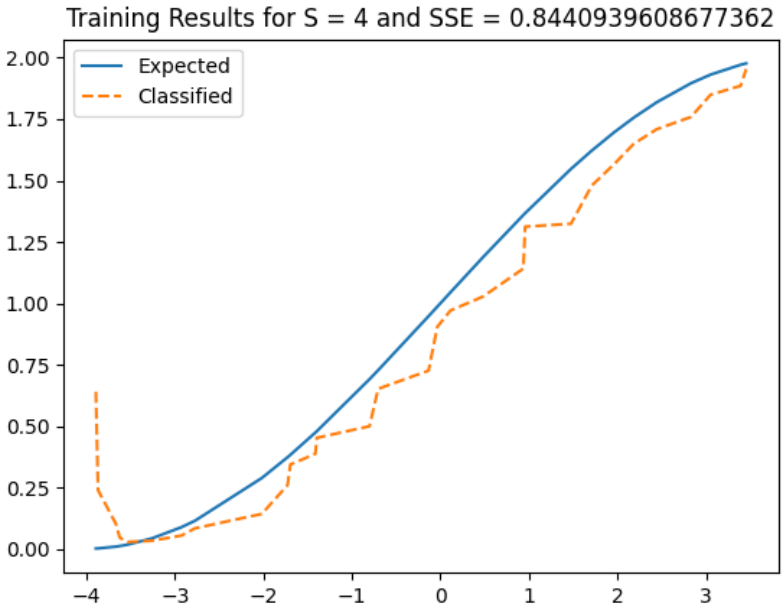

Training Results for S = 4 and SSE = 1.6031714812106295
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−4
−3
−2
−1
0
1
2
3
4

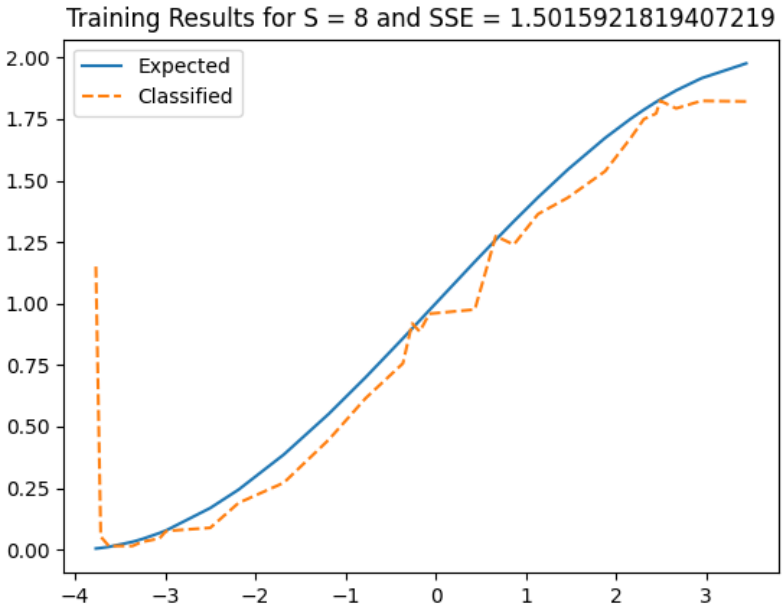

Training Results for S = 8 and SSE = 1.9484766771806956
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−4
−3
−2
−1
0
1
2
3
4

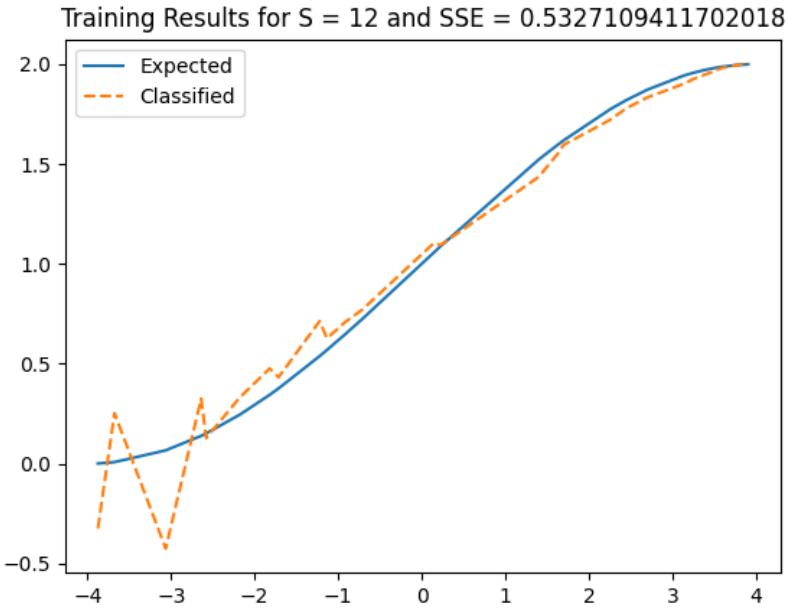

Training Results for S = 12 and SSE = 1.4830189969552556
Expected
Classified
2.0
1.5
1.0
0.5
0.0
−4
−3
−2
−1
0
1
2
3
4

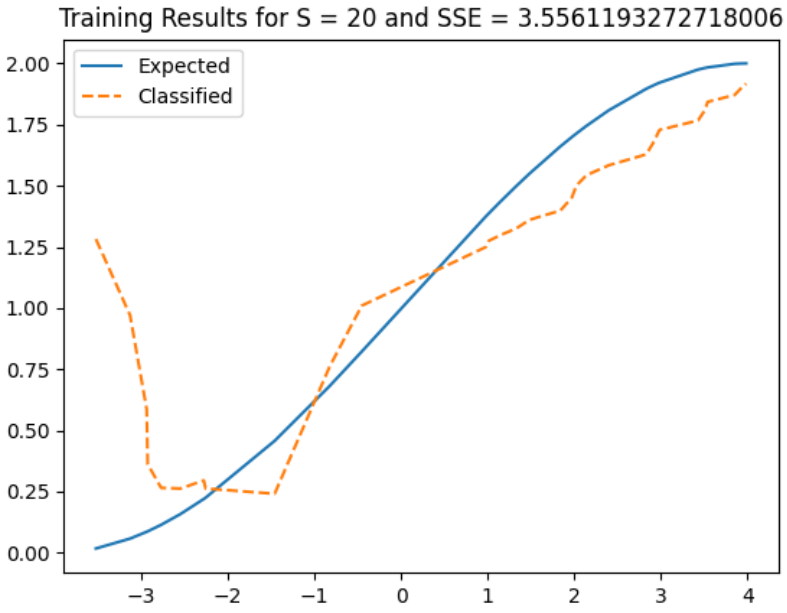

Training Results for S = 20 and SSE = 1.6288943106928744
Expected
Classified
2.0
1.5
1.0
0.5
0.0
−4
−3
−2
−1
0
1
2
3
4

– For $a = 0.1$:

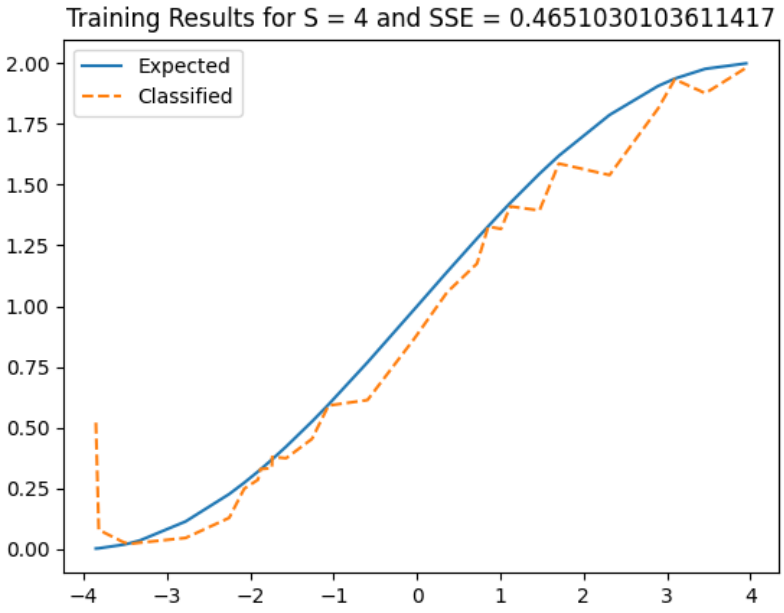

Training Results for S = 4 and SSE = 0.8440939608677362
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−4
−3
−2
−1
0
1
2
3

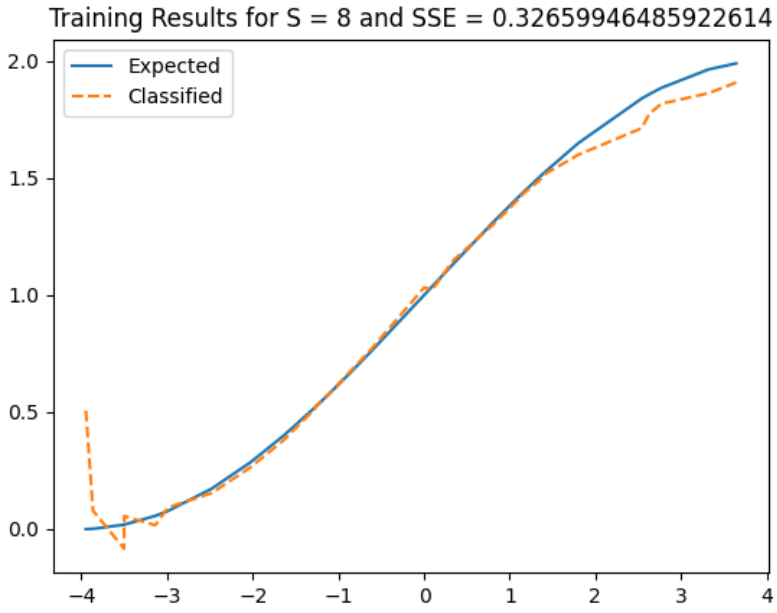

Training Results for S = 8 and SSE = 1.5015921819407219
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−4
−3
−2
−1
0
1
2
3

