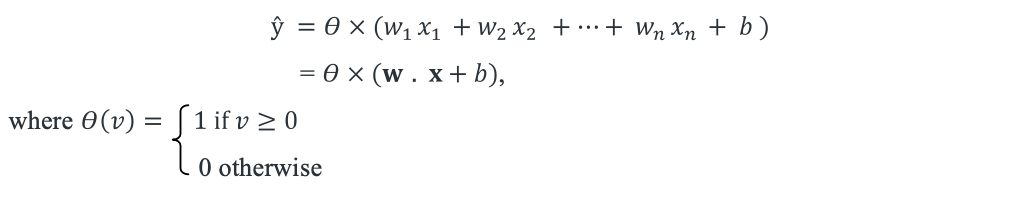
**EXPERIMENT-1**

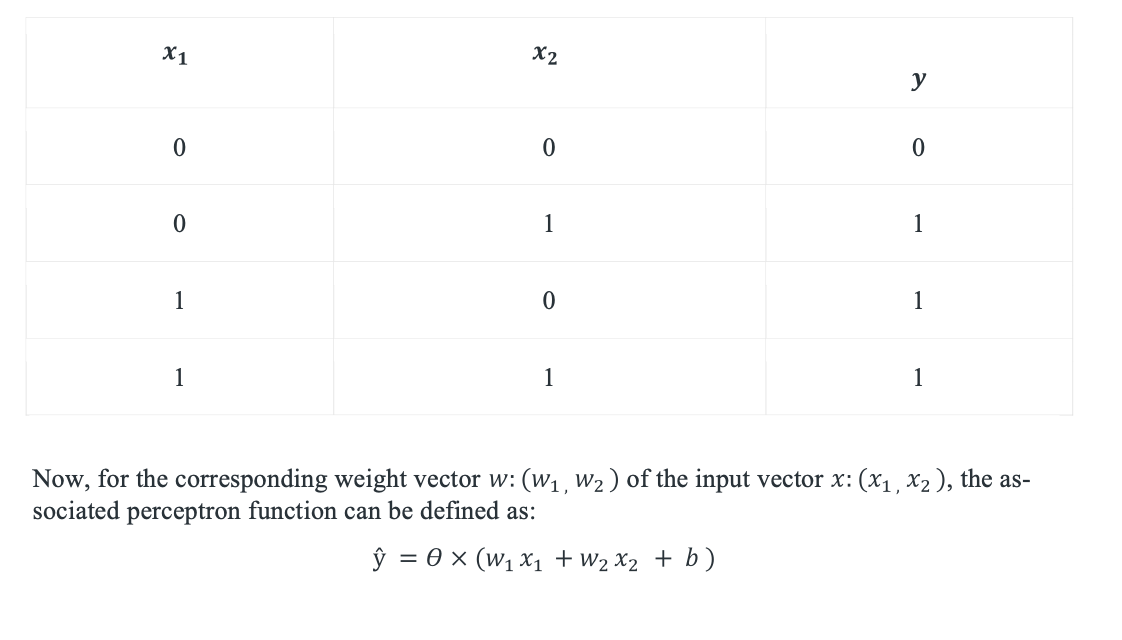
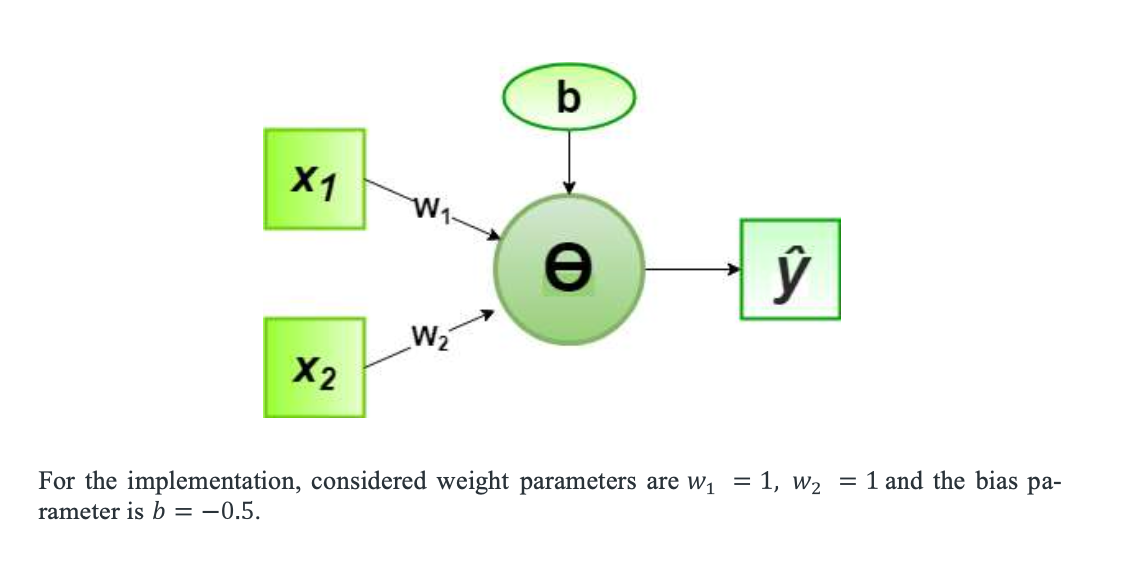
**Aim**: Write a program for implementation of Perceptron algorithm for OR logic gate with 2-bit binary input.

**Theory :**

In the field of Machine Learning, the Perceptron is a Supervised Learning Algorithm for binary classifiers. The Perceptron Model implements the following function:



For a particular choice of the weight vector w and bias parameter b, the model predicts output for the corresponding input vector x.

OR logical function truth table for 2-bit binary variables, i.e. the input vector and the corresponding output, y

**Python Script :**

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

if v >= 0:

return 1

else:

return 0

# design Perceptron Model

def perceptronModel(x, w, b):

v = np.dot(w, x) + b

y = unitStep(v)

return y

# OR Logic Function

# w1 = 1, w2 = 1, b = -0.5 def OR\_logicFunction(x):

def OR\_logicFunction(x):

w = np.array([1, 1])

b = -0.5

return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

print("OR({}, {}) = {}".format(0, 1, OR\_logicFunction(test1)))

print("OR({}, {}) = {}".format(1, 1, OR\_logicFunction(test2)))

print("OR({}, {}) = {}".format(0, 0, OR\_logicFunction(test3)))

print("OR({}, {}) = {}".format(1, 0, OR\_logicFunction(test4)))

**Output:**

OR(0, 1) = 1

OR(1, 1) = 1

OR(0, 0) = 0

OR(1, 0) = 1

\*\* Process exited - Return Code: 0 \*\*

Press Enter to exit terminal

**Conclusion :** Successfully demonstrated e a program for implementation of Perceptron algorithm for OR logic gate with 2-bit binary input.

**VIVA VOICE**

**Question 1: What is the Perceptron algorithm?**

**Answer**  : The Perceptron algorithm is a supervised learning algorithm used for binary classification tasks. It's a type of linear classifier that makes predictions based on a linear combination of input features weighted by learned parameters.

**Question 2: How does the Perceptron algorithm work?**

**Answer**  : The algorithm works by iteratively updating the weights of the input features until the model converges to a solution. It calculates the weighted sum of the input features, applies an activation function (often a step function), and compares the predicted output to the actual output to adjust the weights accordingly.

**Question 3: What is the role of the activation function in the Perceptron algorithm?**

**Answer**  :The activation function determines the output of the Perceptron based on whether the weighted sum of inputs is above or below a certain threshold. It introduces non-linearity to the model, allowing it to learn complex patterns in the data.

**Question 4: What is the purpose of the bias term in the Perceptron algorithm?**

**Answer**  : The bias term allows the Perceptron to account for situations where all input features are zero but the desired output is not zero. It shifts the decision boundary away from the origin and enables the model to learn more complex patterns.

