#1A Fuzzy and or and not operation

```
import numpy as np

# Fuzzy values (between 0 and 1)

A = np.array([0.1, 0.4, 0.7])

B = np.array([0.2, 0.5, 0.9])

# Fuzzy AND = min(A, B)

fuzzy_and = np.minimum(A, B)

# Fuzzy OR = max(A, B)

fuzzy_or = np.maximum(A, B)

# Fuzzy NOT = 1 - A

fuzzy_not = 1 - A

print("A =", A)

print("B =", B)

print("Fuzzy AND =", fuzzy_and)

print("Fuzzy OR =", fuzzy_or)

print("Fuzzy NOT (of A) =", fuzzy_not)
```

```
#1B plot triangular and trapezoidal membership
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)

# Triangular membership function
def triangular(x, a, b, c):
    return np.maximum(np.minimum((x - a) / (b - a), (c - x) / (c - b)), 0)

# Trapezoidal membership function
def trapezoidal(x, a, b, c, d):
    return np.maximum(np.minimum(np.minimum((x - a) / (b - a), 1), (d - x) / (d - c)), 0)

# Compute membership values
tri = triangular(x, 2, 5, 8)
trap = trapezoidal(x, 2, 4, 6, 8)

# Plotting
plt.plot(x, tri, label='Triangular')
plt.plot(x, trap, label='Trapezoidal')
plt.title('Membership Functions')
plt.xlabel('x')
plt.ylabel('Membership')
plt.legend()
plt.grid(True)
plt.show()
```

```
#2A simple fuzzy inference system (fis)
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

temperature = ctrl.Antecedent(np.arange(0, 41, 1), 'temperature')
heater = ctrl.Consequent(np.arange(0, 101, 1), 'heater')

temperature['cold'] = fuzz.trimf(temperature.universe, [0, 0, 20])
temperature['warm'] = fuzz.trimf(temperature.universe, [15, 25, 35])
temperature['hot'] = fuzz.trimf(temperature.universe, [30, 40, 40])

heater['low'] = fuzz.trimf(heater.universe, [0, 0, 50])
heater['medium'] = fuzz.trimf(heater.universe, [25, 50, 75])
heater['high'] = fuzz.trimf(heater.universe, [50, 100, 100])

rule1 = ctrl.Rule(temperature['cold'], heater['high'])
rule2 = ctrl.Rule(temperature['warm'], heater['medium'])
rule3 = ctrl.Rule(temperature['hot'], heater['low'])

heater_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
heater_sim = ctrl.ControlSystemSimulation(heater_ctrl)

heater_sim.input['temperature'] = 100
heater_sim.compute()

print("Temperature: 100°C")
print("Heater Output:", heater_sim.output['heater'])
```

```
#3A McCulloch-Pitts neuron for AND gate


# Inputs and expected output
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

# Weights and threshold for AND gate
weights = [1, 1]
threshold = 2

print("AND Gate using McCulloch-Pitts Neuron:")
for x1, x2 in inputs:
   # Weighted sum
   net_input = x1 * weights[0] + x2 * weights[1]


   # Activation function (step function)
   output = 1 if net_input >= threshold else 0


   print(f"Input: ({x1}, {x2}) → Output: {output}")
```

#3B McCulloch-Pitts Neuron for OR Gate

```
# Inputs and expected output
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

# Weights and threshold for OR gate
weights = [1, 1]
threshold = 1

print("OR Gate using McCulloch-Pitts Neuron:")

for x1, x2 in inputs:
    # Weighted sum
    net_input = x1 * weights[0] + x2 * weights[1]


    # Activation function (step function)
    output = 1 if net_input >= threshold else 0


    print(f"Input: ({x1}, {x2}) → Output: {output}")
```

```
# 5(A) Adaline Implementation
import numpy as np
# Step 1: Generate synthetic linearly separable data
X = np.array([[1, 1],
              [2, 1],
              [1, 2],
              [2, 2],
              [3, 1],
              [3, 2],
              [3, 4],
              [4, 3],
              [4, 4]])

y = np.array([-1, -1, -1, -1, 1, 1, 1, 1, 1])  # Binary class labels

# Step 2: Add bias term to input
X_bias = np.c_[np.ones((X.shape[0], 1)), X]  # Add bias as first column

# Step 3: Adaline Training Function
def adaline_train(X, y, lr=0.01, epochs=20):
    weights = np.zeros(X.shape[1])
    for epoch in range(epochs):
        output = np.dot(X, weights)
        error = y - output
        weights += lr * X.T.dot(error)
    return weights

# Step 4: Train the model
weights = adaline_train(X_bias, y)

# Step 5: Prediction
```

```
def predict(X, weights):
    X_bias = np.c_[np.ones((X.shape[0], 1)), X]
    return np.where(np.dot(X_bias, weights) >= 0, 1, -1)

# Step 6: Test the model
X_test = np.array([[1, 1], [4, 4], [2.5, 2.5]])
predictions = predict(X_test, weights)
print("Predictions:", predictions)
```

```
# 5(B) Adaline Error vs Epoch Graph
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Generate simple dataset
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([-1, -1, 1, 1, 1])

# Step 2: Add bias term
X_bias = np.c_[np.ones((X.shape[0], 1)), X]

# Step 3: Adaline with error tracking
def adaline_train_error(X, y, lr=0.01, epochs=20):
    weights = np.zeros(X.shape[1])
    errors = []

    for epoch in range(epochs):
        output = np.dot(X, weights)
        error = y - output
        weights += lr * X.T.dot(error)
        mse = (error**2).mean()
        errors.append(mse)

    return weights, errors