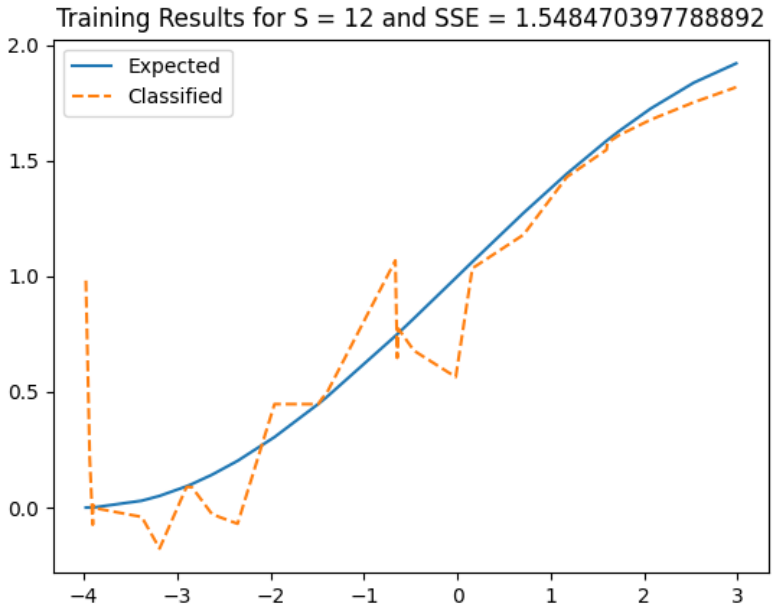

Training Results for S = 12 and SSE = 0.5327109411702018
Expected
Classified
2.0
1.5
1.0
0.5
0.0
−0.5
−4
−3
−2
−1
0
1
2
3
4

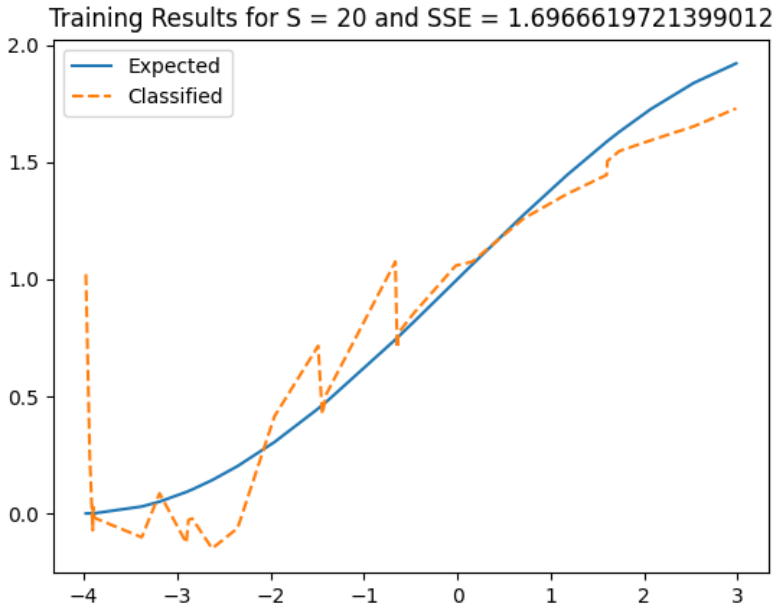

Training Results for S = 20 and SSE = 3.5561193272718006
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−3
−2
−1
0
1
2
3
4

– For $a = 0.2$:


Training Results for S = 4 and SSE = 0.4651030103611417
Expected
Classified
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−4
−3
−2
−1
0
1
2
3
4


Training Results for S = 8 and SSE = 0.32659946485922614
Expected
Classified
2.0
1.5
1.0
0.5
0.0
−4
−3
−2
−1
0
1
2
3
4


Training Results for S = 12 and SSE = 1.548470397788892
Expected
Classified
2.0
1.5
1.0
0.5
0.0
−4
−3
−2
−1
0
1
2
3



2.1 Observations

- For Lower Learning Rates the convergence appears to be better when the number of centers S is higher. Especially for lr = 0.05 the convergence is better for $S = 12$ or $S = 20$. This suggests that with a lower learning rate, the model requires more complexity (more centers) to capture the patterns in the data without overshooting the optimal solution during training.
- For lr = 0.1 the system seems to converge for every possible option of centers so it is the optimal learning rate.
- For Higher Learning Rates the convergence seems to be better when the number of centers is lower. Especially for lr = 0.2 the convergence is better for $S = 4$ or $S = 8$. A higher learning rate might be causing the model with more centers to converge too quickly, leading to suboptimal solutions or overshooting the minimum error. With fewer centers, the model has less complexity and is less likely to overfit, which might be why it benefits more from a higher learning rate.

The code for this problem is:

```
import matplotlib.pyplot as plt
from math import sin, pi, exp, sqrt
from random import uniform
import os

# Initialize input vectors
```

```
p = [uniform(-4.0, 4.0) for _ in range(30)]
p.sort()

learning_rate = 0.1

def g_function(p):
    return 1 + sin(p * (pi / 8))

def radial_basis(n):
    return exp(-n * n)

def purelin(n):
    return n

def purelin_derivative():
    return 1

def radbas_derivative(n):
    return -2 * n * exp(-n * n)

def initialize_weights(S):
    w1 = []
    b1 = []
    w2 = []
    w1 = [uniform(0, 0.3) for _ in range(S)]
    b1 = [uniform(0, 0.3) for _ in range(S)]
    w2 = [uniform(0, 0.3) for _ in range(S)]
    b2 = uniform(0, 0.1)
    return w1, b1, w2, b2

def input_propagation(p, S, w1, b1, w2, b2):
    n1 = []
    a1 = []
    n2 = b2
    for j in range(S):
        n = sqrt((p[i] - w1[j]) * (p[i] - w1[j])) * b1[j]
        n1.append(n)
        a = radial_basis(n)
        a1.append(a)
        n2 += a * w2[j]
    return n1, n2, a1

def calculate_error(p, a2, sum_sq_error):
    e = g_function(p[i]) - a2
    sum_sq_error += e ** 2
    return e, sum_sq_error

def backpropagation(e, a1, n1, S, w1, b1, w2, b2, learning_rate)
    :
    s2 = -2 * purelin_derivative() * (e)
```

```
    s1 = []
    for j in range(S):
        s1.append(radbas_derivative(n1[j]) * w2[j] * s2)
        w2[j] -= learning_rate * s2 * a1[j]

    b2 -= learning_rate * s2

    for j in range(S):
        w1[j] -= learning_rate * s1[j] * p[i]
        b1[j] -= learning_rate * s1[j]


S_values = [4, 8, 12, 20]

for S in S_values:
    print(f"\nFor S = {S}")
    w1, b1, w2, b2 = initialize_weights(S)

    epoch = 0  # Track the number of epochs

    # Start training
    while True:
        sum_sq_error = 0
        result = []
        g = []
        for i in range(30):
            n1, n2, a1 = input_propagation(p, S, w1, b1, w2, b2)

            a2 = purelin(n2)
            result.append(a2)
            g.append(g_function(p[i]))
            print(f"P{i}: Classified as {a2}, Expected {
    g_function(p[i])}")

            # Calculate error
            e, sum_sq_error = calculate_error(p, a2,
    sum_sq_error)

            # Calculate sensitivities and recalculate weights
    and biases
            backpropagation(e, a1, n1, S, w1, b1, w2, b2,
    learning_rate)

        print(f"Epoch {epoch}: Sum Sq Error = {sum_sq_error}")
        epoch += 1  # Increment epoch count

        # Check sum square error threshold
        if sum_sq_error <= 2:
            print("Training completed.")
            break
```

```
print(f"Final weight1: {w1} and bias1: {b1}")
print(f"Final weight2: {w2} and bias2: {b2}")
# Design plot
plt.plot(p, g, label="Expected")
plt.plot(p, result, label="Classified", linestyle='--')
plt.legend()
plt.title(f"Training Results for S = {S}")  # Optional: Add
a title to the plot
# Save the plot
plot_filename = os.path.join(f"plot_S_{S}.png")
plt.savefig(plot_filename)
print(f"Plot saved as {plot_filename}")  # Print the
filename of the saved plot
plt.show()
```

3 PROBLEM-03

We have 2 classes because we have a two-neuron competitive layer with 2 initial weights, and 1 subclass for each of the classes (one winning neuron). We begin by creating W^2:

$$W^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This means that it connects hidden neuron 1 to output neuron 1 and it connects hidden neuron 2 to output neuron 2.