**Question 5: What are some limitations of the Perceptron algorithm?**

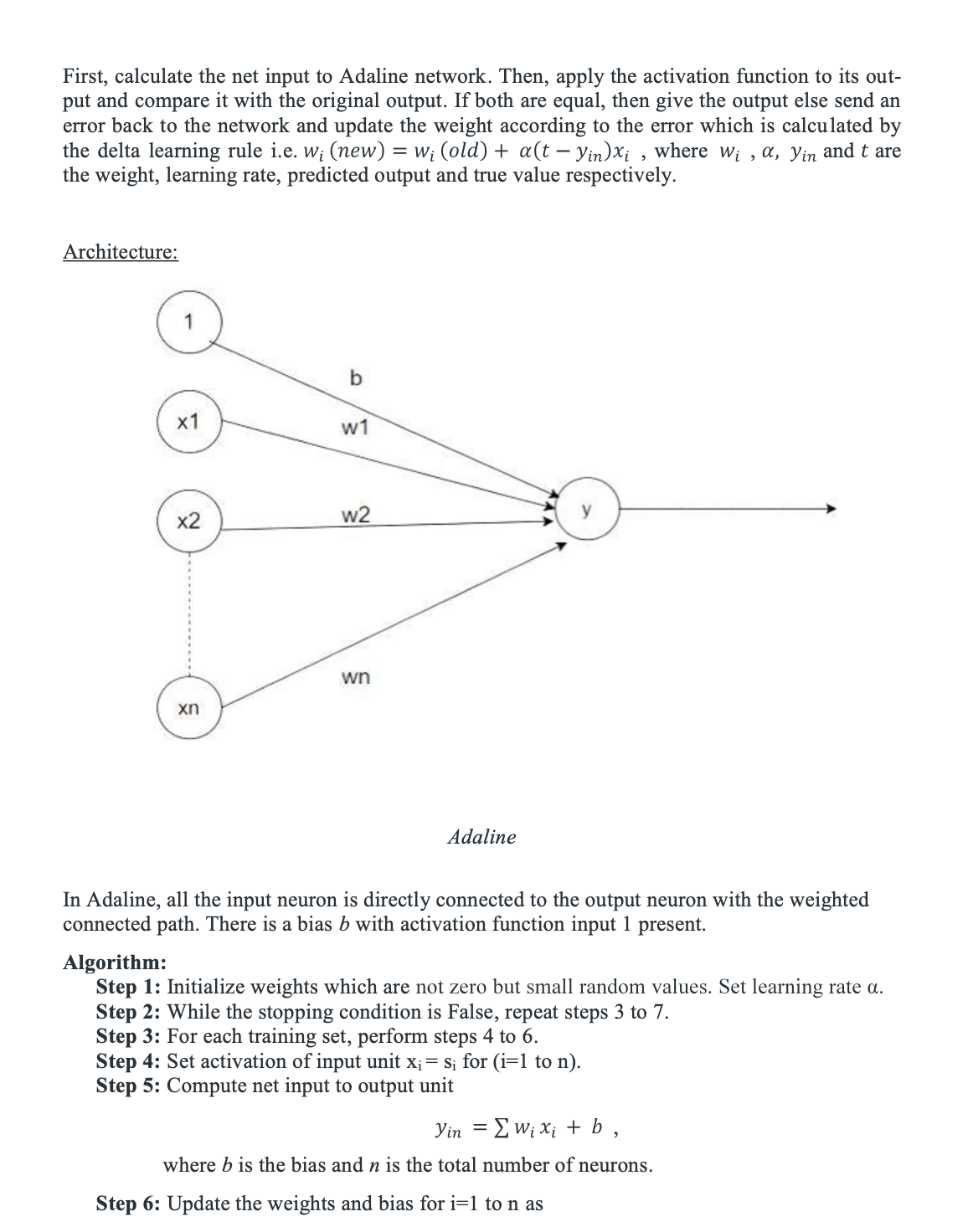
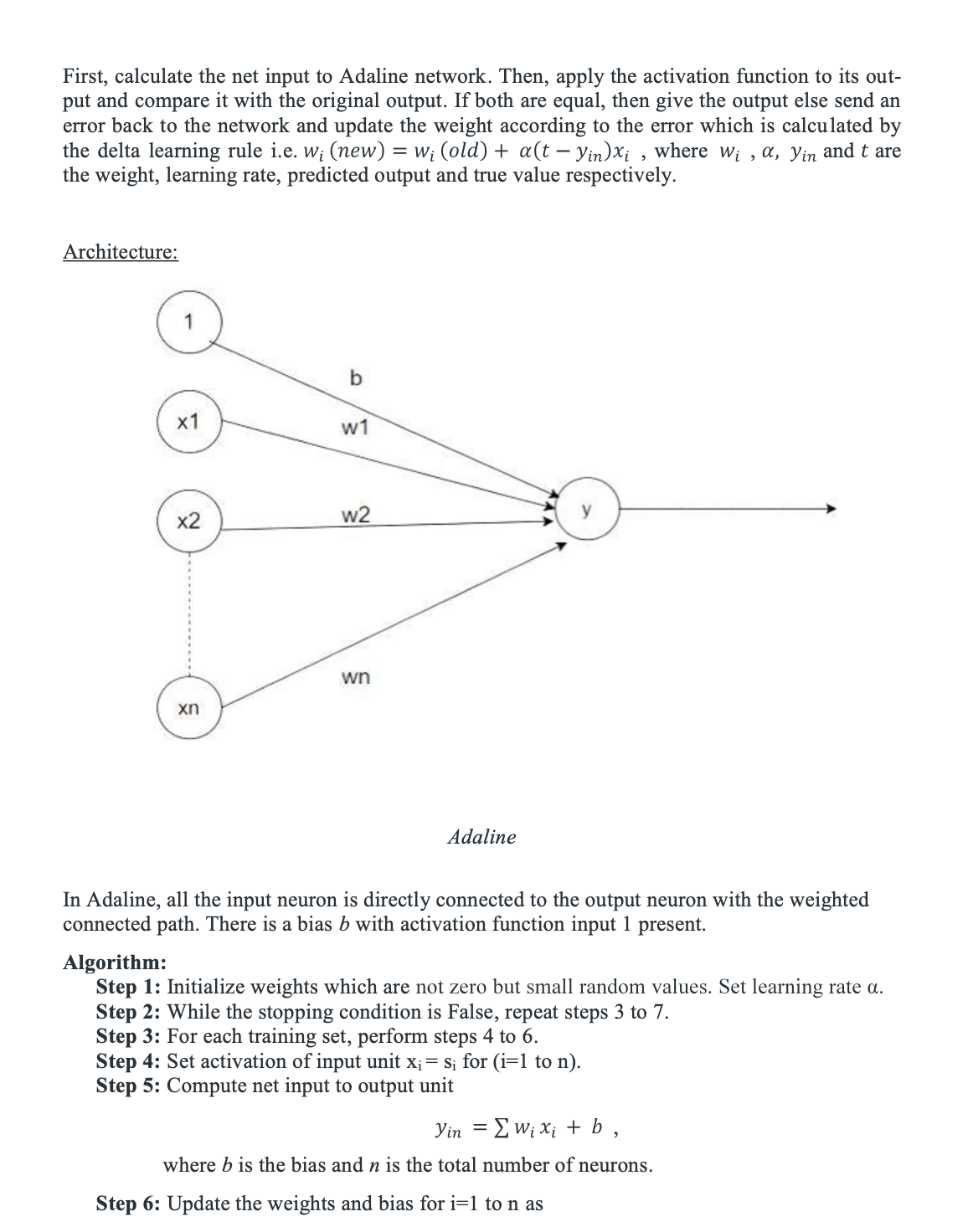
**Answer**  : The Perceptron algorithm can only learn linearly separable patterns, meaning it cannot solve problems that are not linearly separable. It also converges only if the data is linearly separable. Additionally, it may require many iterations to converge, and the choice of the learning rate can significantly affect its performance.

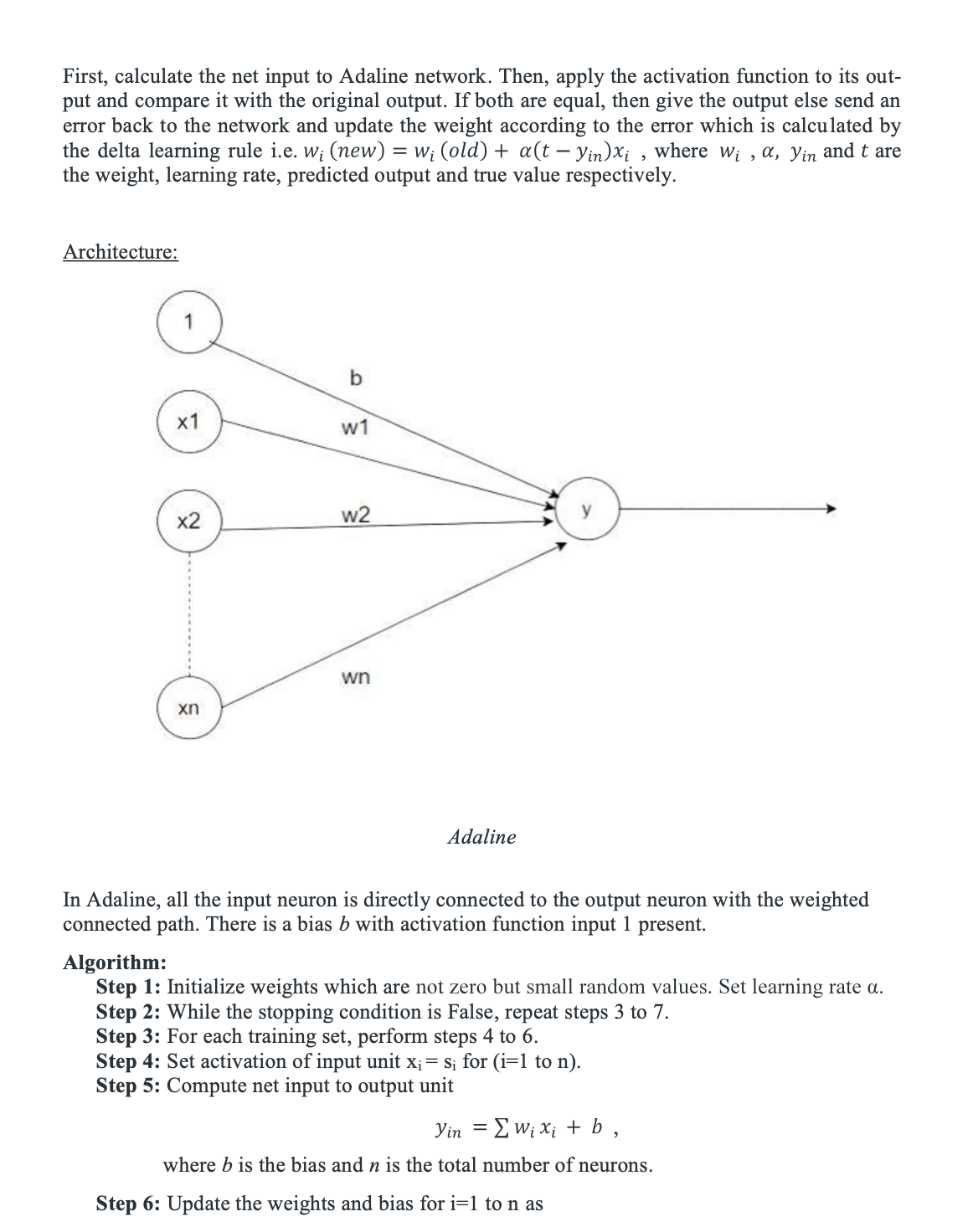
**EXPERIMENT 2**

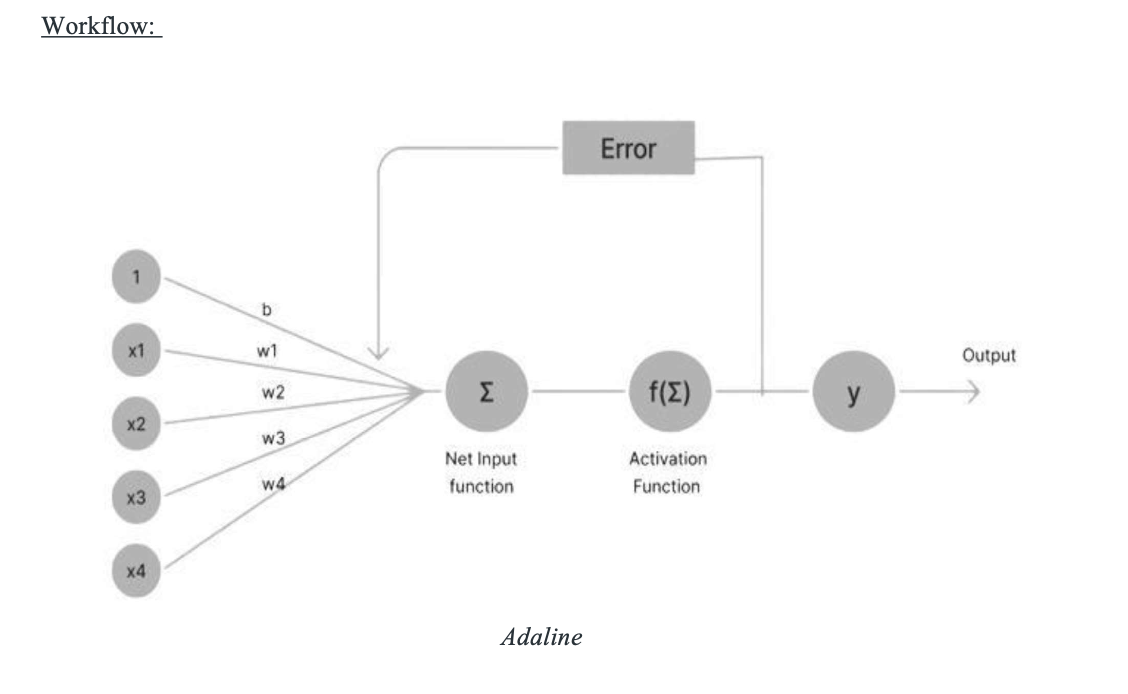
**Aim:** Write a program to create a simple ADALINE network for implementing OR logic gate with 2-bit binary inputs and one output node. Train it using delta learning rule until no change in weights is required. Output the final weights.

