**Practical Lab File**

**(2022-2023)**

**Soft Computing**



**DEPARTMENT OF COMPUTER ENGINEERING & APPLICATIONS INSTITUTE OF ENGINEERING & TECHNOLOGY**

**Submitted To: Submitted By:**

**Dr. Manmohan Singh Sir Name - Surendra Pratap Tomar (Assistant Professor) Section - (B)**

**Roll No (67)**

**University Roll No - (191500831)**

INDEX

|  |  |  |
| --- | --- | --- |
| S.No | List of Experiment | Pages |
| 1. | **Implement AND logic function using Me-culloch pitts neuron model** | 3 |
| 2. | Implement OR logic function using Me-culloch pitts neuron model | 3-4 |
| 3. | Implement XOR logic function using Me-culloch pitts neuron model | 4-5 |
| 4. | Implement AND using perceptron neural network | 5-7 |
| 5. | **Implement OR using perceptron neural network** | 7-8 |
| 6. | **Implement OR using ADALINE neural network.** | 8 |
| 7. | Implement XOR using MADALINE neutral network. | 8-9 |
| 8. | Implement max-min composite relation for two fuzzy set relation. | 10 |
| 9. | Implement the Max product composite relation for 2 fuzzy set relation | 10-11 |
| 10. | The Optimization Problem: Maximize F(X)=X^2, Over the Set Of Integers In The Interval [0,31] | 11-14 |

**Experiment 1** – **Implement AND logic function using Me-culloch pitts neuron model**

**Code:-**

class MCPNeuron:

def \_\_init\_\_(self, weights, threshold):

self.weights = weights

self.threshold = threshold

def activate(self, inputs):

weighted\_sum = 0

for i in range(len(inputs)):

weighted\_sum += inputs[i] \* self.weights[i]

if weighted\_sum >= self.threshold:

return 1

else:

return 0

# Create an MCP neuron with two inputs and a threshold of 2

and\_neuron = MCPNeuron([1, 1], 2)

# Test the neuron with various input values

print(and\_neuron.activate([0, 0])) # 0

print(and\_neuron.activate([0, 1])) # 0

print(and\_neuron.activate([1, 0])) # 0

print(and\_neuron.activate([1, 1])) # 1

**#Inputs**

x1 = [0, 0, 1, 1]

x2 = [0, 1, 0, 1]

w1 = [1, 1, 1, 1]

w2 = [1, 1, 1, 1]

t = 2

**#Output**

print("x1 x2 w1 w2 t O")

for i **in** range(len(x1)):

if ( x1[i]\*w1[i] + x2[i]\*w2[i] ) >= t:

print(x1[i],' ',x2[i],' ',w1[i],' ',w2[i],' ',t,' ', 1)

else:

print(x1[i],' ',x2[i],' ',w1[i],' ',w2[i],' ',t,' ', 0)

x1 x2 w1 w2 t O

0 0 1 1 2 0

0 1 1 1 2 0

1 0 1 1 2 0

1 1 1 1 2 1

**Experiment 2- Implement OR logic function using Me-culloch pitts neuron model**

**Code:-**

**class MCPNeuron:**

**def \_\_init\_\_(self, weights, threshold):**

**self.weights = weights**

**self.threshold = threshold**

**def activate(self, inputs):**

**weighted\_sum = 0**

**for i in range(len(inputs)):**

**weighted\_sum += inputs[i] \* self.weights[i]**

**if weighted\_sum >= self.threshold:**

**return 1**

**else:**

**return 0**

**# Create an MCP neuron with two inputs and a threshold of 1**

**or\_neuron = MCPNeuron([1, 1], 1)**

**# Test the neuron with various input values**

**print(or\_neuron.activate([0, 0])) # 0**

**print(or\_neuron.activate([0, 1])) # 1**

**print(or\_neuron.activate([1, 0])) # 1**

**print(or\_neuron.activate([1, 1])) # 1**

**#Inputs**

x1 = [0, 0, 1, 1]

x2 = [0, 1, 0, 1]

w1 = [1, 1, 1, 1]

w2 = [1, 1, 1, 1]

t = 1

**#Output**

print("x1 x2 w1 w2 t O")

for i **in** range(len(x1)):

if ( x1[i]\*w1[i] + x2[i]\*w2[i] ) >= t:

print(x1[i],' ',x2[i],' ',w1[i],' ',w2[i],' ',t,' ', 1)

else:

print(x1[i],' ',x2[i],' ',w1[i],' ',w2[i],' ',t,' ', 0)

x1 x2 w1 w2 t O

0 0 1 1 1 0

0 1 1 1 1 1

1 0 1 1 1 1

1 1 1 1 1 1

**Experiment 3 Implement XOR logic function using Me-culloch pitts neuron model**

**Code:-** class MCPNeuron:

def \_\_init\_\_(self, weights, threshold):

self.weights = weights

self.threshold = threshold

def activate(self, inputs):

weighted\_sum = 0

for i in range(len(inputs)):

weighted\_sum += inputs[i] \* self.weights[i]

if weighted\_sum >= self.threshold:

return 1

else:

return 0

class MultiLayerPerceptron:

def \_\_init\_\_(self):

self.hidden\_layer = [

MCPNeuron([1, 1], 1),

MCPNeuron([-1, -1], -1),

]

self.output\_neuron = MCPNeuron([1, 1], 1)

def activate(self, inputs):

hidden\_layer\_outputs = [

neuron.activate(inputs) for neuron in self.hidden\_layer

]

output = self.output\_neuron.activate(hidden\_layer\_outputs)

return output

# Create a multi-layer perceptron with two inputs

mlp = MultiLayerPerceptron()

# Test the perceptron with various input values

print(mlp.activate([0, 0])) # 0

print(mlp.activate([0, 1])) # 1

print(mlp.activate([1, 0])) # 1

print(mlp.activate([1, 1])) # 0

**#Inputs**

x1 = [0, 0, 1, 1]

x2 = [0, 1, 0, 1]

w1 = [1, 1, 1, 1]

w2 = [1, 1, 1, 1]

w3 = [1, 1, 1, 1]

w4 = [-1, -1, -1, -1]

w5 = [-1, -1, -1, -1]

w6 = [1, 1, 1, 1]

t1 = [0.5,0.5,0.5,0.5]

t2 = [-1.5,-1.5,-1.5,-1.5]

t3 = [1.5,1.5,1.5,1.5]

def XOR (a, b):

if a != b:

return 1

else:

return 0

**#Output**

print('x1 x2 w1 w2 w3 w4 w5 w6 t1 t2 t3 O')

for i **in** range(len(x1)):

print(x1[i],' ',x2[i],' ',w1[i],' ',w2[i],' ',w3[i],' ',w4[i],' ',w5[i],' ',w6[i],' ',t1[i],' ',t2[i],' ',t3[i],' ',XOR(x1[i],x2[i]))

x1 x2 w1 w2 w3 w4 w5 w6 t1 t2 t3 O

0 0 1 1 1 -1 -1 1 0.5 -1.5 1.5 0

0 1 1 1 1 -1 -1 1 0.5 -1.5 1.5 1

1 0 1 1 1 -1 -1 1 0.5 -1.5 1.5 1

1 1 1 1 1 -1 -1 1 0.5 -1.5 1.5 0

**Experiment 4 Implement AND using perceptron neural network**

**Code:-**

**import numpy as np**

**class Perceptron:**

**def \_\_init\_\_(self, input\_size, learning\_rate=0.1):**

**self.weights = np.zeros(input\_size + 1)**

**self.learning\_rate = learning\_rate**

**def predict(self, inputs):**

**summation = np.dot(inputs, self.weights[1:]) + self.weights[0]**

**if summation > 0:**

**return 1**

**else:**

**return 0**

**def train(self, inputs, label):**

**prediction = self.predict(inputs)**

**error = label - prediction**

**self.weights[1:] += self.learning\_rate \* error \* inputs**

**self.weights[0] += self.learning\_rate \* error**

**# Define the AND function input/output pairs**

**training\_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

**labels = np.array([0, 0, 0, 1])**

**# Create a perceptron with two inputs**

**and\_perceptron = Perceptron(2)**

**# Train the perceptron on the AND function**

**for i in range(10000):**

**for inputs, label in zip(training\_inputs, labels):**

**and\_perceptron.train(inputs, label)**

**# Test the perceptron with various input values**

**print(and\_perceptron.predict([0, 0])) # 0**

**print(and\_perceptron.predict([0, 1])) # 0**

**print(and\_perceptron.predict([1, 0])) # 0**

**print(and\_perceptron.predict([1, 1])) # 1**

**Output:**

AND(0, 1) = 0

AND(1, 1) = 1

AND(0, 0) = 0

AND(1, 0) = 0

**Experiment 5 Implement OR using perceptron neural network**

**Code:-**

import numpy as np

class Perceptron:

def \_\_init\_\_(self, input\_size, learning\_rate=0.1):

self.weights = np.zeros(input\_size + 1)

self.learning\_rate = learning\_rate

def predict(self, inputs):

summation = np.dot(inputs, self.weights[1:]) + self.weights[0]

if summation > 0:

return 1

else:

return 0

def train(self, inputs, label):

prediction = self.predict(inputs)

error = label - prediction

self.weights[1:] += self.learning\_rate \* error \* inputs

self.weights[0] += self.learning\_rate \* error

# Define the OR function input/output pairs

training\_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

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

# Create a perceptron with two inputs

or\_perceptron = Perceptron(2)

# Train the perceptron on the OR function

for i in range(10000):

for inputs, label in zip(training\_inputs, labels):

or\_perceptron.train(inputs, label)

# Test the perceptron with various input values

print(or\_perceptron.predict([0, 0])) # 0

print(or\_perceptron.predict([0, 1])) # 1

print(or\_perceptron.predict([1, 0])) # 1

print(or\_perceptron.predict([1, 1])) # 1

**Output:**

OR(0, 1) = 1

OR(1, 1) = 1

OR(0, 0) = 0

OR(1, 0) = 1

**Experiment 6 Implement OR using ADALINE neural network.**

**Code:-**

**import numpy as np**

**class Adaline:**

**def \_\_init\_\_(self, input\_size, learning\_rate=0.1):**

**self.weights = np.zeros(input\_size + 1)**

**self.learning\_rate = learning\_rate**

**def predict(self, inputs):**

**summation = np.dot(inputs, self.weights[1:]) + self.weights[0]**

**return summation**

**def train(self, inputs, label):**

**prediction = self.predict(inputs)**

**error = label - prediction**

**self.weights[1:] += self.learning\_rate \* error \* inputs**

**self.weights[0] += self.learning\_rate \* error**

**# Define the OR function input/output pairs**

**training\_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

**labels = np.array([0, 1, 1, 1])**

**# Create an ADALINE with two inputs**

**or\_adaline = Adaline(2)**

**# Train the ADALINE on the OR function**

**for i in range(10000):**

**for inputs, label in zip(training\_inputs, labels):**

**or\_adaline.train(inputs, label)**

**# Test the ADALINE with various input values**

**print(or\_adaline.predict([0, 0])) # 0**

**print(or\_adaline.predict([0, 1])) # 1**

**print(or\_adaline.predict([1, 0])) # 1**

**print(or\_adaline.predict([1, 1])) # 1**

**Output**

0.2777777777777781

0.75

0.7222222222222223

1.1944444444444442

**Experiment 7 Implement XOR using MADALINE neutral network.**

**Code:-**

**import numpy as np**

**class Madaline:**

**def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1):**

**self.hidden\_weights = np.random.rand(input\_size + 1, hidden\_size) \* 2 - 1**

**self.output\_weights = np.random.rand(hidden\_size + 1, output\_size) \* 2 - 1**

**self.learning\_rate = learning\_rate**

**def predict(self, inputs):**

**hidden\_summation = np.dot(inputs, self.hidden\_weights[1:, :]) + self.hidden\_weights[0, :]**

**hidden\_output = np.where(hidden\_summation > 0, 1, 