– For p_1:

$$n^1 = -\begin{bmatrix} \|w_1 - p_1\| \\ \|w_2 - p_1\| \end{bmatrix} = -\begin{bmatrix} \left\| \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\| \\ \left\| \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\| \end{bmatrix} = -\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix} -1 \\ -1 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_1. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This output indicates that p_1 is a member of class 1. So W_1 is updated by moving it toward p_1:

$$W_1(1) = W_1(0) + \alpha \cdot (p_1 - W_1(0)) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$$

– For p_2:

$$n^1 = -\begin{bmatrix}\|w_1 - p_2\| \\ \|w_2 - p_2\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}0.5 \\ 1\end{bmatrix} - \begin{bmatrix}1 \\ 2\end{bmatrix}\right\| \\ \left\|\begin{bmatrix}1 \\ 0\end{bmatrix} - \begin{bmatrix}1 \\ 2\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}1.8027 \\ 2.8284\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-1.8027 \\ -2.8284\end{bmatrix}) = \begin{bmatrix}1 \\ 0\end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_2. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}1 \\ 0\end{bmatrix} = \begin{bmatrix}1 \\ 0\end{bmatrix}$$

This output indicates that p_2 is a member of class 1. So W_1 is updated by moving it toward p_2:

$$W_1(2) = W_1(1) + \alpha \cdot (p_2 - W_1(1)) = \begin{bmatrix}0.5 \\ 1\end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix}-1 \\ 2\end{bmatrix} - \begin{bmatrix}0.5 \\ 1\end{bmatrix}\right) = \begin{bmatrix}0.5 \\ 1\end{bmatrix} + \begin{bmatrix}-0.75 \\ 0.5\end{bmatrix} = \begin{bmatrix}-0.25 \\ 1.5\end{bmatrix}$$

– For p_3:

$$n^1 = -\begin{bmatrix}\|w_1 - p_3\| \\ \|w_2 - p_3\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}-0.25 \\ 1.5\end{bmatrix} - \begin{bmatrix}-2 \\ -2\end{bmatrix}\right\| \\ \left\|\begin{bmatrix}1 \\ 0\end{bmatrix} - \begin{bmatrix}-2 \\ -2\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}3.9131 \\ 3.6055\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-3.9131 \\ -3.6055\end{bmatrix}) = \begin{bmatrix}0 \\ 1\end{bmatrix}$$

The second hidden neuron has the closest weight vector to p_3. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}0 \\ 1\end{bmatrix} = \begin{bmatrix}0 \\ 1\end{bmatrix}$$

This output indicates that p_3 is a member of class 2. So W_2 is updated by moving it toward p_3:

$$W_2(1) = W_2(0) + \alpha \cdot (p_3 - W_2(0)) = \begin{bmatrix}1 \\ 0\end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix}-2 \\ -2\end{bmatrix} - \begin{bmatrix}1 \\ 0\end{bmatrix}\right) = \begin{bmatrix}1 \\ 0\end{bmatrix} + \begin{bmatrix}-1.5 \\ -1\end{bmatrix} = \begin{bmatrix}-0.5 \\ -1\end{bmatrix}$$

– For p_2:

$$n^1 = -\begin{bmatrix}\|w_1 - p_2\| \\ \|w_2 - p_2\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}-0.25 \\ 1.5\end{bmatrix} - \begin{bmatrix}-1 \\ 2\end{bmatrix}\right\| \\ \left\|\begin{bmatrix}-0.5 \\ -1\end{bmatrix} - \begin{bmatrix}-1 \\ 2\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}0.9013 \\ 3.0413\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-0.9013 \\ -3.0413\end{bmatrix}) = \begin{bmatrix}1 \\ 0\end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_2. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}1 \\ 0\end{bmatrix} = \begin{bmatrix}1 \\ 0\end{bmatrix}$$

This output indicates that p_3 is a member of class 1. So W_1 is updated by moving it toward p_2:

$$W_1(3) = W_1(2) + \alpha \cdot (p_2 - W_1(2)) = \begin{bmatrix}-0.25 \\ 1.5\end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix}-1 \\ 2\end{bmatrix} - \begin{bmatrix}-0.25 \\ 1.5\end{bmatrix}\right) = \begin{bmatrix}-0.25 \\ 1.5\end{bmatrix} + \begin{bmatrix}-0.375 \\ 0.25\end{bmatrix} = \begin{bmatrix}-0.625 \\ 1.75\end{bmatrix}$$

– For p_3:

$$n^1 = -\begin{bmatrix}\|w_1 - p_3\| \\ \|w_2 - p_3\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}-0.625 \\ 1.75\end{bmatrix} - \begin{bmatrix}-2 \\ -2\end{bmatrix}\right\| \\ \left\|\begin{bmatrix}-0.5 \\ -1\end{bmatrix} - \begin{bmatrix}-2 \\ -2\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}3.9941 \\ 1.8027\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-3.9941 \\ -1.8027\end{bmatrix}) = \begin{bmatrix}0 \\ 1\end{bmatrix}$$

The second hidden neuron has the closest weight vector to p_3. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}0 \\ 1\end{bmatrix} = \begin{bmatrix}0 \\ 1\end{bmatrix}$$

This output indicates that p_3 is a member of class 2. So W_2 is updated by moving it toward p_3:

$$W_2(2) = W_2(1) + \alpha \cdot (p_3 - W_2(1)) = \begin{bmatrix}-0.5 \\ -1\end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix}-2 \\ -2\end{bmatrix} - \begin{bmatrix}-0.5 \\ -1\end{bmatrix}\right) = \begin{bmatrix}-0.5 \\ -1\end{bmatrix} + \begin{bmatrix}-0.75 \\ -0.5\end{bmatrix} = \begin{bmatrix}-1.25 \\ -1.5\end{bmatrix}$$

– For p_1:

$$n^1 = -\begin{bmatrix}\|w_1 - p_1\| \\ \|w_2 - p_1\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}-0.625 \\ 1.75\end{bmatrix} - \begin{bmatrix}1 \\ 1\end{bmatrix}\right\| \\ \left\|\begin{bmatrix}-1.25 \\ -1.5\end{bmatrix} - \begin{bmatrix}1 \\ 1\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}1.7897 \\ 3.3634\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-1.7897\\-3.3634\end{bmatrix}) = \begin{bmatrix}1\\0\end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_1. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$$

This output indicates that p_1 is a member of class 1. So W_1 is updated by moving it toward p_1:

$$W_1(4) = W_1(3) + \alpha \cdot (p_1 - W_2(3)) = \begin{bmatrix}-0.625\\1.75\end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}-0.625\\1.75\end{bmatrix} \right) = \begin{bmatrix}-0.625\\1.75\end{bmatrix} + \begin{bmatrix}0.8125\\-0.375\end{bmatrix} = \begin{bmatrix}0.1875\\1.375\end{bmatrix}$$

```
import numpy as np

def euclidean_distance(w, p):
  return np.linalg.norm(w - p)

def competitive(distances):
  return np.array([1 if d == max(distances) else 0 for d in
    distances])

# Initialize input vectors and targets
p1 = np.array([1, 1])

p2 = np.array([-1, 2])

p3 = np.array([-2, -2])

# Initialize weights
W1 = np.array([0, 1])
W2 = np.array([1, 0])

alpha = 0.5
mat = np.eye(2)

for i, p in enumerate([p1, p2, p3, p2, p3, p1], start=1):
  # Compute distances from the weight vectors
  distances = -np.array([euclidean_distance(W1, p),
    euclidean_distance(W2, p)])

  # Calulate a1
  a1 = competitive(distances)

  # Calulate a2
  a2 = np.dot(mat, a1.reshape(-1, 1))
```

```
# Update the weights for the winning neuron
if a2[0] == 1:
  W1 = W1 + alpha * (p - W1)
else:
  W2 = W2 + alpha * (p - W2)

print(f"After input {i}:")
print(f"W1 = {W1}")
print(f"W2 = {W2}\n")
```

4 PROBLEM-04

Just like problem 3, we have 2 classes because we have a two-neuron competitive layer with 2 initial weights, and 1 subclass for each of the classes (one winning neuron). We begin by creating W^2:

$$W^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

– For p_1:

$$n^1 = -\begin{bmatrix} \|w_1 - p_1\| \\ \|w_2 - p_1\| \end{bmatrix} = -\begin{bmatrix} \left\| \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\| \\ \left\| \begin{bmatrix} -1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\| \end{bmatrix} = -\begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix} -1 \\ -3 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_1. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This output indicates that p_1 is a member of class 1. So W_1 is updated by moving it toward p_1:

$$W_1(1) = W_1(0) + \alpha \cdot (p_1 - W_1(0)) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix} 2 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 0 \end{bmatrix}$$

– For p_2:

$$n^1 = -\begin{bmatrix} \|w_1 - p_2\| \\ \|w_2 - p_2\| \end{bmatrix} = -\begin{bmatrix} \left\| \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\| \\ \left\| \begin{bmatrix} -1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\| \end{bmatrix} = -\begin{bmatrix} 1.8027 \\ 1.414 \end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-1.8027\\-1.414\end{bmatrix}) = \begin{bmatrix}0\\1\end{bmatrix}$$

The second hidden neuron has the closest weight vector to p_2. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1\ 0\\0\ 1\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$$

This output indicates that p_2 is a member of class 2. So W_2 is updated by moving it towards p_2:

$$W_2(1) = W_2(0)+\alpha\cdot(p_2-W_2(0)) = \begin{bmatrix}-1\\0\end{bmatrix}+0.5\cdot\left(\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}-1\\0\end{bmatrix}\right) = \begin{bmatrix}-1\\0\end{bmatrix}+\begin{bmatrix}0.5\\0.5\end{bmatrix} = \begin{bmatrix}-0.5\\0.5\end{bmatrix}$$

– For p_3:

$$n^1 = -\begin{bmatrix}\|w_1 - p_3\|\\\|w_2 - p_3\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}1.5\\0\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix}\right\|\\\left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}2.0615\\2.9154\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-2.0615\\-2.9154\end{bmatrix}) = \begin{bmatrix}1\\0\end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_3. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix}1\ 0\\0\ 1\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$$

This output indicates that p_3 is a member of class 1. So W_1 is updated by moving it towards p_3:

$$W_1(2) = W_1(1)+\alpha\cdot(p_3-W_1(1)) = \begin{bmatrix}1.5\\0\end{bmatrix}+0.5\cdot\left(\begin{bmatrix}2\\2\end{bmatrix} - \begin{bmatrix}1.5\\0\end{bmatrix}\right) = \begin{bmatrix}1.5\\0\end{bmatrix}+\begin{bmatrix}0.25\\1\end{bmatrix} = \begin{bmatrix}1.75\\1\end{bmatrix}$$

– For p_2:

$$n^1 = -\begin{bmatrix}\|w_1 - p_2\|\\\|w_2 - p_2\|\end{bmatrix} = -\begin{bmatrix}\left\|\begin{bmatrix}1.75\\1\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|\\\left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|\end{bmatrix} = -\begin{bmatrix}1.75\\0.7071\end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix}-1.75\\-0.7071\end{bmatrix}) = \begin{bmatrix}0\\1\end{bmatrix}$$

The second hidden neuron has the closest weight vector to p_2. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