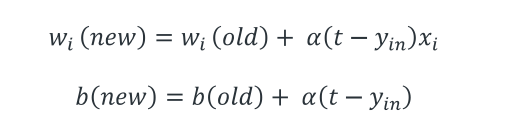
**Theory:**

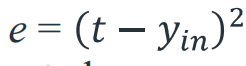
The two basic types of neural networks are (i.) Adaline (Adaptive Linear Neural) which doesn’t have any hidden layer, and (ii.) Madaline (Multiple Adaptive Linear Neural) which has one hidden layer. In this experiment, we are discussing the Adaline network.

Adaline - A network with a single linear unit is called Adaline. A unit with a linear activation function (such as sigmoid, ReLu etc.) is called a linear unit. In Adaline, there is only one out- put unit and output values are bipolar (+1,-1). Weights between the input unit and output unit are adjustable. It uses the delta rule, i.e. , where wi ,and are the weight, learning rate, predicted output, and true value respectively. The learning rule is found to minimize the mean square error between activation and target values. Adaline consists of trainable weights, it compares actual output with calculated output, and based on error, training algorithm is applied.





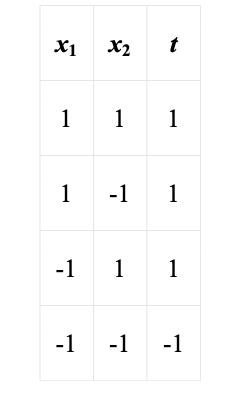


and calculate the squared-error, i.e. When the predicted output and the true value are same then the weight will not change.

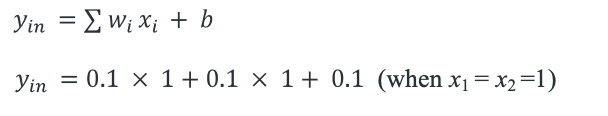
Test the stopping condition. The stopping condition may be when the weight changes at a low rate or there is no change in the weight.

Implementation

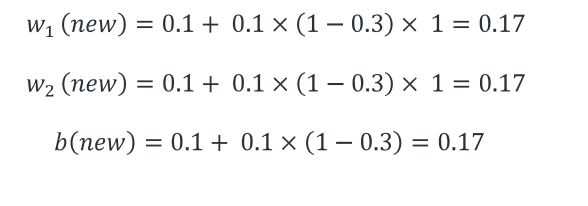
* Initially, all weights are assumed to be small random values, say 0.1, and the learning rate is set to 0.1.
* Also, the least squared error is set to 0.001.
* The weights will be updated until the total error is greater than the least squared error.



* Calculate the net input

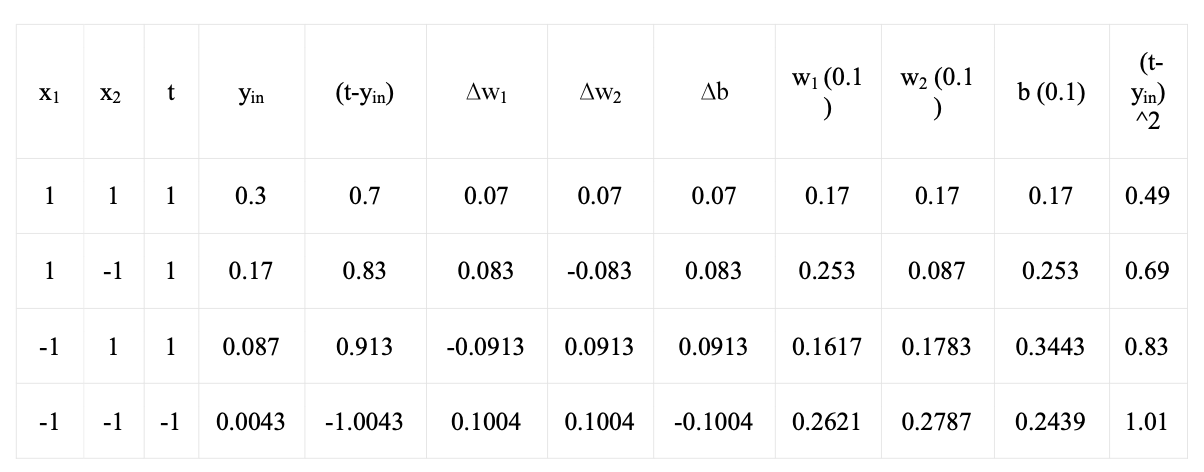


* Now compute, (t-yin)=(1-0.3)=0.7
* Now, update the weights and bias



* Calculate the error as: 

Similarly, repeat the same steps for other input vectors and we will get.



This is epoch 1 where the total error is 0.49 + 0.69 + 0.83 + 1.01 = 3.02. So, more epochs will run until the total error becomes less than equal to the least squared error i.e 0.001.

**Python Script:**

# Import necessary libraries

import numpy as np

# Adaline neural network

def Adaline(Input, Target, lr, stop):

weight = np.random.random(Input.shape[1])

bias = np.random.random(1)

Error = [stop + 1]

# check the stop condition for the network

while Error[-1] > stop or Error[-1] - Error[-2] > 0.0001:

error = []

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

Y\_input = sum(weight \* Input[i]) + bias

# Update the weight

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

weight[j] = weight[j] + lr \* (Target[i] - Y\_input) \* Input[i][j]

print('weight :', weight)

# Update the bias

bias = bias + lr \* (Target[i] - Y\_input)

# Store squared error value

error.append((Target[i] - Y\_input) \*\* 2) # Store sum of square errors Error.append(sum(error))

# Store sum of square errors Error.append(sum(error))

print('Error :', Error[-1])

return weight, bias

# Input dataset

x = np.array([[1.0, 1.0, 1.0],

[1.0, -1.0, 1.0],

[-1.0, 1.0, 1.0],

[-1.0, -1.0, -1.0]])

# Target values

t = np.array([1, 1, 1, -1])

w, b = Adaline(x, t, lr=0.1, stop=0.001)

print('Bias :', b)

# Predict from the evaluated weight and bias of adaline

def prediction(X,w,b):

y=[]

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

x = X[i]

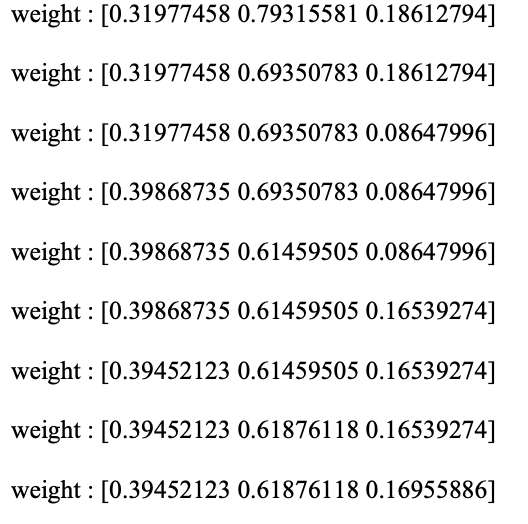
y.append(sum(w\*x)+b)

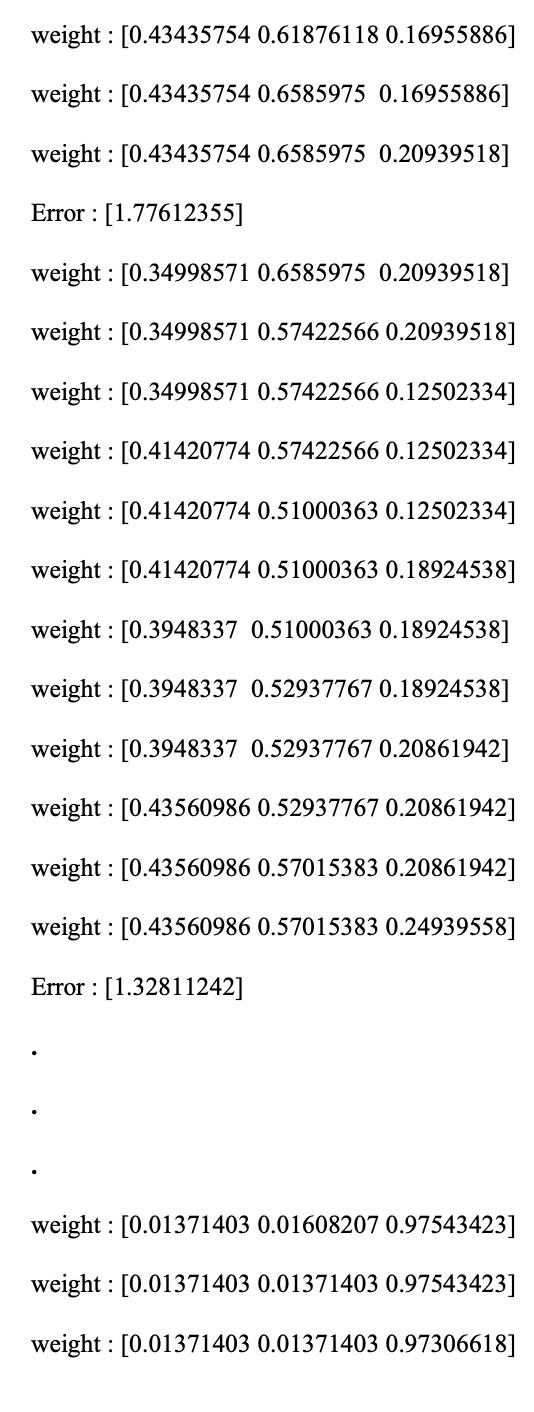
return y

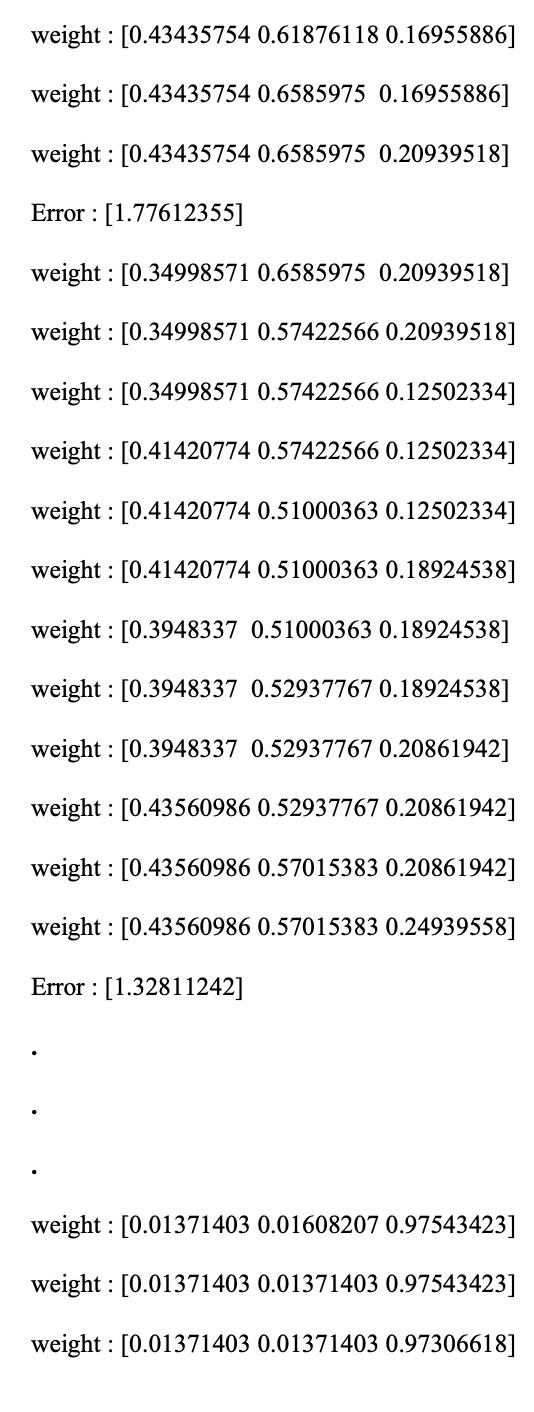
print('Prediction is')