# Step 4: Train model and collect errors
weights, errors = adaline_train_error(X_bias, y)
# Step 5: Plot error vs epochs
plt.plot(range(1, len(errors)+1), errors, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Mean Squared Error')
plt.title('Adaline - Error vs Epoch')
plt.grid(True)
plt.show()
```

```
#6.A
#XOR Problem with Backpropagation
import numpy as np

# Activation and its derivative
def sigmoid(x): return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x): return x * (1 - x)

# Input/Output data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

np.random.seed(1)

# Initialize weights and biases
w_h = np.random.random((2, 4)) - 1  # Weights, hidden layer
bh = np.random.random((1, 4)) - 1   # Bias, hidden layer
w_o = np.random.random((4, 1)) - 1  # Weights, output layer
bo = np.random.random((1, 1)) - 1   # Bias, output layer

lr = 0.1  # Learning rate
epochs = 10000

print("Training on XOR problem...")
for i in range(epochs):
   # Forward propagation
   h_in = np.dot(X, w_h) + bh
   h_out = sigmoid(h_in)
   o_in = np.dot(h_out, w_o) + bo
   o_out = sigmoid(o_in)
```

```
    # Backpropagation
    err_out = y - o_out
    err_out = err_out * sigmoid_derivative(o_out)
    err_h = err_out.dot(w_o.T) * sigmoid_derivative(h_out)

    # Update weights and biases
    w_h += X.T.dot(err_h) * lr
    w_o += h_out.T.dot(err_out) * lr
    bh += np.sum(err_h, axis=0, keepdims=True) * lr
    bo += np.sum(err_out, axis=0, keepdims=True) * lr

print("Training complete.")
print("Final predictions:\n", o_out)
```

```
#P.7 (A)
#Implement GA to maximize f(x) = x² 2

import random
# Objective function
def fitness(x):
    return x**2

# Create initial population
def create_population(size, lower, upper):
    return [random.randint(lower, upper) for _ in range(size)]

# Selection (tournament style)
def select(population):
    a, b = random.sample(population, 2)
    return a if fitness(a) > fitness(b) else b

# Crossover (average of parents)
def crossover(parent1, parent2):
    return (parent1 + parent2) // 2

# Mutation (small random change)
def mutate(x, lower, upper, mutation_rate=0.1):
    if random.random() < mutation_rate:
        return random.randint(lower, upper)
    return x

# GA main loop
def genetic_algorithm(generations=20, pop_size=6, lower=-10, upper=10):
    population = create_population(pop_size, lower, upper)
    for gen in range(generations):
```

```
        new_population = [ ]
        for _ in range(pop_size):
            p1 = select(population)
            p2 = select(population)
            child = crossover(p1, p2)
            child = mutate(child, lower, upper)
            new_population.append(child)
        population = new_population
        best = max(population, key=fitness)
        print(f"Gen {gen+1}: Best = {best}, Fitness = {fitness(best)}")

# Run the GA
genetic_algorithm()
```

```
#P.7.B
#Compare different crossover/mutation rates
import random

# Objective function
def fitness(x):
    return x ** 2

# Create initial population
def create_population(size, lower, upper):
    return [random.randint(lower, upper) for _ in range(size)]

# Selection (tournament style)
def select(population):
    a, b = random.sample(population, 2)
    return a if fitness(a) > fitness(b) else b

# Crossover with crossover rate
def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        return (parent1 + parent2) // 2
    return parent1

# Mutation with mutation rate
def mutate(x, lower, upper, mutation_rate):
    if random.random() < mutation_rate:
        return random.randint(lower, upper)
    return x

# Genetic Algorithm with customizable rates
```

```python
def genetic_algorithm(crossover_rate, mutation_rate, generations=20, pop_size=6, lower=-10, upper=10):
    population = create_population(pop_size, lower, upper)
    print(f"\nRunning GA with crossover_rate={crossover_rate}, mutation_rate={mutation_rate}")

    for gen in range(generations):
        new_population = []
        for _ in range(pop_size):
            p1 = select(population)
            p2 = select(population)
            child = crossover(p1, p2, crossover_rate)
            child = mutate(child, lower, upper, mutation_rate)
            new_population.append(child)

        population = new_population
        best = max(population, key=fitness)
        print(f"Gen {gen+1}: Best = {best}, Fitness = {fitness(best)}")

# Run GA with different configurations
genetic_algorithm(0.9, 0.05)  # High crossover, low mutation
genetic_algorithm(0.6, 0.2)   # Moderate crossover, higher mutation
```

```
# P.9.A
#Hebbian Learning Binary Input-Output

# Binary input-output pairs
inputs = [[1, 0], [0, 1], [1, 1]]
outputs = [1, 1, 0]

# Initialize weights
weights = [0, 0]
print("Initial Weights:", weights)

# Hebb Rule: Δw = x * y
for x, y in zip(inputs, outputs):
  for i in range(len(weights)):
    weights[i] += x[i] * y
  print(f"After input {x}, output {y} → Weights: {weights}")

print("Final Weights:", weights)
```