This output indicates that p_3 is a member of class 2. So W_2 is updated by moving it towards p_2:

$$W_2(2) = W_2(1) + \alpha \cdot (p_2 - W_2(1)) = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \right) = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 0.25 \\ 0.25 \end{bmatrix} = \begin{bmatrix} -0.25 \\ 0.75 \end{bmatrix}$$

– For p_3:

$$n^1 = - \begin{bmatrix} \|w_1 - p_3\| \\ \|w_2 - p_3\| \end{bmatrix} = - \begin{bmatrix} \left\| \begin{bmatrix} 1.75 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} \right\| \\ \left\| \begin{bmatrix} -0.25 \\ 0.75 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} \right\| \end{bmatrix} = - \begin{bmatrix} 1.0307 \\ 2.5739 \end{bmatrix}$$

We calculate a^1:

$$a^1 = compet(\begin{bmatrix} -1.0307 \\ -2.5739 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_3. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This output indicates that p_3 is a member of class 1. So W_1 is updated by moving it towards p_3:

$$W_1(3) = W_1(2) + \alpha \cdot (p_3 - W_1(2)) = \begin{bmatrix} 1.75 \\ 1 \end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix} 2 \\ 2 \end{bmatrix} - \begin{bmatrix} 1.75 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 1.75 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.125 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 1.875 \\ 1.5 \end{bmatrix}$$

– For p_1:

$$n^1 = - \begin{bmatrix} \|w_1 - p_1\| \\ \|w_2 - p_1\| \end{bmatrix} = - \begin{bmatrix} \left\| \begin{bmatrix} 1.875 \\ 1.5 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\| \\ \left\| \begin{bmatrix} -0.25 \\ 0.75 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\| \end{bmatrix} = - \begin{bmatrix} 1.5051 \\ 2.3717 \end{bmatrix}$$

We calculate a^1:

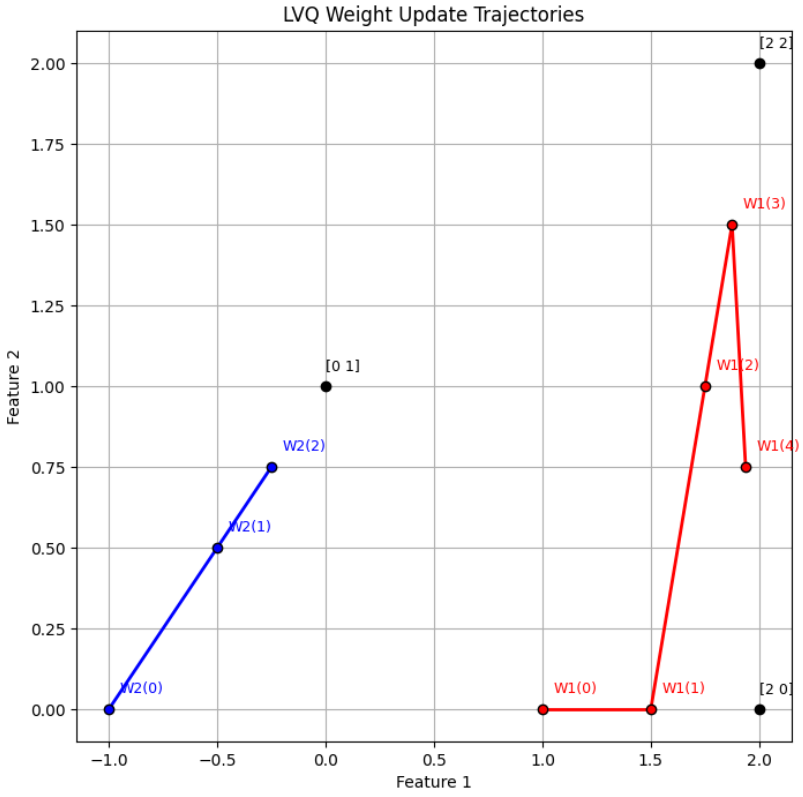
$$a^1 = compet(\begin{bmatrix} -1.5051 \\ -2.3717 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The first hidden neuron has the closest weight vector to p_1. We calculate a^2:

$$a^2 = W^2 \cdot a^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This output indicates that p_1 is a member of class 1. So W_1 is updated by moving it toward p_1:

$$W_1(4) = W_1(3) + \alpha \cdot (p_1 - W_1(3)) = \begin{bmatrix} 1.875 \\ 1.5 \end{bmatrix} + 0.5 \cdot \left(\begin{bmatrix} 2 \\ 0 \end{bmatrix} - \begin{bmatrix} 1.875 \\ 1.5 \end{bmatrix} \right) = \begin{bmatrix} 1.875 \\ 1.5 \end{bmatrix} + \begin{bmatrix} 0.0625 \\ -0.75 \end{bmatrix} = \begin{bmatrix} 1.9375 \\ 0.75 \end{bmatrix}$$



We observe that after many iterations vectors p1 and p3 will belong to the same cluster (class) and that vector p2 will belong in a different cluster (class).

```
import numpy as np
import matplotlib.pyplot as plt

def euclidean_distance(w, p):
```

```
  return np.linalg.norm(w - p)

def competitive(distances):
  return np.array([1 if d == max(distances) else 0 for d in
    distances])

# Initialize input vectors and targets
p1 = np.array([2, 0])

p2 = np.array([0, 1])

p3 = np.array([2, 2])

# Initialize weights
W1 = np.array([1, 0])
W2 = np.array([-1, 0])

alpha = 0.5
mat = np.eye(2)

plt.figure(figsize=(8, 8))

# Plot the inputs
for i, p in enumerate([p1, p2, p3]):
  plt.scatter(*p, color='black', zorder=5)
  plt.text(p[0], p[1] + 0.05, f'{p}', color='black', fontsize=9)

# Store the initial weights for trajectory plotting
W1_history = [W1.copy()]
W2_history = [W2.copy()]

for i, p in enumerate([p1, p2, p3, p2, p3, p1], start=1):
  # Compute distances from the weight vectors
  distances = -np.array([euclidean_distance(W1, p),
    euclidean_distance(W2, p)])

  # Calulate a1
  a1 = competitive(distances)

  # Calulate a2
  a2 = np.dot(mat, a1.reshape(-1, 1))

  # Store the old weights for plotting
  W1_old, W2_old = W1.copy(), W2.copy()

  # Update the weights for the winning neuron
  if a2[0] == 1:
    W1 = W1 + alpha * (p - W1)
    W1_history.append(W1.copy())
  else:
```

```
    W2 = W2 + alpha * (p - W2)
    W2_history.append(W2.copy())

  # Print the input, old and new weights
  print(f"Iteration {i}:")
  print(f"Old W1: {W1_old}, New W1: {W1}")
  print(f"Old W2: {W2_old}, New W2: {W2}\n")

# Plot the starting positions of the weights
plt.scatter(*W1_history[0], color='red', edgecolor='k', zorder
    =5)
plt.text(*W1_history[0] + 0.05, 'W1(0)', color='red', fontsize
    =9)

plt.scatter(*W2_history[0], color='blue', edgecolor='k', zorder
    =5)
plt.text(*W2_history[0] + 0.05, 'W2(0)', color='blue', fontsize
    =9)

# Plot the trajectories for W1 and W2
for i in range(1, 6):
  if i < len(W1_history):
    plt.plot(*zip(W1_history[i-1], W1_history[i]), color='red',
    linestyle='-', linewidth=2, zorder=3)
    plt.scatter(*W1_history[i], color='red', edgecolor='k',
    zorder=5)
    plt.text(*W1_history[i] + 0.05, f'W1({i})', color='red',
    fontsize=9)

  if i < len(W2_history):
    plt.plot(*zip(W2_history[i-1], W2_history[i]), color='blue',
     linestyle='-', linewidth=2, zorder=3)
    plt.scatter(*W2_history[i], color='blue', edgecolor='k',
    zorder=5)
    plt.text(*W2_history[i] + 0.05, f'W2({i})', color='blue',
    fontsize=9)

plt.title('LVQ Weight Update Trajectories')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()
```