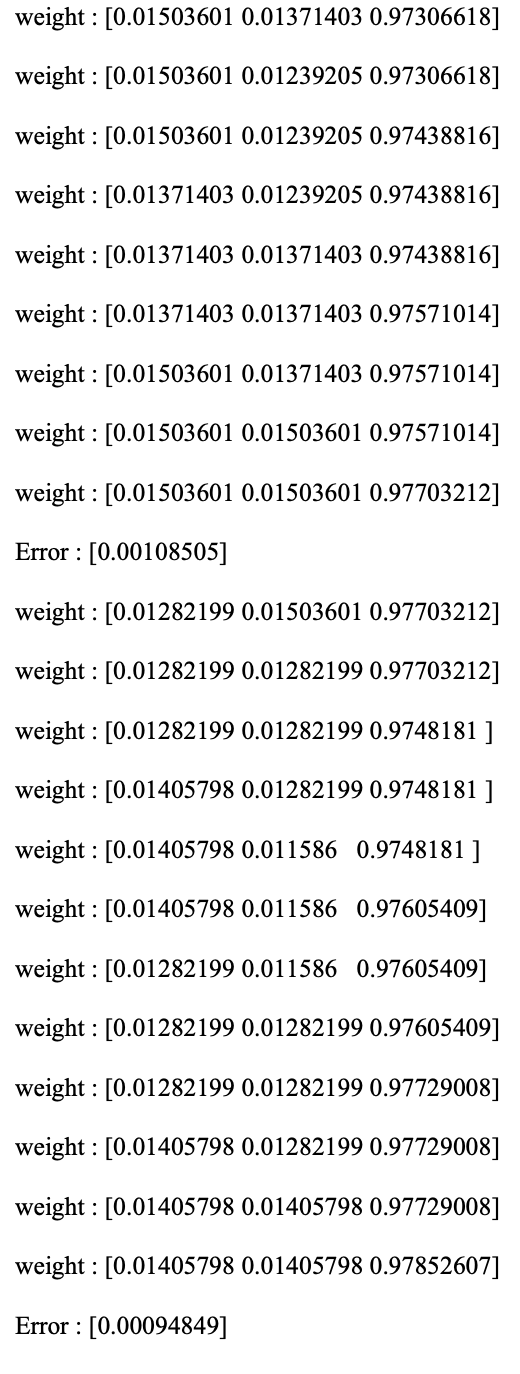
print(prediction(x,w,b))

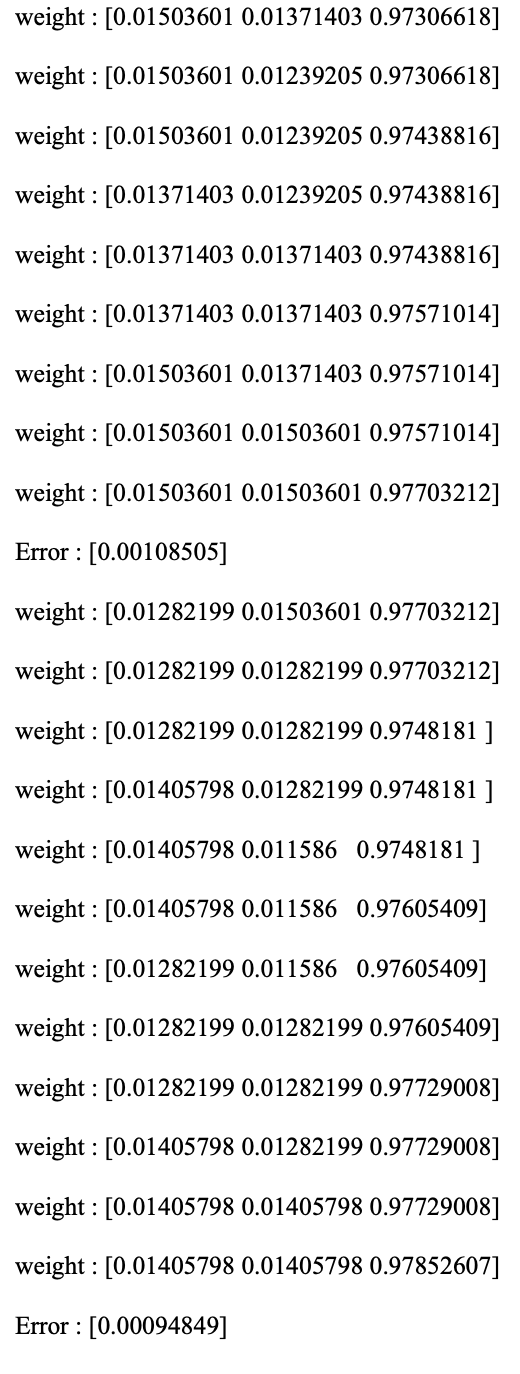
**Output:**

****









Bias : [0.01405798]

Prediction is

[array([1.02070003]), array([0.99258406]), array([0.99258406]), array([-0.99258406])]

**Conclusion :** Successfully demonstrated a program to create a simple ADALINE network for implementing OR logic gate with 2-bit binary inputs and one output node. Train it using delta learning rule until no change in weights is required. Output the final weights.

**VIVA VOICE**

**Question 1: What is ADALINE?**

**Answer:** ADALINE stands for Adaptive Linear Neuron or Adaptive Linear Element. It's a type of single-layer artificial neural network with one or more inputs and a single output. ADALINE adjusts its weights iteratively to minimize the difference between its output and the desired output using a learning rule.

**Question 2 : What is the delta learning rule?**

**Answer:** The delta learning rule is a method used to adjust the weights of a neural network iteratively in order to minimize the error between the actual output and the desired output. It's commonly employed in supervised learning scenarios, where the network learns from a dataset with known input-output pairs. The rule calculates the change in weights based on the difference between the actual output and the desired output, and then adjusts the weights accordingly.

**Question 3 : How does the ADALINE network differ from the perceptron?**

**Answer:** While both ADALINE and the perceptron are single-layer neural networks, they differ in their activation functions and learning algorithms. The perceptron uses a step function as its activation function and employs the perceptron learning rule, which only works for linearly separable problems. On the other hand, ADALINE uses a linear activation function and employs the delta learning rule, which can handle both linear and non-linear problems through continuous adjustment of weights.

**Question 4 : What is the significance of the OR logic gate in this experiment?**

**Answer:** The OR logic gate is one of the fundamental building blocks of digital logic circuits. It produces a true output (1) if at least one of its inputs is true (1). Implementing the OR logic gate using an ADALINE network demonstrates the capability of neural networks to learn and mimic simple logical functions. It also provides a basic example for understanding how neural networks can be trained to perform specific tasks through weight adjustments.

**Question 5 : How do you determine when no change in weights is required during training?**

**Answer:** In the context of this experiment, no change in weights is required when the difference between the actual output of the network and the desired output becomes negligible or falls below a predefined threshold. This typically indicates that the network has learned the task sufficiently well and further adjustments to the weights would not significantly improve performance. It's a termination condition often used in iterative learning algorithms to stop the training process once the desired accuracy or convergence is achieved.

**EXPERIMENT 3**

**Aim:** Write a program to implement a simple Fuzzy neural network (FNN) for an air conditioning temperature control system. The FNN will take two inputs: temperature and humidity, and provide a fuzzy output representing the air conditioner's strength (weak, moderate, strong).

**Theory:**

**Introduction**

Fuzzy Neural Networks (FNNs) represent a powerful hybrid computational model that integrates the concepts of fuzzy logic and artificial neural networks. FNNs are designed to handle complex and uncertain data by leveraging the strengths if both fuzzy logic’s ability to deal with imprecise information and neural network’s capability to learn from data. This essay explores the fundamental principles, architecture, advantages, and applications of Fuzzy Neural Networks.

**Understanding Fuzzy Neural Networks**

Fuzzy logic is a mathematical approach that allows variables to possess partial truth values be- tween 0 and 1, representing degrees of membership in a given set. This flexibility makes fuzzy logic an ideal tool for modeling and handling uncertainty and imprecision in various real-world applications. On the other hand, artificial neural networks are inspired by the human brain’s structure and learning capabilities. They consist of interconnected nodes (neurons) arranged in layers, and through an iterative process, they can adapt and learn from input-output data pairs.

Fuzzy Neural Networks aim to combine the advantages of both fuzzy logic and neural networks to create a more robust and flexible computational system. The primary components of an FNN include fuzzification of input data, a fuzzy rule base, a neural network structure, and defuzzifica- tion of the output.

**Architecture of Fuzzy Neural Networks**

1. **Fuzzification**: The process of converting crisp (numerical) input data into fuzzy sets. Mem- bership functions are used to determine the degree of membership for each data point in the fuzzy sets.

2. **Fuzzy Rule Base**: This component stores the linguistic rules that link fuzzy input variables to fuzzy output variables. These rules are typically in the form of “if-then” statements and express relationships between input and output variables.

3. **Neural Network Structure**: The FNN incorporates an artificial neural network that takes the fuzzy input variables and processes them through hidden layers. The hidden layers perform computations to combine the fuzzy rules and extract underlying patterns from the input data.

4. **Defuzzification**: The final step involves converting the fuzzy output generated by the neural network into crisp numerical values. Various defuzzification methods, such as the center of gravity, are employed to obtain the final output.

**Advantages of Fuzzy Neural Networks**

1. **Handling Uncertainty**: FNNs can effectively deal with uncertain and imprecise data, making them well-suited for applications in domains with incomplete information or ambiguous data.

2. **Non-linearity: Fuzzy** Neural Networks can approximate complex, nonlinear relationships be- tween input and output variables, surpassing the capabilities of traditional linear models.

3. **Learning and Adaptation**: By incorporating neural network architecture, FNNs can learn from data, adapt to changing conditions, and continuously improve their performance over time.

4. **Interpretability**: Unlike black-box machine learning models, Fuzzy Neural Networks employ fuzzy rules that can be easily interpreted and understood by domain experts, enhancing the model’s transparency.

**Applications of Fuzzy Neural Networks**

Fuzzy Neural Networks have found applications in a wide range of fields:

1. **Pattern Recognition**: FNNs can effectively recognize patterns in image and speech process- ing, where the data may be noisy or imprecise.

