Practical Lab File

(2022-2023)

Soft Computing

DEPARTMENT OF COMPUTER ENGINEERING & APPLICATIONS

INSTITUTE OF ENGINEERING & TECHNOLOGY

Submitted To:

Dr. Manmohan Singh Sir

(Assistant Professor)

Submitted By:

Name - Surendra Pratap Tomar

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]]
```

```
R1oR2 => Max-Min :
[[ 0.6  0.6  0. ]
 [ 0.3  0.3  0.1]
 [ 0.   0.5  0.1]]

R1oR2 => Max-Product :
[[ 0.6   0.6   0.  ]
 [ 0.18  0.18  0.02]
 [ 0.    0.3   0.05]]


R1oR3 => Max-Min :
[[ 1.   0.   0.7]
 [ 0.3  0.2  0.3]
 [ 0.7  0.5  1. ]]

R1oR3 => Max-Product :
[[ 1.    0.    0.7 ]
 [ 0.3   0.2   0.21]
 [ 0.7   0.5   1.  ]]


R1oR2oR3 => Max-Min :
[[ 0.6  0.6  0.6]
 [ 0.3  0.3  0.3]
 [ 0.1  0.5  0.1]]

R1oR2oR3 => Max-Product :
[[ 0.6    0.6    0.42 ]
 [ 0.18   0.18   0.126]
 [ 0.035  0.3    0.05 ]]
```

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
```