5 PROBLEM-05

Code for this problem:

```
import numpy as np
```

```
from statsmodels.regression.linear_model import yule_walker

def sigmoid(x):
    return 1. / (1 + np.exp(-x))

def sigmoid_derivative(values):
    return values*(1-values)

def tanh_derivative(values):
    return 1. - values ** 2

# createst uniform random array w/ values in [a,b) and shape
    args
def rand_arr(a, b, *args):
    np.random.seed(0)
    return np.random.rand(*args) * (b - a) + a

class LstmParam:
    def __init__(self, mem_cell_ct, x_dim):
        self.mem_cell_ct = mem_cell_ct
        self.x_dim = x_dim
        concat_len = x_dim + mem_cell_ct
        # weight matrices
        self.wg = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wi = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wf = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wo = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        # bias terms
        self.bg = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bi = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bf = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bo = rand_arr(-0.1, 0.1, mem_cell_ct)
        # diffs (derivative of loss function w.r.t. all
    parameters)
        self.wg_diff = np.zeros((mem_cell_ct, concat_len))
        self.wi_diff = np.zeros((mem_cell_ct, concat_len))
        self.wf_diff = np.zeros((mem_cell_ct, concat_len))
        self.wo_diff = np.zeros((mem_cell_ct, concat_len))
        self.bg_diff = np.zeros(mem_cell_ct)
        self.bi_diff = np.zeros(mem_cell_ct)
        self.bf_diff = np.zeros(mem_cell_ct)
        self.bo_diff = np.zeros(mem_cell_ct)

    def apply_diff(self, lr = 1):
        self.wg -= lr * self.wg_diff
        self.wi -= lr * self.wi_diff
        self.wf -= lr * self.wf_diff
        self.wo -= lr * self.wo_diff
        self.bg -= lr * self.bg_diff
        self.bi -= lr * self.bi_diff
```

```
        self.bf -= lr * self.bf_diff
        self.bo -= lr * self.bo_diff
        # reset diffs to zero
        self.wg_diff = np.zeros_like(self.wg)
        self.wi_diff = np.zeros_like(self.wi)
        self.wf_diff = np.zeros_like(self.wf)
        self.wo_diff = np.zeros_like(self.wo)
        self.bg_diff = np.zeros_like(self.bg)
        self.bi_diff = np.zeros_like(self.bi)
        self.bf_diff = np.zeros_like(self.bf)
        self.bo_diff = np.zeros_like(self.bo)

class LstmState:
    def __init__(self, mem_cell_ct, x_dim):
        self.g = np.zeros(mem_cell_ct)
        self.i = np.zeros(mem_cell_ct)
        self.f = np.zeros(mem_cell_ct)
        self.o = np.zeros(mem_cell_ct)
        self.s = np.zeros(mem_cell_ct)
        self.h = np.zeros(mem_cell_ct)
        self.bottom_diff_h = np.zeros_like(self.h)
        self.bottom_diff_s = np.zeros_like(self.s)

class LstmNode:
    def __init__(self, lstm_param, lstm_state):
        # store reference to parameters and to activations
        self.state = lstm_state
        self.param = lstm_param
        # non-recurrent input concatenated with recurrent input
        self.xc = None

    def bottom_data_is(self, x, s_prev = None, h_prev = None):
        # if this is the first lstm node in the network
        if s_prev is None: s_prev = np.zeros_like(self.state.s)
        if h_prev is None: h_prev = np.zeros_like(self.state.h)
        # save data for use in backprop
        self.s_prev = s_prev
        self.h_prev = h_prev

        # concatenate x(t) and h(t-1)
        xc = np.hstack((x,  h_prev))
        self.state.g = np.tanh(np.dot(self.param.wg, xc) + self.
    param.bg)
        self.state.i = sigmoid(np.dot(self.param.wi, xc) + self.
    param.bi)
        self.state.f = sigmoid(np.dot(self.param.wf, xc) + self.
    param.bf)
        self.state.o = sigmoid(np.dot(self.param.wo, xc) + self.
    param.bo)
```

```
        self.state.s = self.state.g * self.state.i + s_prev *
    self.state.f
        self.state.h = np.tanh(self.state.s) * self.state.o

        self.xc = xc

    def top_diff_is(self, top_diff_h, top_diff_s):
        # notice that top_diff_s is carried along the constant
    error carousel
        ds = self.state.o * top_diff_h + top_diff_s
        do = self.state.s * top_diff_h
        di = self.state.g * ds
        dg = self.state.i * ds
        df = self.s_prev * ds

        # diffs w.r.t. vector inside sigma / tanh function
        di_input = sigmoid_derivative(self.state.i) * di
        df_input = sigmoid_derivative(self.state.f) * df
        do_input = sigmoid_derivative(self.state.o) * do
        dg_input = tanh_derivative(self.state.g) * dg

        # diffs w.r.t. inputs
        self.param.wi_diff += np.outer(di_input, self.xc)
        self.param.wf_diff += np.outer(df_input, self.xc)
        self.param.wo_diff += np.outer(do_input, self.xc)
        self.param.wg_diff += np.outer(dg_input, self.xc)
        self.param.bi_diff += di_input
        self.param.bf_diff += df_input
        self.param.bo_diff += do_input
        self.param.bg_diff += dg_input

        # compute bottom diff
        dxc = np.zeros_like(self.xc)
        dxc += np.dot(self.param.wi.T, di_input)
        dxc += np.dot(self.param.wf.T, df_input)
        dxc += np.dot(self.param.wo.T, do_input)
        dxc += np.dot(self.param.wg.T, dg_input)

        # save bottom diffs
        self.state.bottom_diff_s = ds * self.state.f
        self.state.bottom_diff_h = dxc[self.param.x_dim:]

class LstmNetwork():
    def __init__(self, lstm_param):
        self.lstm_param = lstm_param
        self.lstm_node_list = []
        # input sequence
        self.x_list = []

    def y_list_is(self, y_list, loss_layer):
```

```
        assert len(y_list) == len(self.x_list)
        idx = len(self.x_list) - 1
        # first node only gets diffs from label ...
        loss = loss_layer.loss(self.lstm_node_list[idx].state.h,
 y_list[idx])
        diff_h = loss_layer.bottom_diff(self.lstm_node_list[idx
].state.h, y_list[idx])
        # here s is not affecting loss due to h(t+1), hence we
set equal to zero
        diff_s = np.zeros(self.lstm_param.mem_cell_ct)
        self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
        idx -= 1

        ### ... following nodes also get diffs from next nodes,
hence we add diffs to diff_h
        ### we also propagate error along constant error
carousel using diff_s
        while idx >= 0:
            loss += loss_layer.loss(self.lstm_node_list[idx].
state.h, y_list[idx])
            diff_h = loss_layer.bottom_diff(self.lstm_node_list[
idx].state.h, y_list[idx])
            diff_h += self.lstm_node_list[idx + 1].state.
bottom_diff_h
            diff_s = self.lstm_node_list[idx + 1].state.
bottom_diff_s
            self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
            idx -= 1

        return loss

    def x_list_clear(self):
        self.x_list = []

    def x_list_add(self, x):
        self.x_list.append(x)
        if len(self.x_list) > len(self.lstm_node_list):
            # need to add new lstm node, create new state mem
            lstm_state = LstmState(self.lstm_param.mem_cell_ct,
self.lstm_param.x_dim)
            self.lstm_node_list.append(LstmNode(self.lstm_param,
 lstm_state))

        # get index of most recent x input
        idx = len(self.x_list) - 1
        if idx == 0:
            # no recurrent inputs yet
            self.lstm_node_list[idx].bottom_data_is(x)
        else:
            s_prev = self.lstm_node_list[idx - 1].state.s
```

```
            h_prev = self.lstm_node_list[idx - 1].state.h
            self.lstm_node_list[idx].bottom_data_is(x, s_prev,
    h_prev)

class LossLayer:
    @staticmethod
    def loss(y_pred, y):
        # Mean squared error
        return ((y_pred - y) ** 2).sum()

    @staticmethod
    def bottom_diff(y_pred, y):
        # Derivative of MSE for backpropagation
        return 2 * (y_pred - y)

# AR Model Data Generation
np.random.seed(0)
a1, a2, a3 = 0.5, -0.1, 0.2
n_samples = 1000
train_size = int(n_samples * 0.8)
test_size = n_samples - train_size
U = np.random.uniform(-0.25, 0.25, n_samples)
X = np.zeros(n_samples)
for t in range(3, n_samples):
    X[t-3] = a1 * X[t-1] + a2 * X[t-2] + a3 * X[t-3] + U[t]

rho, sigma = yule_walker(X[3:], order=3, method='mle')
print(f"Estimated coefficients: {rho}")
print(f"Estimated noise variance: {sigma**2}")

# Data Preparation for LSTM
def create_dataset(X, U, look_back):
    dataX, dataY = [], []
    for i in range(len(X) - look_back):
        a = X[i:(i + look_back)] + U[i + look_back]
        dataX.append(a)
        dataY.append(X[i + look_back])
    return np.array(dataX), np.array(dataY)

look_back = 3
trainU, testU = U[:train_size], U[train_size - look_back:]
trainX, testX = X[:train_size], X[train_size - look_back:]
trainX, trainY = create_dataset(trainX, trainU, look_back)
testX, testY = create_dataset(testX, testU, look_back)

# Initialize and Train LSTM Model
mem_cell_ct = 4
x_dim = look_back
lstm_param = LstmParam(mem_cell_ct, x_dim)
lstm_net = LstmNetwork(lstm_param)
```

```
loss_layer = LossLayer()

epochs = 100
learning_rate = 0.001
for epoch in range(epochs):
    lstm_net.x_list_clear()
    total_loss = 0  # Initialize total_loss to accumulate loss
    over the epoch
    for i in range(len(trainX)):
        lstm_net.x_list_clear()  # Clear previous inputs
        lstm_net.x_list_add(trainX[i])  # Add current input
        loss = lstm_net.y_list_is([trainY[i]], loss_layer)  #
    Process current input
        total_loss += loss  # Accumulate the loss
    lstm_param.apply_diff(learning_rate)  # Apply the learning
    rate to the gradients
    average_loss = total_loss / len(trainX)  # Calculate the
    average loss
    print(f"Epoch: {epoch}, Average Loss: {average_loss}")

# Adjusted function to predict with the LSTM network
def predict_with_lstm(lstm_net, dataX):
    predictions = []
    for x in dataX:
        lstm_net.x_list_clear()  # Clear the network's state for
     the new sequence
        lstm_net.x_list_add(x)  # Add the current input to the
    network
        # Append the last hidden state's value as the prediction
        # This assumes a single value prediction per sequence
        predictions.append(lstm_net.lstm_node_list[-1].state.h
    [-1])
    return np.array(predictions)

# Function to calculate Mean Squared Error
def calculate_mse(predictions, actuals):
    # Ensure both are numpy arrays to support element-wise
    operations
    predictions = np.array(predictions)
    actuals = np.array(actuals)
    # Flatten both arrays to ensure MSE is calculated correctly
    across all elements
    return np.mean((actuals.flatten() - predictions.flatten())
    ** 2)

# Use the adjusted prediction function on the test dataset
test_predictions = predict_with_lstm(lstm_net, testX)

# Calculate the Mean Squared Error between LSTM predictions and
    actual test data
```

```
test_mse = calculate_mse(test_predictions, testY)
print("Test Score (Mean Squared Error):", test_mse)

yule_walker_predictions = np.zeros(n_samples)
for t in range(3, n_samples):
    yule_walker_predictions[t] = sum(rho * X[t-3:t][::-1])

yule_walker_mse = np.mean((X[3:] - yule_walker_predictions[3:])
    **2)
print("Yule-Walker MSE:", yule_walker_mse)
```