2. **Control Systems**: Fuzzy Neural Networks are used in control systems to create adaptive and robust controllers capable of handling uncertain environments.

3. **Forecasting and Decision Making**: FNNs are employed in predicting financial markets, weather patterns, and other uncertain systems, assisting in better decision-making.

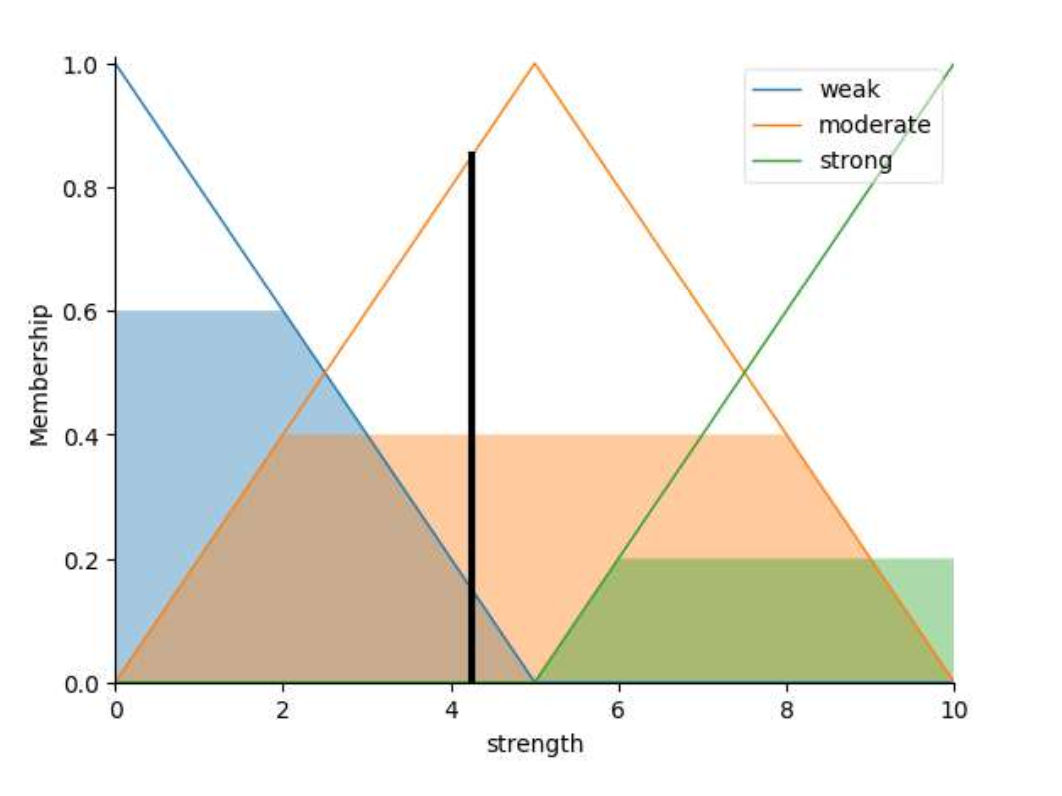
4. **Data Mining**: Fuzzy Neural Networks aid in extracting useful information from large data- sets, particularly when dealing with incomplete or fuzzy data.

**Code**

Below is a complete Python code implementation of a Fuzzy Neural Network (FNN) using the scikit-fuzzy library. This example demonstrates a simple FNN for a temperature control sys- tem. The FNN will take two inputs: temperature and humidity, and provide a fuzzy output repre- senting the air conditioner's strength (weak, moderate, strong).

In this example, we define three fuzzy sets each for the input variables (temperature and humid- ity) and the output variable (strength). We also define the rules for the FNN, which represent the fuzzy logic that governs the relationship between inputs and output. The rules use linguistic terms (cold, moderate, hot, low, moderate, high, weak, moderate, strong) to represent fuzzy conditions and conclusions.

The usage example sets the input values (temperature and humidity) to 80 and 40, respectively, and computes the corresponding output (air conditioner strength). Finally, the output is printed along with a visual representation of the fuzzy result using the strength.view(sim=air\_conditioner) statement.



Please note that this is a simplified example, and real-world Fuzzy Neural Networks may involve more inputs, outputs, and complex rule bases. The scikit-fuzzy library provides various tools to build and simulate more sophisticated FNNs.

**Python Script:**

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# Input variables

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

humidity = ctrl.Antecedent(np.arange(0, 101, 1), 'humidity')

# Output variable

strength = ctrl.Consequent(np.arange(0, 11, 1), 'strength')

# Membership functions for input variables

temperature['cold'] = fuzz.trimf(temperature.universe, [0, 0, 50])

temperature['moderate'] = fuzz.trimf(temperature.universe, [0, 50, 100])

temperature['hot'] = fuzz.trimf(temperature.universe, [50, 100, 100])

humidity['low'] = fuzz.trimf(humidity.universe, [0, 0, 50])

humidity['moderate'] = fuzz.trimf(humidity.universe, [0, 50, 100])

humidity['high'] = fuzz.trimf(humidity.universe, [50, 100, 100])

# Membership functions for output variable

strength['weak'] = fuzz.trimf(strength.universe, [0, 0, 5])

strength['moderate'] = fuzz.trimf(strength.universe, [0, 5, 10])

strength['strong'] = fuzz.trimf(strength.universe, [5, 10, 10])

# Rules

rule1 = ctrl.Rule(temperature['cold'] | humidity['low'], strength['strong'])

rule2 = ctrl.Rule(temperature['moderate'] & humidity['moderate'], strength['moderate'])

rule3 = ctrl.Rule(temperature['hot'] | humidity['high'], strength['weak'])

# Control System

ac\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

air\_conditioner = ctrl.ControlSystemSimulation(ac\_ctrl)

# Example usage

# Input values (temperature, humidity) air\_conditioner.input['temperature'] = 80 air\_conditioner.input['humidity'] = 40

# Compute the result

air\_conditioner.compute()

# Print the output

print("Air Conditioner Strength:", air\_conditioner.output['strength'])

strength.view(sim=air\_conditioner)

**Output**:

Air Conditioner Strength: 4.253968253968254

Process finished with exit code 0

**Conclusion:**

Fuzzy Neural Networks represent a compelling synergy between fuzzy logic and artificial neural networks, providing a powerful tool for handling uncertainty and complex relationships in real- world applications. Their ability to learn from data, interpretability, and capacity to address chal- lenging problems with imprecise information make them a valuable asset in the realm of artificial intelligence and computational intelligence. As technology continues to evolve, Fuzzy Neural Networks will likely play an essential role in addressing increasingly complex and uncertain prob- lems across various domains.

**VIVA VOICE**

**Question 1: What is a Fuzzy Neural Network (FNN)?**

**Answer:** A Fuzzy Neural Network (FNN) is a type of neural network that integrates fuzzy logic principles with neural network architectures. It combines the ability of neural networks to learn from data with the flexibility of fuzzy logic in handling imprecise or uncertain information. FNNs are particularly useful for systems where inputs and outputs may not have precise numerical values, such as in temperature control systems.

**Question 2: How does an FNN handle fuzzy inputs and outputs?**

**Answer:** In an FNN, fuzzy inputs and outputs are represented using linguistic variables and fuzzy sets. Linguistic variables describe qualitative terms (e.g., "hot," "cold," "moderate") and fuzzy sets assign degrees of membership to these terms based on the input or output values. The network then learns to associate input fuzzy sets with output fuzzy sets through training, allowing it to make decisions or predictions based on fuzzy input information.

**Question 3: What are the advantages of using an FNN for air conditioning temperature control?**

**Answer:** One advantage is the ability of FNNs to handle imprecise or uncertain inputs, such as temperature and humidity readings, which may not have precise numerical values. FNNs can also capture the complex relationships between these inputs and the desired air conditioner strength output, allowing for more flexible and adaptive control systems compared to traditional methods.

**Question 4: How would you design the fuzzy membership functions for temperature and humidity inputs in the FNN?**

**Answer:** The design of fuzzy membership functions involves defining linguistic variables (e.g., "low," "medium," "high") and determining how input values map to degrees of membership in each fuzzy set. This can be done based on domain knowledge, data analysis, or experimentation. For temperature, for example, membership functions could include "cold," "comfortable," and "hot," with appropriate thresholds for each. Similarly, humidity membership functions could include "dry," "moderate," and "humid."

**Question 5: How is the output of the FNN interpreted in the context of air conditioning control?**

Answer: The output of the FNN represents the recommended strength of the air conditioner, which is typically categorized into fuzzy sets such as "weak," "moderate," and "strong." The degree of membership in each output fuzzy set indicates the level of recommendation for that strength. For instance, if the output fuzzy set "strong" has a high degree of membership, it suggests that the air conditioner should operate at a strong strength to maintain comfort conditions based on the input temperature and humidity.

**EXPERIMENT - 4**

**Aim:** Write a program to implement a travelling salesman problem (TSP) using genetic algorithm (GA).

**Theory:**

A genetic algorithm is presented here to solve thetravelling salesman problem. Genetic algorithms are heuristic search algorithms inspired by the process that supports the evolution of life. The algorithm is designed to replicate the natural selection process to carry generation, i.e. survival of the fittest of beings. Standard genetic algorithms are divided into five phases which are:

1. Creating initial population.

2. Calculating fitness.

3. Selecting the best genes.

4. Crossing over.

5. Mutating to introduce variations.

These algorithms can be implemented to find a solution to the optimization problems of vari- ous types. One such problem is the traveling Salesman Problem . The problem says that a salesman is given a set of cities, he has to find the shortest route to as to visit each city exactly once and return to the starting city.

**Approach:**

In the following implementation, cities are taken as genes, string generated using these characters is called a chromosome, while a fitness score which is equal to the path length of all the cities mentioned, is used to target a population.

Fitness Score is defined as the length of the path described by the gene. Lesser the path length fitter is the gene. The fittest of all the genes in the gene pool survive the population test and move to the next iteration. The number of iterations depends upon the value of a cooling vari- able. The value of the cooling variable keeps on decreasing with each iteration and reaches a threshold after a certain number of iterations.