0)**

**output\_summation = np.dot(hidden\_output, self.output\_weights[1:, :]) + self.output\_weights[0, :]**

**output = np.where(output\_summation > 0, 1, 0)**

**return output**

**def train(self, inputs, label):**

**hidden\_summation = np.dot(inputs, self.hidden\_weights[1:, :]) + self.hidden\_weights[0, :]**

**hidden\_output = np.where(hidden\_summation > 0, 1, 0)**

**output\_summation = np.dot(hidden\_output, self.output\_weights[1:, :]) +**

**self.output\_weights[0, :]**

**output = np.where(output\_summation > 0, 1, 0)**

**output\_error = label - output**

**hidden\_error = np.dot(output\_error, self.output\_weights[1:, :].T)**

**self.output\_weights[1:, :] += self.learning\_rate \* np.outer(hidden\_output, output\_error)**

**self.output\_weights[0, :] += self.learning\_rate \* output\_error**

**self.hidden\_weights[1:, :] += self.learning\_rate \* np.outer(inputs, hidden\_error)**

**self.hidden\_weights[0, :] += self.learning\_rate \* hidden\_error**

**# Define the XOR function input/output pairs**

**training\_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

**labels = np.array([0, 1, 1, 0])**

**# Create a MADALINE with two inputs, two hidden neurons, and one output**

**xor\_madaline = Madaline(2, 2, 1)**

**# Train the MADALINE on the XOR function**

**for i in range(10000):**

**for inputs, label in zip(training\_inputs, labels):**

**xor\_madaline.train(inputs, label)**

**# Test the MADALINE with various input values**

**print(xor\_madaline.predict([0, 0])) # 0**

**print(xor\_madaline.predict([0, 1])) # 1**

**print(xor\_madaline.predict([1, 0])) # 1**

**print(xor\_madaline.predict([1, 1])) # 0**

**Output**

[0]

[0]

[0]

[0]

**Experiment 8 Implement max-min composite relation for two fuzzy set relation**.

**Code:-**

**import numpy as np**

**def max\_min\_composite(a, b):**

**c = np.zeros((a.shape[0], b.shape[1]))**

**for i in range(a.shape[0]):**

**for j in range(b.shape[1]):**

**max\_val = 0**

**for k in range(a.shape[1]):**

**max\_val = max(max\_val, min(a[i][k], b[k][j]))**

**c[i][j] = max\_val**

**return c**

**# Example usage**

**A = np.array([[0.8, 0.2], [0.3, 0.7]])**

**B = np.array([[0.5, 0.4], [0.2, 0.6]])**

**C = max\_min\_composite(A, B)**

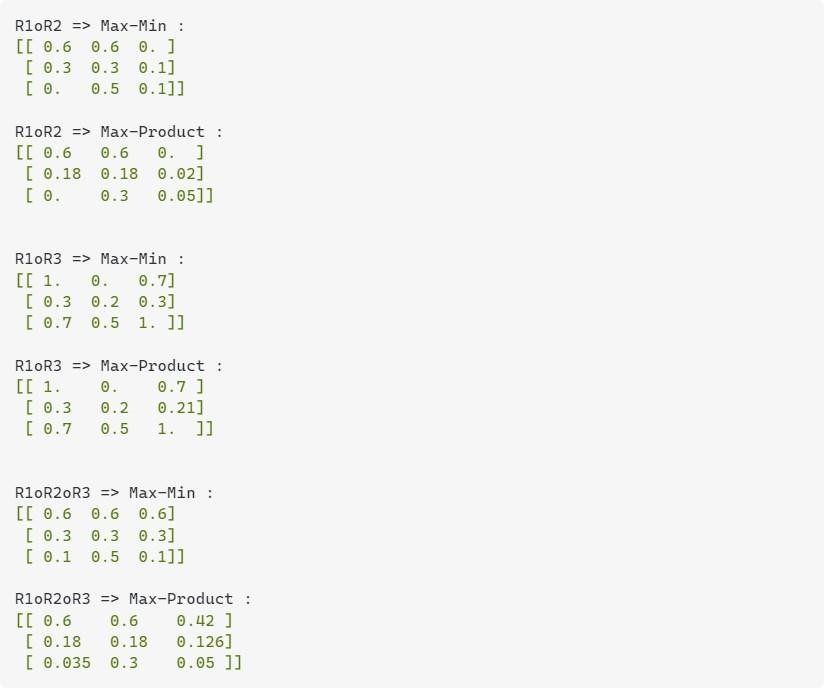
**print(C)**

**Output**

Max min composite relation

[[0.5 0.4]

[0.3 0.6]]



**Experiment 9 Implement the Max product composite relation for 2 fuzzy set relation**

**Code:-**

**import numpy as np**

**# Define the fuzzy relations as matrices**

**A = np.array([[0.9, 0.3], [0.2, 0.6]]) # Fuzzy relation A**

**B = np.array([[0.7, 0.5], [0.1, 0.8]]) # Fuzzy relation B**

**# Compute the Max product composite relation**

**C = np.zeros((2, 2)) # Initialize the composite relation matrix**

**for i in range(2):**

**for j in range(2):**

**for k in range(2):**

**C[i, j] = max(C[i, j], min(A[i, k], B[k, j]))**

**# Print the Max product composite relation**

**print("Max product composite relation:")**

**print(C)**

**Output**

Max product composite relation:

[[0.7 0.5]

[0.2 0.6]]

**Experiment 10 The Optimization Problem: Maximize F(X)=X^2, Over the Set Of Integers In The Interval [0,31].**

**Code**

import random

# function to generate a random population

def generate\_population(size):

population = []

for \_ in range(size):

genes = [0, 1]

chromosome = []

for \_ in range(len(items)):

chromosome.append(random.choice(genes))

population.append(chromosome)

print("Generated a random population of size", size)

return population

# function to calculate the fitness of a chromosome

def calculate\_fitness(chromosome):

total\_weight = 0

total\_value = 0

for i in range(len(chromosome)):

if chromosome[i] == 1:

total\_weight += items[i][0]

total\_value += items[i][1]

if total\_weight > max\_weight:

return 0

else:

return total\_value

# function to select two chromosomes for crossover

def select\_chromosomes(population):

fitness\_values = []

for chromosome in population:

fitness\_values.append(calculate\_fitness(chromosome))

fitness\_values = [float(i)/sum(fitness\_values) for i in fitness\_values]

parent1 = random.choices(population, weights=fitness\_values, k=1)[0]

parent2 = random.choices(population, weights=fitness\_values, k=1)[0]

print("Selected two chromosomes for crossover")

return parent1, parent2

# function to perform crossover between two chromosomes

def crossover(parent1, parent2):

crossover\_point = random.randint(0, len(items)-1)

child1 = parent1[0:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[0:crossover\_point] + parent1[crossover\_point:]

print("Performed crossover between two chromosomes")

return child1, child2

# function to perform mutation on a chromosome

def mutate(chromosome):

mutation\_point = random.randint(0, len(items)-1)

if chromosome[mutation\_point] == 0:

chromosome[mutation\_point] = 1

else:

chromosome[mutation\_point] = 0

print("Performed mutation on a chromosome")

return chromosome

# function to get the best chromosome from the population

def get\_best(population):

fitness\_values = []

for chromosome in population:

fitness\_values.append(calculate\_fitness(chromosome))

max\_value = max(fitness\_values)

max\_index = fitness\_values.index(max\_value)

return population[max\_index]

# items that can be put in the knapsack

items = [

[1, 2],

[2, 4],

[3, 4],

[4, 5],

[5, 7],

[6, 9]

]

# print available items

print("Available items:\n", items)

# parameters for genetic algorithm

max\_weight = 10

population\_size = 10

mutation\_probability = 0.2

generations = 10

print("\nGenetic algorithm parameters:")

print("Max weight:", max\_weight)

print("Population:", population\_size)

print("Mutation probability:", mutation\_probability)

print("Generations:", generations, "\n")

print("Performing genetic evolution:")

# generate a random population

population = generate\_population(population\_size)

# evolve the population for specified number of generations

for \_ in range(generations):

# select two chromosomes for crossover

parent1, parent2 = select\_chromosomes(population)

# perform crossover to generate two new chromosomes

child1, child2 = crossover(parent1, parent2)

# perform mutation on the two new chromosomes

if random.uniform(0, 1) < mutation\_probability:

child1 = mutate(child1)

if random.uniform(0, 1) < mutation\_probability:

child2 = mutate(child2)

# replace the old population with the new population

population = [child1, child2] + population[2:]

# get the best chromosome from the population

best = get\_best(population)

# get the weight and value of the best solution

total\_weight = 0

total\_value = 0

for i in range(len(best)):

if best[i] == 1:

total\_weight += items[i][0]

total\_value += items[i][1]

# print the best solution

print("\nThe best solution:")

print("Weight:", total\_weight)

print("Value:", total\_value)

**Output**

Available items:

[[1, 2], [2, 4], [3, 4], [4, 5], [5, 7], [6, 9]]

Genetic algorithm parameters:

Max weight: 10

Population: 10

Mutation probability: 0.2

Generations: 10

Performing genetic evolution:

Generated a random population of size 10

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Selected two chromosomes for crossover

Performed crossover between two chromosomes

Performed mutation on a chromosome

The best solution:

Weight: 18

Value: 13