- For $n = 100$: The average cost square error for different training samples is approximately: 0.017756112125548222. The average cost square error using Yule-Walker equations is : 0.019502233593195393 so are quite identical.
- For $n = 500$: The average cost square error for different training samples is approximately: 0.021754833558894678. The average cost square error using Yule-Walker equations is : 0.020706582804191815 so are quite identical.
- For $n = 1000$: The average cost square error for different training samples is approximately: 0.019848015133495312. The average cost square error using Yule-Walker equations is : 0.020834711526501463 so are quite identical
- For $n = 5000$: The average cost square error for different training samples is approximately: 0.019895739168232472. The average cost square error using Yule-Walker equations is : 0.02090381788722509 so are quite identical
- For $n = 10000$: The average cost square error for different training samples is approximately: 0.020878363256433944. The average cost square error using Yule-Walker equations is : 0.020970077164667908 so are quite identical

6 PROBLEM-06

Code for this problem:

```
import numpy as np

def sigmoid(x):
    return 1. / (1 + np.exp(-x))

def sigmoid_derivative(values):
    return values*(1-values)

def tanh_derivative(values):
    return 1. - values ** 2

# createst uniform random array w/ values in [a,b) and shape
    args
def rand_arr(a, b, *args):
    np.random.seed(0)
    return np.random.rand(*args) * (b - a) + a
```

```
class LstmParam:
    def __init__(self, mem_cell_ct, x_dim):
        self.mem_cell_ct = mem_cell_ct
        self.x_dim = x_dim
        concat_len = x_dim + mem_cell_ct
        # weight matrices
        self.wg = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wi = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wf = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wo = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        # bias terms
        self.bg = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bi = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bf = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bo = rand_arr(-0.1, 0.1, mem_cell_ct)
        # diffs (derivative of loss function w.r.t. all
    parameters)
        self.wg_diff = np.zeros((mem_cell_ct, concat_len))
        self.wi_diff = np.zeros((mem_cell_ct, concat_len))
        self.wf_diff = np.zeros((mem_cell_ct, concat_len))
        self.wo_diff = np.zeros((mem_cell_ct, concat_len))
        self.bg_diff = np.zeros(mem_cell_ct)
        self.bi_diff = np.zeros(mem_cell_ct)
        self.bf_diff = np.zeros(mem_cell_ct)
        self.bo_diff = np.zeros(mem_cell_ct)

    def apply_diff(self, lr = 1):
        self.wg -= lr * self.wg_diff
        self.wi -= lr * self.wi_diff
        self.wf -= lr * self.wf_diff
        self.wo -= lr * self.wo_diff
        self.bg -= lr * self.bg_diff
        self.bi -= lr * self.bi_diff
        self.bf -= lr * self.bf_diff
        self.bo -= lr * self.bo_diff
        # reset diffs to zero
        self.wg_diff = np.zeros_like(self.wg)
        self.wi_diff = np.zeros_like(self.wi)
        self.wf_diff = np.zeros_like(self.wf)
        self.wo_diff = np.zeros_like(self.wo)
        self.bg_diff = np.zeros_like(self.bg)
        self.bi_diff = np.zeros_like(self.bi)
        self.bf_diff = np.zeros_like(self.bf)
        self.bo_diff = np.zeros_like(self.bo)

class LstmState:
    def __init__(self, mem_cell_ct, x_dim):
        self.g = np.zeros(mem_cell_ct)
        self.i = np.zeros(mem_cell_ct)
```

```
        self.f = np.zeros(mem_cell_ct)
        self.o = np.zeros(mem_cell_ct)
        self.s = np.zeros(mem_cell_ct)
        self.h = np.zeros(mem_cell_ct)
        self.bottom_diff_h = np.zeros_like(self.h)
        self.bottom_diff_s = np.zeros_like(self.s)

class LstmNode:
    def __init__(self, lstm_param, lstm_state):
        # store reference to parameters and to activations
        self.state = lstm_state
        self.param = lstm_param
        # non-recurrent input concatenated with recurrent input
        self.xc = None

    def bottom_data_is(self, x, s_prev = None, h_prev = None):
        # if this is the first lstm node in the network
        if s_prev is None: s_prev = np.zeros_like(self.state.s)
        if h_prev is None: h_prev = np.zeros_like(self.state.h)
        # save data for use in backprop
        self.s_prev = s_prev
        self.h_prev = h_prev

        # concatenate x(t) and h(t-1)
        xc = np.hstack((x,  h_prev))
        self.state.g = np.tanh(np.dot(self.param.wg, xc) + self.
    param.bg)
        self.state.i = sigmoid(np.dot(self.param.wi, xc) + self.
    param.bi)
        self.state.f = sigmoid(np.dot(self.param.wf, xc) + self.
    param.bf)
        self.state.o = sigmoid(np.dot(self.param.wo, xc) + self.
    param.bo)
        self.state.s = self.state.g * self.state.i + s_prev *
    self.state.f
        self.state.h = np.tanh(self.state.s) * self.state.o

        self.xc = xc

    def top_diff_is(self, top_diff_h, top_diff_s):
        # notice that top_diff_s is carried along the constant
    error carousel
        ds = self.state.o * top_diff_h + top_diff_s
        do = self.state.s * top_diff_h
        di = self.state.g * ds
        dg = self.state.i * ds
        df = self.s_prev * ds

        # diffs w.r.t. vector inside sigma / tanh function
        di_input = sigmoid_derivative(self.state.i) * di
```

```
        df_input = sigmoid_derivative(self.state.f) * df
        do_input = sigmoid_derivative(self.state.o) * do
        dg_input = tanh_derivative(self.state.g) * dg

        # diffs w.r.t. inputs
        self.param.wi_diff += np.outer(di_input, self.xc)
        self.param.wf_diff += np.outer(df_input, self.xc)
        self.param.wo_diff += np.outer(do_input, self.xc)
        self.param.wg_diff += np.outer(dg_input, self.xc)
        self.param.bi_diff += di_input
        self.param.bf_diff += df_input
        self.param.bo_diff += do_input
        self.param.bg_diff += dg_input

        # compute bottom diff
        dxc = np.zeros_like(self.xc)
        dxc += np.dot(self.param.wi.T, di_input)
        dxc += np.dot(self.param.wf.T, df_input)
        dxc += np.dot(self.param.wo.T, do_input)
        dxc += np.dot(self.param.wg.T, dg_input)

        # save bottom diffs
        self.state.bottom_diff_s = ds * self.state.f
        self.state.bottom_diff_h = dxc[self.param.x_dim:]

class LstmNetwork():
    def __init__(self, lstm_param):
        self.lstm_param = lstm_param
        self.lstm_node_list = []
        # input sequence
        self.x_list = []

    def y_list_is(self, y_list, loss_layer):
        assert len(y_list) == len(self.x_list)
        idx = len(self.x_list) - 1
        # first node only gets diffs from label ...
        loss = loss_layer.loss(self.lstm_node_list[idx].state.h,
     y_list[idx])
        diff_h = loss_layer.bottom_diff(self.lstm_node_list[idx
    ].state.h, y_list[idx])
        # here s is not affecting loss due to h(t+1), hence we
    set equal to zero
        diff_s = np.zeros(self.lstm_param.mem_cell_ct)
        self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
        idx -= 1

        ### ... following nodes also get diffs from next nodes,
    hence we add diffs to diff_h
        ### we also propagate error along constant error
    carousel using diff_s
```

```
        while idx >= 0:
            loss += loss_layer.loss(self.lstm_node_list[idx].
    state.h, y_list[idx])
            diff_h = loss_layer.bottom_diff(self.lstm_node_list[
    idx].state.h, y_list[idx])
            diff_h += self.lstm_node_list[idx + 1].state.
    bottom_diff_h
            diff_s = self.lstm_node_list[idx + 1].state.
    bottom_diff_s
            self.lstm_node_list[idx].top_diff_is(diff_h, diff_s)
            idx -= 1

        return loss

    def x_list_clear(self):
        self.x_list = []

    def x_list_add(self, x):
        self.x_list.append(x)
        if len(self.x_list) > len(self.lstm_node_list):
            # need to add new lstm node, create new state mem
            lstm_state = LstmState(self.lstm_param.mem_cell_ct,
    self.lstm_param.x_dim)
            self.lstm_node_list.append(LstmNode(self.lstm_param,
     lstm_state))

        # get index of most recent x input
        idx = len(self.x_list) - 1
        if idx == 0:
            # no recurrent inputs yet
            self.lstm_node_list[idx].bottom_data_is(x)
        else:
            s_prev = self.lstm_node_list[idx - 1].state.s
            h_prev = self.lstm_node_list[idx - 1].state.h
            self.lstm_node_list[idx].bottom_data_is(x, s_prev,
    h_prev)

class LossLayer:
    @staticmethod
    def loss(y_pred, y):
        # Mean squared error
        return ((y_pred - y) ** 2).sum()

    @staticmethod
    def bottom_diff(y_pred, y):
        # Derivative of MSE for backpropagation
        return 2 * (y_pred - y)

np.random.seed(0)
```

```
# Constants for the MA model
a1, a2 = 5, 5
a3, a4, a5, a6 = -0.5, -0.5, -0.5, -0.5
n_samples = 1000
train_size = int(n_samples * 0.8)
test_size = n_samples - train_size
# Generate MA model data
U = np.random.uniform(0, 0.5, n_samples+6) # Include extra
    values for the initial lags
X = np.zeros(n_samples)

for t in range(6, n_samples + 6):  # Start from 6 to use
    previous values
    X[t-6] = (U[t] + a1 * U[t-1] + a2 * U[t-2] + a3 * U[t-3] +
    a4 * U[t-4] + a5 * U[t-5] + a6 * U[t-6])

# Define create_dataset function
def create_dataset(X, U, look_back):
    dataX, dataY = [], []
    for i in range(len(X) - look_back):
        a = X[i:(i + look_back)] + U[i + look_back]
        dataX.append(a)
        dataY.append(X[i + look_back])
    return np.array(dataX), np.array(dataY)

# Prepare the data for RNN training
look_back = 6  # Look back 6 steps
trainU, testU = U[:train_size], U[train_size - look_back:]
trainX, testX = X[:train_size], X[train_size - look_back:]
trainX, trainY = create_dataset(trainX, trainU, look_back)
testX, testY = create_dataset(testX, testU, look_back)

# Initialize and Train LSTM Model
mem_cell_ct = 4
x_dim = look_back
lstm_param = LstmParam(mem_cell_ct, x_dim)
lstm_net = LstmNetwork(lstm_param)
loss_layer = LossLayer()

epochs = 100
learning_rate = 0.001
for epoch in range(epochs):
    lstm_net.x_list_clear()
    total_loss = 0  # Initialize total_loss to accumulate loss
    over the epoch
    for i in range(len(trainX)):
        lstm_net.x_list_clear()  # Clear previous inputs
        lstm_net.x_list_add(trainX[i])  # Add current input
        loss = lstm_net.y_list_is([trainY[i]], loss_layer)  #
    Process current input
```

```
        total_loss += loss  # Accumulate the loss
    lstm_param.apply_diff(learning_rate)  # Apply the learning
    rate to the gradients
    average_loss = total_loss / len(trainX)  # Calculate the
    average loss
    print(f"Epoch: {epoch}, Average Loss: {average_loss}")

# Adjusted function to predict with the LSTM network
def predict_with_lstm(lstm_net, dataX):
    predictions = []
    for x in dataX:
        lstm_net.x_list_clear()  # Clear the network's state for
     the new sequence
        lstm_net.x_list_add(x)  # Add the current input to the
    network
        # Append the last hidden state's value as the prediction
        # This assumes a single value prediction per sequence
        predictions.append(lstm_net.lstm_node_list[-1].state.h
    [-1])
    return np.array(predictions)

# Function to calculate Mean Squared Error
def calculate_mse(predictions, actuals):
    # Ensure both are numpy arrays to support element-wise
    operations
    predictions = np.array(predictions)
    actuals = np.array(actuals)
    # Flatten both arrays to ensure MSE is calculated correctly
    across all elements
    return np.mean((actuals.flatten() - predictions.flatten())
    ** 2)

# Use the adjusted prediction function on the test dataset
test_predictions = predict_with_lstm(lstm_net, testX)

# Calculate the Mean Squared Error between LSTM predictions and
    actual test data
test_mse = calculate_mse(test_predictions, testY)
print("Test Score (Mean Squared Error):", test_mse)
```