**Algorithm:**

1. Initialize the population randomly.

2. Determine the fitness of the chromosome.

3. Until done repeat:

1. Select parents.

2. Perform crossover and mutation.

3. Calculate the fitness of the new population.

4. Append it to the gene pool.

**Pseudo-code**

Initialize procedure GA{

Set cooling parameter = 0;

Evaluate population P(t);

While( Not Done ){

Parents(t) = Select\_Parents(P(t));

Offspring(t) = Procreate(P(t));

p(t+1) = Select\_Survivors(P(t), Offspring(t));

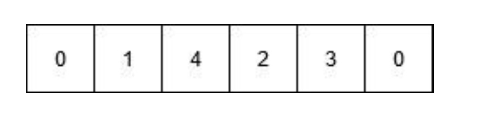
t = t + 1;

}

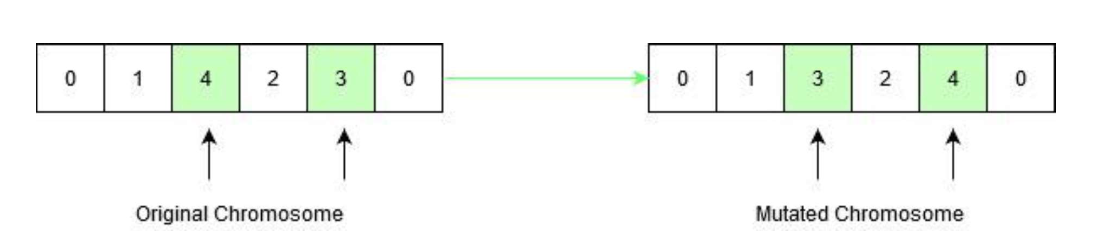
}

**How the mutation works?**

Suppose there are 5 cities: 0, 1, 2, 3, 4. The salesman is in city 0 and he has to find the shortest route to travel through all the cities back to the city 0. A chromosome representing the path chosen can be represented as:



This chromosome undergoes mutation. During mutation, the position of two cities in the chro- mosome is swapped to form a new configuration, except the first and the last cell, as they rep- resent the start and endpoint.



Original chromosome had a path length equal to INT\_MAX, according to the input defined below, since the path between city 1 and city 4 did’nt exist. After mutation, the new child formed has a path length equal to 21, which is a much-optimized answer than the original assumption. This is how the genetic algorithm optimizes solutions to hard problems. Below is the implementation of the above approach:

**Python Script:**

from random import randint

INT\_MAX = 2147483647

# Number of cities in TSP V=5

# Names of the cities

GENES = "ABCDE"

# Starting Node Value

START = 0

# Initial population size for the algorithm

POP\_SIZE = 10

# Structure of a GNOME

# defines the path traversed

# by the salesman while the fitness value # of the path is stored in an integer

class individual:

def \_\_init\_\_(self) -> None:

self.gnome = ""

self.fitness = 0

def \_\_lt\_\_(self, other):

return self.fitness < other.fitness

def \_\_gt\_\_(self, other):

return self.fitness > other.fitness

# Function to return a random number # from start and end

def rand\_num(start, end):

return randint(start, end - 1)

# Function to check if the character # has already occurred in the string def repeat(s, ch):

for i in range(len(s)):

if s[i] == ch:

return True

return False

# Function to return a mutated GNOME

# Mutated GNOME is a string

# with a random interchange

# of two genes to create variation in species def mutatedGene(gnome):

def mutatedGene(gnome):

gnome = list(gnome)

while True:

r = rand\_num(1, V)

r1 = rand\_num(1, V)

if r1 != r:

temp = gnome[r]

gnome[r] = gnome[r1]

gnome[r1] = temp

break

return ''.join(gnome)

# Function to return a valid GNOME string

# required to create the population

def create\_gnome():

gnome = "0"

while True:

if len(gnome) == V:

gnome += gnome[0]

break

temp = rand\_num(1, V)

if not repeat(gnome, chr(temp + 48)):

gnome += chr(temp + 48)

return gnome

# Function to return the fitness value of a gnome.

# The fitness value is the path length

# of the path represented by the GNOME.

def cal\_fitness(gnome):

mp = [

[0, 2, INT\_MAX, 12, 5], [2, 0, 4, 8, INT\_MAX], [INT\_MAX, 4, 0, 3, 3], [12, 8, 3, 0, 10],

[5, INT\_MAX, 3, 10, 0],

]

f=0

for i in range(len(gnome) - 1):

if mp[ord(gnome[i]) - 48][ord(gnome[i + 1]) - 48] == INT\_MAX:

return INT\_MAX

f += mp[ord(gnome[i]) - 48][ord(gnome[i + 1]) - 48]

return f

# Function to return the updated value

# of the cooling element.

def cooldown(temp):

return (90 \* temp) / 100

# Utility function for TSP problem.

def TSPUtil(mp):

# Generation Number

gen = 1

# Number of Gene Iterations

gen\_thres = 5

population = []

temp = individual()

# Populating the GNOME pool.

for i in range(POP\_SIZE):

temp.gnome = create\_gnome()

temp.fitness = cal\_fitness(temp.gnome)

population.append(temp)

print("\nInitial population: \nGNOME FITNESS VALUE\n")

for i in range(POP\_SIZE):

print(population[i].gnome, population[i].fitness)

print()

found = False

temperature = 10000

# Iteration to perform

# population crossing and gene mutation.

while temperature > 1000 and gen <= gen\_thres:

population.sort()

print("\nCurrent temp: ", temperature)

new\_population = []

for i in range(POP\_SIZE):

p1 = population[i]

while True:

new\_g = mutatedGene(p1.gnome)

new\_gnome = individual()

new\_gnome.gnome = new\_g

new\_gnome.fitness = cal\_fitness(new\_gnome.gnome)

if new\_gnome.fitness <= population[i].fitness:

new\_population.append(new\_gnome)

break

else:

# Accepting the rejected children at

# a possible probability above threshold.

prob = pow(2.7, -1 \* ((float)(new\_gnome.fitness - population[i].fitness)/temperature),)

if prob > 0.5:

new\_population.append(new\_gnome)

break

temperature = cooldown(temperature)

population = new\_population

print("Generation", gen)

print("GNOME FITNESS VALUE")

for i in range(POP\_SIZE):

print(population[i].gnome, population[i].fitness)

gen += 1

if \_\_name\_\_ == "\_\_main\_\_":

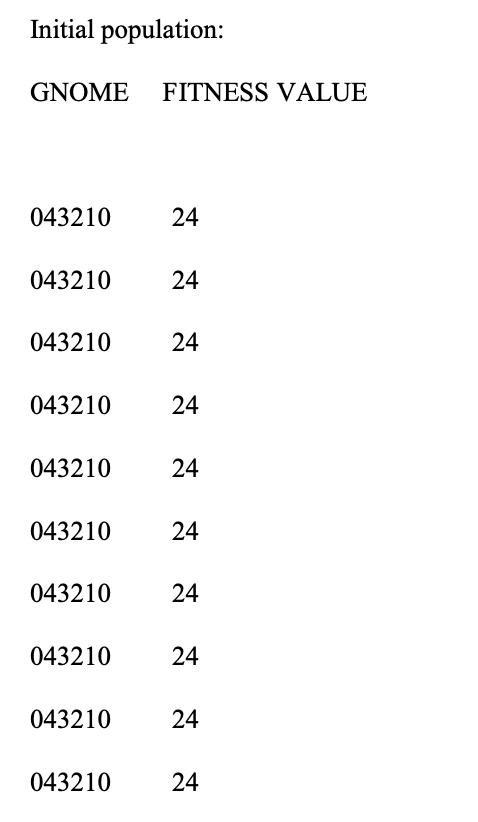
mp = [

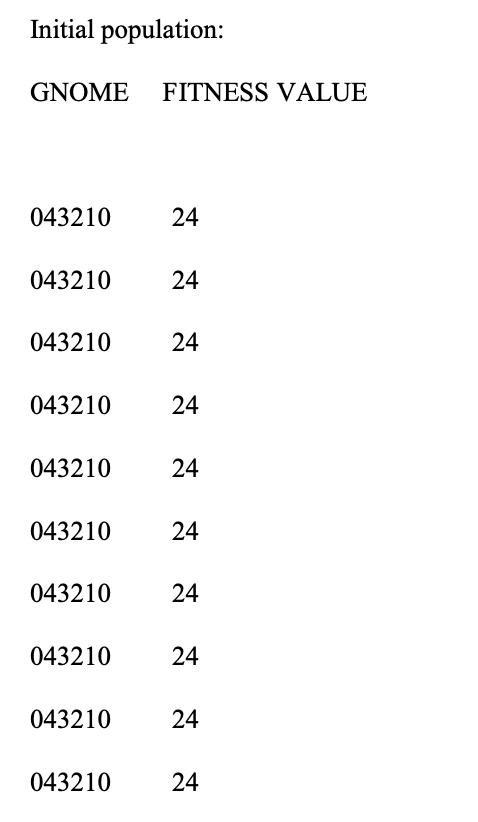
[0, 2, INT\_MAX, 12, 5], [2, 0, 4, 8, INT\_MAX], [INT\_MAX, 4, 0, 3, 3], [12, 8, 3, 0, 10],

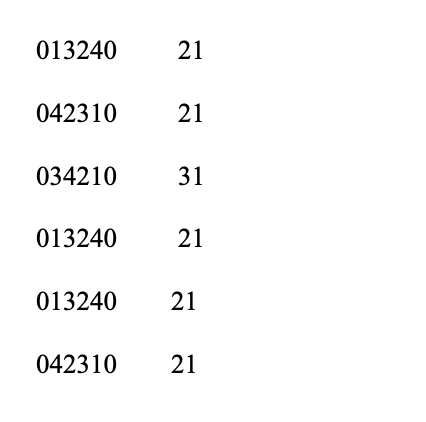
[5, INT\_MAX, 3, 10, 0], ]

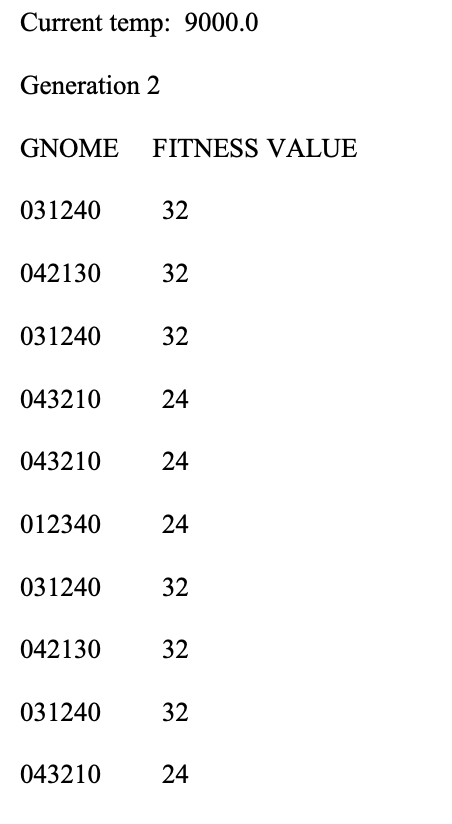
TSPUtil(mp)