```
# P.9.B.
#Hebbian Learning Modified Inputs

# New input-output pairs
new_inputs = [[1, 1], [0, 0], [1, 0]]
new_outputs = [1, 0, 1]

# Initialize weights
weights = [0, 0]
print("Initial Weights:", weights)

# Apply Hebb Rule with new data
for x, y in zip(new_inputs, new_outputs):
    for i in range(len(weights)):
        weights[i] += x[i] * y
    print(f"After input {x}, output {y} → Weights: {weights}")

print("Final Weights:", weights)
```

```
#P.10.A
#1D SOM Implementation on 2D Data
import numpy as np
import matplotlib.pyplot as plt

# Generate simple 2D dataset (two clusters)
np.random.seed(0)  # for reproducibility
data = np.vstack([
    np.random.randn(50, 2) + np.array([2, 2]),
    np.random.randn(50, 2) + np.array([-2, -2])
])

# SOM parameters
n_neurons = 10
n_epochs = 50
learning_rate = 0.3
sigma = 2.0  # neighbourhood width

# Initialize neuron weights randomly in 2D space
weights = np.random.rand(n_neurons, 2) * 4 - 2  # values in range [-2, 2]

# Training loop
for epoch in range(n_epochs):
    for x in data:
        # Find Best Matching Unit (BMU)
        bmu_index = np.argmin(np.linalg.norm(weights - x, axis=1))

        # Update BMU and its neighbours
        for i in range(n_neurons):
            dist = abs(i - bmu_index)
            h = np.exp(-dist**2 / (2 * sigma**2))
```

```
        weights[i] += learning_rate * h * (x - weights[i])

# Plot final weights and data
plt.scatter(data[:, 0], data[:, 1], c='gray', alpha=0.5, label="Data")
plt.plot(weights[:, 0], weights[:, 1], 'ro-', label="SOM Neurons")
plt.legend()
plt.title("1D SOM on 2D Data")
plt.grid(True)
plt.show()
```

```
#P.10.B
#Visualizing Neighbourhood Effects
import numpy as np
import matplotlib.pyplot as plt

# Create simple 2D dataset in [-1, 1] range
np.random.seed(42)
data = np.random.rand(100, 2) * 2 - 1  # shape: (100, 2)

# SOM parameters
n_neurons = 8
epochs = 30
lr = 0.2  # initial learning rate
sigma = 1.5

# Initialize neurons in a straight line from [-1, -1] to [1, 1]
weights = np.linspace([-1, -1], [1, 1], n_neurons)

plt.ion()  # interactive mode on

# Training loop
for epoch in range(epochs):
    for x in data:
        # Find Best Matching Unit (BMU)
        bmu_index = np.argmin(np.linalg.norm(weights - x, axis=1))

        # Update BMU and its neighbors
        for i in range(n_neurons):
            dist = abs(i - bmu_index)
            h = np.exp(-dist**2 / (2 * sigma**2))  # neighborhood function
            weights[i] += lr * h * (x - weights[i])
```

```
    # Live plot update
    plt.clf()
    plt.scatter(data[:, 0], data[:, 1], c='lightgray', label="Data")
    plt.plot(weights[:, 0], weights[:, 1], 'ro-', linewidth=2, label="SOM Neurons")
    plt.title(f"Epoch {epoch+1}")
    plt.legend()
    plt.pause(0.3)

plt.ioff()  # turn off interactive mode
plt.show()


#OP WILL BE 30 epoch
```