- For $n = 100$: The average cost square error for different training samples is approximately: 2.3569437390041643.
- For $n = 500$: The average cost square error for different training samples is approximately: 3.4031113046744057.
- For $n = 1000$: The average cost square error for different training samples is approximately: 3.1940250750087795.
- For $n = 5000$: The average cost square error for different training samples is approximately: 5.826560541290559.

– For n = 10000 : The average cost square error for different training samples is approximately: 6.113200248460094.

7 PROBLEM-7

7.1 A. Large Integers

A large integer can be subjective, so we define it as a number bigger than 50000 for example.The membership function could assign a membership of 0 to integers below a certain threshold (say, 50000) and then gradually increase to 1 as the numbers get larger. The membership function for large integers is defined as a sigmoid function shifted by 50000, which gives an S-shaped curve:

$$\mu_{\text{large integers}}(x) = \frac{1}{1 + e^{-k(x - x_0)}}$$

=

$$\mu_{\text{large integers}}(x) = \frac{1}{1 + e^{-0.0001(x - 50000)}}$$

where x_0 is the value at which integers are considered large, and k determines the steepness of the curve. So the fuzzy subset for large integers is:

$$A = \left\{ (x, \mu_{\text{large integers}}(x) = \frac{1}{1 + e^{-0.0001(x - 50000)}}) \mid \forall x \in X \right\}$$

where X is basically all integers.

7.2 B. Very Small Numbers

Very small numbers can be considered in the context of values approaching zero. the membership would be 1 for numbers close to 0 and would decrease to 0 as numbers get larger. A suitable function could be a decreasing exponential function:

$$\mu_{\text{very small numbers}}(x) = e^{-10x}$$

So the fuzzy subset for large integers is:

$$B = \left\{ (x, \mu_{\text{very small numbers}}(x) = e^{-10x}) \mid \forall x \in X \right\}$$

where X is basically all numbers.

7.3 C. Medium-weight men

Assuming medium-weight is defined within a certain range, such as from 65 to 85 kilograms, we could create a trapezoidal membership function that is 1 within this range and decreases to 0 outside of it:

$$\mu_{\text{medium weight men}}(x) = \max\left(\min\left(\min\left(\frac{x - a}{b - a}, 1\right), \frac{d - x}{d - c}\right), 0\right)$$

where a, b, c, and d define the ”feet” and ”shoulders” of the trapezoidal membership function. So the fuzzy subset for large integers is:

$$C = \{(x, \mu_{\text{medium weight men}}(x)) \mid \forall x \in X\}$$

where

$$\mu_{\text{medium weight men}}(x) = \max\left(\min\left(\min\left(\frac{x-a}{b-a}, 1\right), \frac{d-x}{d-c}\right), 0\right)$$

and X is basically weights of men.

7.4 D. Numbers approximately between 10 and 20

A trapezoidal membership function could be used here, where its 1 for numbers between 10 and 20, and 0 for all the others. So the membership function is:

$$\mu_{\text{Numbers approximately between 10 and 20}}(x) = \begin{cases} 1 & \text{if } 10 \leq x \leq 20 \\ 0 & \text{otherwise} \end{cases}$$

So the fuzzy subset for large integers is:

$$D = \{(x, \mu_{\text{Numbers approximately between 10 and 20}}) \mid \forall x \in X\}$$

where

$$\mu_{\text{Numbers approximately between 10 and 20}} = \begin{cases} 1 & \text{if } 10 \leq x \leq 20 \\ 0 & \text{otherwise} \end{cases}$$

and X is basically all numbers.

8 PROBLEM-8

Given the fuzzy relation with the membership function:

$$\mu_{\tilde{R}}(x, y) = 1 - \frac{1}{1 + x^2 + y^2}$$

We want to find the ordinary relation of level $\alpha = 0.3$. This requires us to solve the inequality:

$$\mu_{\tilde{R}}(x, y) \geq 0.3$$

Substituting the membership function into the inequality, we have:

$$1 - \frac{1}{1 + x^2 + y^2} \geq 0.3$$

Solving for x and y, we get:

$$\frac{1}{1 + x^2 + y^2} \leq 0.7$$

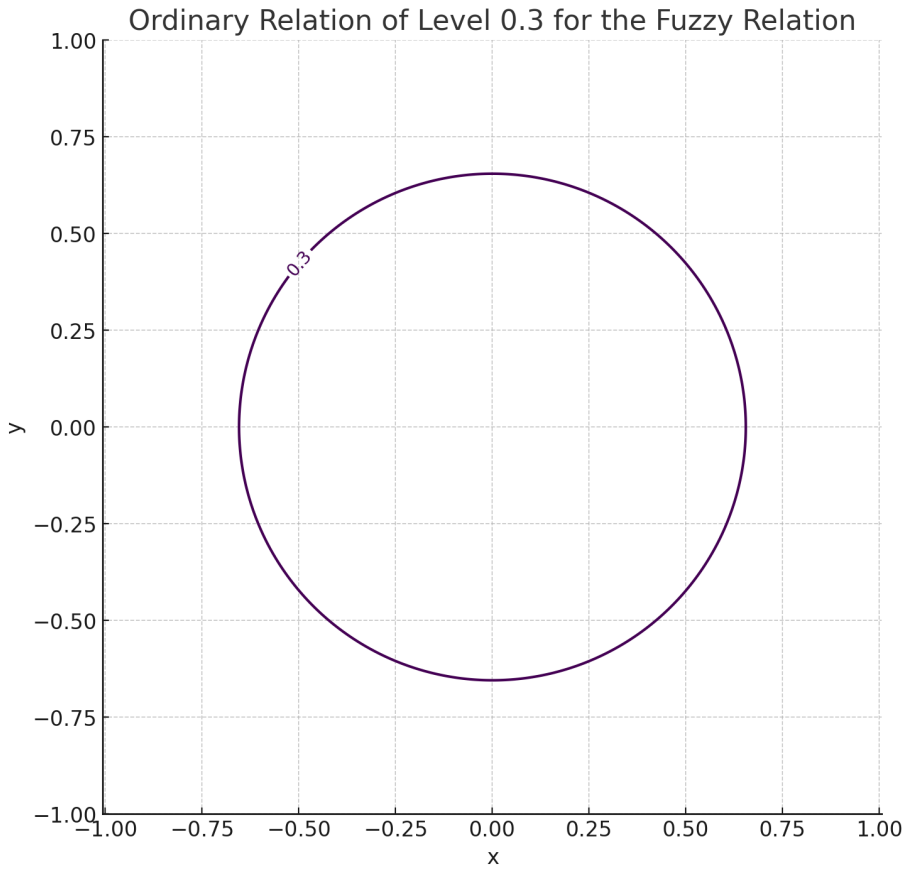
$$1 + x^2 + y^2 \geq \frac{1}{0.7}$$

$$x^2 + y^2 \geq \frac{1}{0.7} - 1$$

$$x^2 + y^2 \geq \frac{3}{7}$$

The solutions are the set of points (x, y) that satisfy the above inequality, which forms a circle centered at the origin with a radius squared of $\frac{3}{7}$.