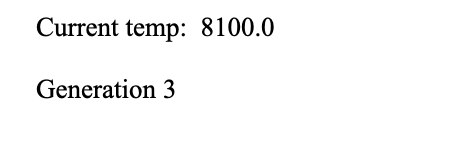
**Output:**

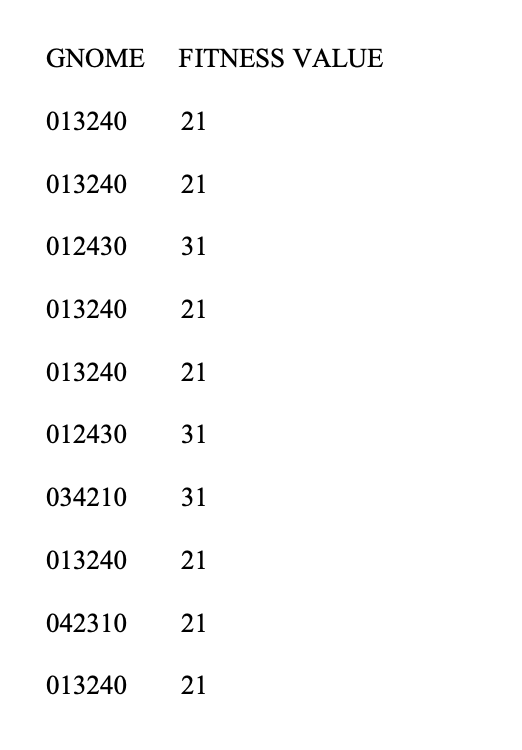


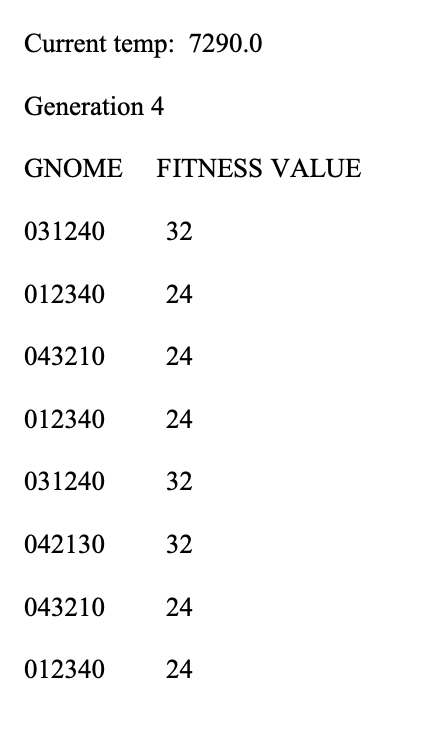


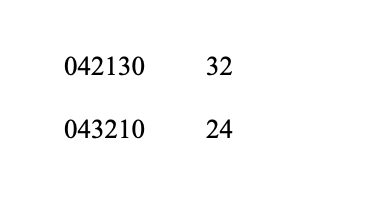


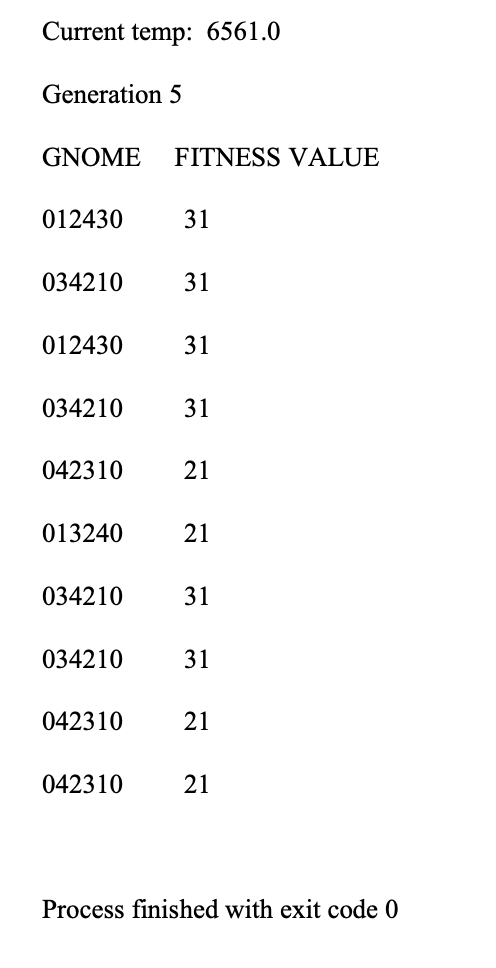












**Conclusion:** Successfully demonstrated program that implement a travelling salesman problem (TSP) using genetic algorithm (GA).

**VIVA VOICE**

**Question 1: What is the Travelling Salesman Problem (TSP)?**

**Answer:** The Travelling Salesman Problem is a classic problem in computer science and combinatorial optimization. It involves finding the shortest possible route that visits each city exactly once and returns to the origin city. The problem is NP-hard, meaning that there is no known efficient algorithm to solve it optimally for large problem instances.

**Question 2: How does a Genetic Algorithm (GA) solve the TSP?**

**Answer:** A Genetic Algorithm is a metaheuristic inspired by the process of natural selection. To solve the TSP, a GA typically represents each candidate solution (or route) as a chromosome in a population. The algorithm then iteratively evolves the population through processes like selection, crossover (recombination), mutation, and elitism. Over generations, the algorithm converges towards better solutions, often finding near-optimal or good solutions for the TSP.

**Question 3: What are the key components of a Genetic Algorithm for the TSP?**

**Answer:** The key components include:

* Representation: Each candidate solution (route) is represented as a chromosome, typically a permutation of cities.
* Fitness Function: Evaluates the quality of each solution by calculating the total distance of the route.
* Selection: Selects individuals (solutions) from the population for reproduction based on their fitness.
* Crossover: Combines genetic material from selected individuals to create offspring.
* Mutation: Introduces small random changes to offspring to maintain diversity.
* Elitism: Preserves the best solutions from each generation to ensure the algorithm converges towards better solutions.

**Question 4: How do you prevent premature convergence in a GA for the TSP?**

**Answer:** Premature convergence occurs when the algorithm settles on a suboptimal solution without sufficiently exploring the search space. To prevent this, several techniques can be employed, such as:

* Diverse Initial Population: Ensure the initial population contains a diverse set of solutions.
* Dynamic Parameters: Adjust parameters like mutation rate or population size over time to encourage exploration.
* Local Search: Apply local search heuristics to exploit promising regions of the search space.
* Niching: Encourage diversity by penalizing similar solutions or introducing niching strategies.

**EXPERIMENT 5**

**Aim:** A simple python code implementing associative memory (Correlator) neural network to train and store 4 digits.

**Theory:**

An associative memory network can store a set of patterns as memories. When the associative memory is being presented with a key pattern, it responds by producing one of the stored patterns, which closely resembles or relates to the key pattern. Thus, the recall is through association of the key pattern, with the help of information memorized. These types of memories are also called as content-addressable memories (CAM). The CAM can also be viewed as associating data to address, i.e.; for every data in the memory there is a corresponding unique address. Also, it can be viewed as data correlator. Here input data is correlated with that of the stored data in the CAM. It should be noted that the stored patterns must be unique, i.e., different patterns in each location. If the same pattern exists in more than one location in the CAM, then, even though the correlation is correct, the address is noted to be ambiguous. Associative memory makes a parallel search within a stored data file. The concept behind this search is to Output any one or all stored items Which match the given search argument.

**Python Script:**

import numpy as np

def training(): #trains on patterns

final\_weights = np.zeros((35,35))

patterns = []

for g in range(4):

fname = str(g+1)+'.txt'

arr = np.loadtxt(fname)

lol = []

for k in range(7):

megalol = arr[k]

lol.extend(megalol)

#print len(lol)

patterns.append(lol)

print ('one pattern trained!') for j in range(len(lol)):

lol[j] = int(lol[j])

#print lol

lol = np.matrix(lol)

lolt = lol.T

multx = np.dot(lolt,lol)

final\_weights = np.add(final\_weights,multx)

#print final\_weights,'\n'

return final\_weights, patterns

def match\_pattern(patx): pattern

#matches a new pattern with an existing

for l in range(4):

matchwith = patterns[l]

if np.array\_equal(matchwith,patx) == True:

print ('matched with pattern ', l+1)

return l+1

return 0

def check\_pattern(patt): sponding output

patt2 = np.matrix(patt)

'training()'

#print patt2.shape

generated = []

for t in range(35):

hola = final\_weights[:,t]

hola2 = np.matrix(hola)

#print hola2.shape

matx = np.dot(patt2,hola2)

#print 'mm>', matx

if matx>0:

generated.append(1)

else:

generated.append(-1)

return generated

def check\_newpattern(fnamex):

present from file

brr = np.loadtxt(fnamex)

lol = []

for k in range(7):

megalol = brr[k]

lol.extend(megalol)

tacobell = check\_pattern(lol)

numx = match\_pattern(tacobell)

return numx

#directly checks if a pattern if

final\_weights,patterns = training()

print ('...trained on......',len(patterns),'patterns!')

correct = 0

for f in range(4):

gotit = check\_pattern(patterns[f])

numx = match\_pattern(gotit)

if numx!= 0:

correct+=1

print ('matched pattern',numx) print ('correct=',correct)

if correct==4:

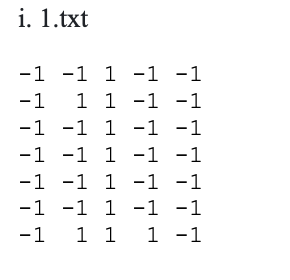
print ('all trained patterns can be retrieved!\n\nSUCCESS!')

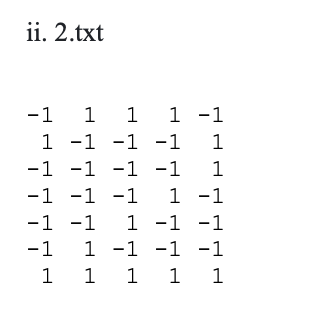
print ('\n\n')

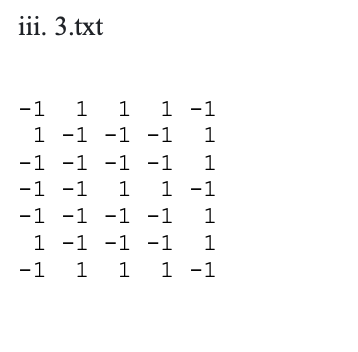
print ('matched',check\_newpattern('2\_distorted1.txt')) #one noise

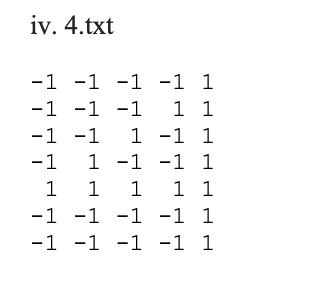
print ('matched',check\_newpattern('2\_distorted2.txt')) #two noise

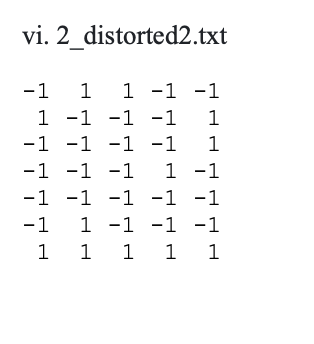
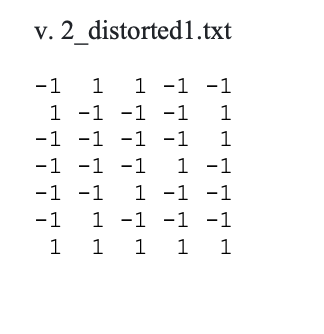
Text files:











**Output:**

one pattern trained!

one pattern trained!

one pattern trained!

one pattern trained! ...trained on...... 4 patterns!

matched with pattern 1

matched pattern 1

matched with pattern 2

matched pattern 2

matched with pattern 3

matched pattern 3

matched with pattern 4

matched pattern 4

correct= 4

all trained patterns can be retrieved!