This means that inside the circle the value of the membership function will be 0 and outside 1. At the circle itself will be 0 or 1. Here is the graph:



9 PROBLEM-9

9.1 Part A

The index of fuzziness $v(\tilde{A})$ is given by the integral representing the Hamming distance over $[0, \alpha]$. Since n is the cardinality of the reference set E, in the continuous case, this would correspond to the length of the interval, which is α. Also, the complement $\tilde{A}^c$ for any fuzzy set $\tilde{A}$ is given by:

$$\mu_{\tilde{A}^c}(x) = 1 - \mu_{\tilde{A}}(x)$$

Thus, the index of fuzziness is ($\frac{2x^2}{\alpha^2} - 1$ is less than zero between $x \in [0, \frac{\alpha\sqrt{2}}{2}]$ and greater than zero between $x \in [\frac{\alpha\sqrt{2}}{2}], \alpha)$:

$$\begin{aligned}
v(\tilde{A}) &= \frac{2}{\alpha} \cdot d(\tilde{A}, \tilde{A}^c) \\
&= \frac{2}{\alpha} \cdot \int_0^{\alpha} \left| \frac{x^2}{\alpha^2} - \left(1 - \frac{x^2}{\alpha^2}\right) \right| dx \\
&= \frac{2}{\alpha} \cdot \left(\int_0^{\alpha} \left| \frac{2x^2}{\alpha^2} - 1 \right| dx \right) \\
&= \frac{2}{\alpha} \cdot \left(\int_0^{\frac{\alpha\sqrt{2}}{2}} 1 - \frac{2x^2}{\alpha^2} dx + \int_{\frac{\alpha\sqrt{2}}{2}}^{\alpha} \frac{2x^2}{\alpha^2} - 1 dx \right) \\
&= \frac{2}{\alpha} \cdot \left(\left[x - \frac{2x^3}{3\alpha^2} \right]_0^{\frac{\alpha\sqrt{2}}{2}} + \left[\frac{2x^3}{3\alpha^2} - x \right]_{\frac{\alpha\sqrt{2}}{2}}^{a} \right) \\
&= \frac{2}{\alpha} \cdot \frac{2 \cdot \alpha \cdot \sqrt{2} - \alpha}{3} \\
&= \frac{4 \cdot \sqrt{2} - 2}{3}
\end{aligned}$$

9.2 Part B

The index of fuzziness $v(\tilde{A})$ is given by the sum of two integrals representing the Hamming distance over the two intervals $[0, \frac{\alpha}{2}]$ and $(\frac{\alpha}{2}, \alpha]$. Since n is the cardinality of the reference set E, in the continuous case, this would correspond to the length of the interval, which is α. Also, the complement $\tilde{A}^c$ for any fuzzy set $\tilde{A}$ is given by:

$$\mu_{\tilde{A}^c}(x) = 1 - \mu_{\tilde{A}}(x)$$

Thus, the index of fuzziness simplifies to:

$$v(\tilde{A}) = \frac{2}{\alpha} \cdot \left(d(\tilde{A}, \tilde{A}^c) = \frac{2}{\alpha} \cdot \int_0^{\frac{\alpha}{2}} \left| \frac{4x^2}{\alpha^2} - \left(1 - \frac{4x^2}{\alpha^2}\right) \right| dx + \int_{\frac{\alpha}{2}}^{\alpha} \left| \frac{4(x-\alpha)^2}{\alpha^2} - \left(1 - \frac{4(x-\alpha)^2}{\alpha^2}\right) \right| dx \right)$$
$$= \frac{2}{\alpha} \cdot \left(\int_0^{\frac{\alpha}{2}} \left| \frac{8x^2}{\alpha^2} - 1 \right| dx + \int_{\frac{\alpha}{2}}^{\alpha} \left| \frac{8(x-\alpha)^2}{\alpha^2} - 1 \right| dx \right)$$

The integral of the difference over the interval $x \in [0, \frac{\alpha}{2}]$ ($\frac{8x^2}{\alpha^2} - 1$ is less than zero between $x \in [0, \frac{\alpha\sqrt{2}}{4}]$ and greater than zero between $x \in [\frac{\alpha\sqrt{2}}{4}], \frac{\alpha}{2})$ is:

$$\int_0^{\frac{\alpha}{2}} \left| \frac{8x^2}{\alpha^2} - 1 dx \right| = \int_0^{\frac{\alpha\sqrt{2}}{4}} 1 - \frac{8x^2}{\alpha^2} dx + \int_{\frac{\alpha\sqrt{2}}{4}}^{\frac{\alpha}{2}} \frac{8x^2}{\alpha^2} - 1 dx$$
$$= \left[x - \frac{8x^3}{3\alpha^2} \right]_0^{\frac{\alpha\sqrt{2}}{4}} + \left[\frac{8x^3}{3\alpha^2} - x \right]_{\frac{\alpha\sqrt{2}}{4}}^{\frac{\alpha}{2}}$$
$$= \frac{2 \cdot \alpha \cdot \sqrt{2} - \alpha}{6}$$

The integral of the absolute difference over the interval $x \in [\frac{\alpha}{2}, \alpha]$ ($\frac{8(x-a)^2}{\alpha^2} - 1$ is greater than zero between $x \in [\frac{\alpha}{2}, \alpha - \frac{\alpha \cdot \sqrt{2}}{4}]$ and less than zero between $x \in [\alpha - \frac{\alpha \cdot \sqrt{2}}{4}, \alpha]$) is:

$$\int_{\frac{\alpha}{2}}^{\alpha} \left| \frac{8(x-a)^2}{\alpha^2} - 1 dx \right| = \int_{\frac{\alpha}{2}}^{\alpha - \frac{\alpha \cdot \sqrt{2}}{4}} \frac{8(x-a)^2}{\alpha^2} - 1 dx + \int_{\alpha - \frac{\alpha \cdot \sqrt{2}}{4}}^{\alpha} 1 - \frac{8(x-a)^2}{\alpha^2} dx$$
$$= \left[\frac{8(x-a)^3}{3\alpha^2} - x \right]_{\frac{\alpha}{2}}^{\alpha - \frac{\alpha \cdot \sqrt{2}}{4}} + \left[x - \frac{8(x-a)^3}{3\alpha^2} \right]_{\alpha - \frac{\alpha \cdot \sqrt{2}}{4}}^{\alpha}$$
$$= \frac{2 \cdot \alpha \cdot \sqrt{2} - \alpha}{6}$$

Thus the index of fuzzines is

$$v(\tilde{A}) = \frac{2}{\alpha} \cdot \left(\frac{2 \cdot \alpha \cdot \sqrt{2} - \alpha}{6} + \frac{2 \cdot \alpha \cdot \sqrt{2} - \alpha}{6} \right) = \frac{4 \cdot \sqrt{2} - 2}{3}$$

10 PROBLEM-10

The max-min composition $\tilde{R}$ is defined as:

$$\mu_{\tilde{R}}(x, z) = \max_y \min(\mu_{\tilde{R}_1}(x, y), \mu_{\tilde{R}_2}(y, z))$$

Specifically, we need to find the minimum of the two membership functions for each y and then find the maximum of these minimums across all y. Since both membership functions are in the form of an exponential decay based on the square of the distance from a certain point, the minimum will also be of the same form because the minimum of two exponentials with the same base is just the exponential of the minimum of the exponents.

So, for any given x and z, and for each y, we have:

$$\min(\mu_{\tilde{R_1}}(x,y), \mu_{\tilde{R_2}}(y,z)) = \min(e^{-k(x-y)^2}, e^{-k(y-z)^2})$$

This is equivalent to:

$$e^{-\max(k(x-y)^2, k(y-z)^2)}$$

Now, to find the max-min composition, we want to maximize this expression over all y. Since we are dealing with an exponential function, which is a decreasing function of its exponent, maximizing the exponential is equivalent to minimizing its exponent. Therefore, we are looking for the value of y that minimizes:

$$\max(k(x-y)^2, k(y-z)^2)$$

This function will have a minimum when both arguments of the max function are equal, which occurs when $k(x-y)^2 = k(y-z)^2$. Solving for y gives us:

$$x - y = y - z$$

$$2y = x + z$$

$$y = \frac{x+z}{2}$$

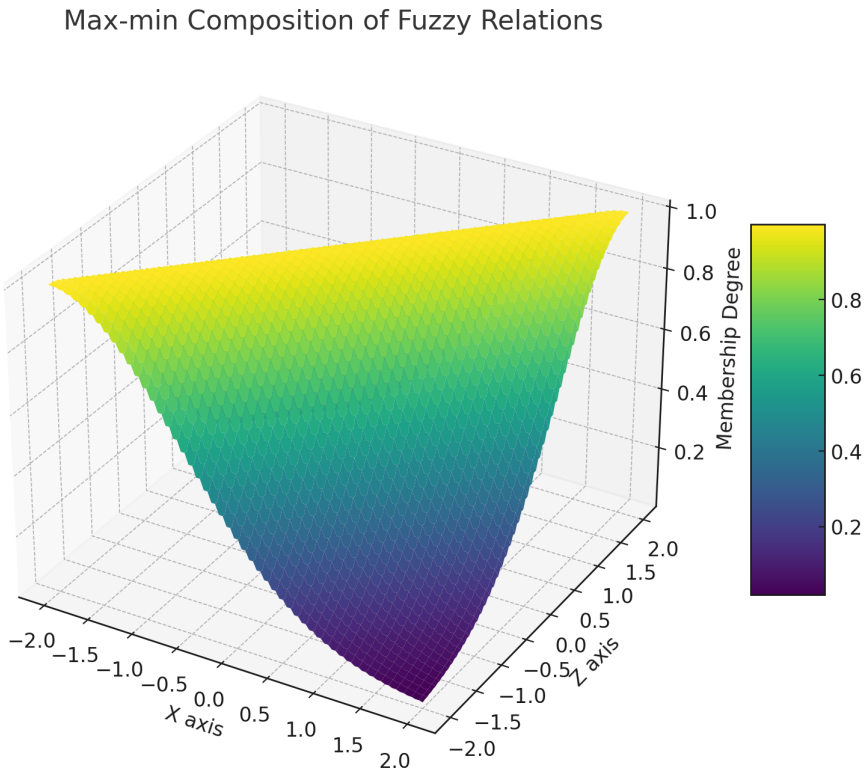
Substituting this value of y back into the membership functions, we obtain:

$$\mu_{\tilde{R}}(x,z) = e^{-k\left(x - \frac{x+z}{2}\right)^2}$$

This is the expression for the membership function of the max-min composition of $\tilde{R_1}$ and $\tilde{R_2}$. Let's simplify this expression:

$$\mu_{\tilde{R}}(x,z) = e^{-\frac{k}{4}(x-z)^2}$$

This gives us the final form of the max-min composition's membership function, which we graphically represented here:


Max-min Composition of Fuzzy Relations
X axis
Z axis
Membership Degree
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
0.2
0.4
0.6
0.8