SUCCESS!

matched with pattern 2 matched 2

matched 0

**Conclusion:** Successfully demonstrated a simple python code implementing associative memory (Correlator) neural network to train and store 4 digits.

**VIVA VOICE**

**Question 1: What is an Associative Memory Neural Network, specifically the Correlator model?**

**Answer:** The Correlator model is a type of associative memory neural network designed to store and recall patterns based on similarity. It associates input patterns with output patterns by forming connections between them during training. Given a partial or noisy input pattern, the network retrieves the closest matching stored pattern as the output.

**Question 2: How does the training process work in a Correlator neural network?**

**Answer:** During training, the Correlator network learns to associate input patterns with corresponding output patterns by adjusting its connection weights. The network calculates the correlation between the input and output patterns and updates the weights based on this correlation. This process continues until the network achieves satisfactory performance in recalling the stored patterns.

**Question 3: What are the limitations of the Correlator model in storing and recalling patterns?**

**Answer:** One limitation is that the Correlator model can only recall patterns similar to those it has been trained on. It may struggle with patterns that are significantly different or noisy compared to the stored patterns. Additionally, the network's performance may degrade if the number of stored patterns becomes too large, leading to overlap or interference between patterns.

**Question 4:** How do you measure the performance of a Correlator neural network in recalling stored patterns?

**Answer:** The performance of the Correlator network can be measured using metrics such as accuracy, precision, and recall. Accuracy measures the percentage of correctly recalled patterns out of the total recall attempts. Precision measures the ratio of correctly recalled patterns to the total number of recalled patterns, while recall measures the ratio of correctly recalled patterns to the total number of stored patterns.

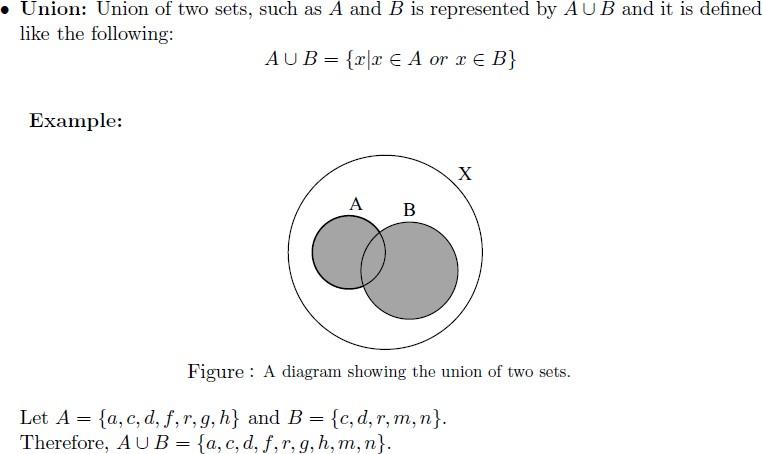
**Question 5:** Can you describe a real-world application where a Correlator neural network might be useful?

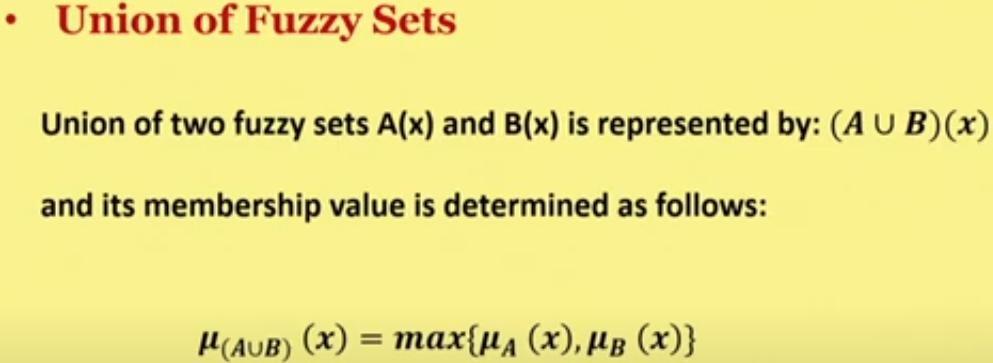
**Answer:** One potential application is in pattern recognition systems, such as recognizing handwritten digits or characters. The Correlator network could be trained to store a set of reference patterns representing different digits or characters and then used to classify new input patterns based on their similarity to the stored references. This could be useful in optical character recognition (OCR) systems or signature verification systems, among others.

**EXPERIMENT 6**

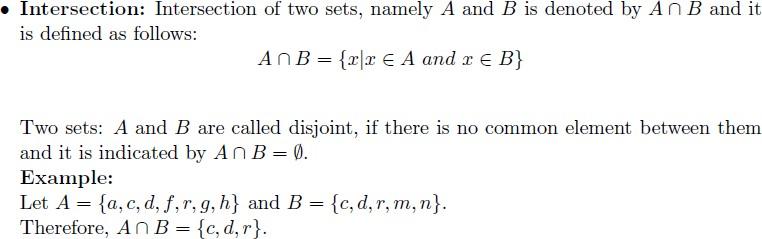
**Aim:** Write a Program for implementing operations (Union, Intersection, Complement, Difference) on Fuzzy sets.

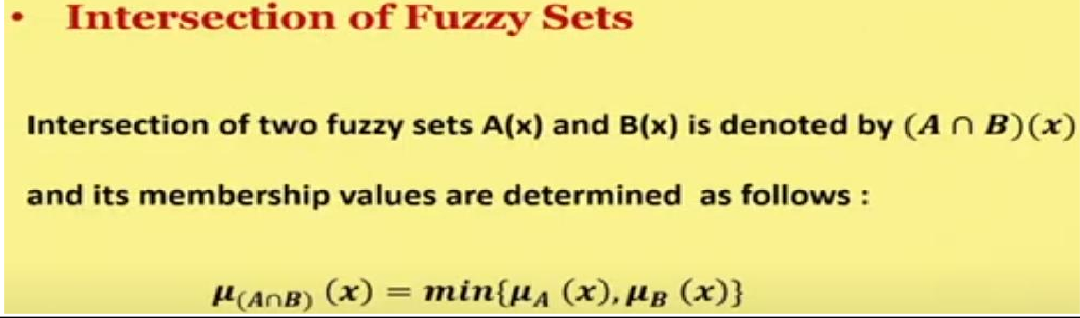
**Theory:**

Union of Crisp (Classical) sets:



Intersection of Crisp (Classical) sets





### 

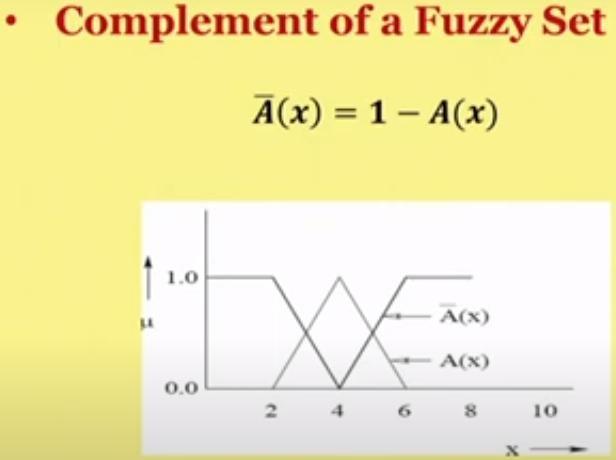
### Complement of Crisp (Classical) sets:

### Fuzzy complement is identical to [crisp complement operation](https://codecrucks.com/crisp-set-operations/). The membership value of every element in the fuzzy set is complemented with respect to 1, i.e. it is subtracted from 1.

The **complement**of fuzzy set A, denoted by AC, is defined as

AC = {(x, μAC (x)) | ∀x ∈ X}

AC (x) = 1 – μA(x)



**Python script for *Union operation* on fuzzy sets:**

# Example to Demonstrate the

# Union of Two Fuzzy Sets

A = dict()

B = dict()

Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)

print('The Second Fuzzy Set is :', B)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

if A\_value > B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Union is :', Y)

**Output:**

The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}

The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}

Fuzzy Set Union is : {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}

**Python script for finding Intersection of two fuzzy sets:**

# Example to Demonstrate

# Intersection of Two Fuzzy Sets

A = dict()

B = dict()

Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)

print('The Second Fuzzy Set is :', B)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

if A\_value < B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Intersection is :', Y)

**Output:**

The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}

The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}

Fuzzy Set Intersection is : {'a': 0.2, 'b': 0.3, 'c': 0.4, 'd': 0.5}

**Python script for finding the complement of a fuzzy set**

# Example to Demonstrate the

# Complement Two Fuzzy Sets

A = dict()

Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

print('The Fuzzy Set is :', A)

for A\_key in A:

Y[A\_key]= 1-A[A\_key]

print('Fuzzy Set Complement is :', Y)

**Output:**

The Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}

Fuzzy Set Complement is : {'a': 0.8, 'b': 0.7, 'c': 0.4, 'd': 0.4}

**Python script for finding the difference of a fuzzy set:**

# Example to Demonstrate the

# Difference Between Two Fuzzy Sets

A = dict()

B = dict()

Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)

print('The Second Fuzzy Set is :', B)

for A\_key, B\_key in zip(A, B):

A\_value = A[A\_key]

B\_value = B[B\_key]

B\_value = 1 - B\_value

if A\_value < B\_value:

Y[A\_key] = A\_value

else:

Y[B\_key] = B\_value

print('Fuzzy Set Difference is :', Y)

**Output**:

The First Fuzzy Set is : {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

The Second Fuzzy Set is : {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

Fuzzy Set Difference is : {"a": 0.1, "b": 0.1, "c": 0.6, "d": 0.5}

**Conclusion** : Successfully demonstrated the operations on fuzzy sets.

**VIVA VOICE**

**Question 1: What are fuzzy sets, and how do they differ from classical sets?**

**Answer 1:** Fuzzy sets extend the concept of classical sets by allowing elements to have partial membership rather than just binary membership (either fully belonging or not belonging). In a fuzzy set, each element has a membership grade between 0 and 1, indicating the degree to which the element belongs to the set.

**Question 2: How do you represent fuzzy sets in a programming context?**

**Answer 2:** Fuzzy sets can be represented in a programming context using various data structures. One common approach is to use arrays or dictionaries to store the membership grades of elements. Each element is paired with its corresponding membership grade, indicating the degree of membership in the fuzzy set.

**Question 3: What are the operations that can be performed on fuzzy sets?**

**Answer 3:** The main operations on fuzzy sets include union, intersection, complement, and difference. These operations are analogous to their counterparts in classical set theory but are performed on the membership grades of elements rather than the elements themselves.

**Question 4: How do you implement the union operation on fuzzy sets?**

**Answer 4:** To implement the union operation on fuzzy sets, you take the maximum membership grade for each element from the two fuzzy sets being unioned. This ensures that the resulting fuzzy set contains elements that have at least the same degree of membership in either of the original sets.

**Question 5: What are some practical applications of fuzzy set operations?**

**Answer 5:** Fuzzy set operations find applications in various fields such as decision making, control systems, and pattern recognition. For example, in decision making, fuzzy set operations can be used to combine fuzzy rules or preferences. In control systems, they can help in modeling uncertainty and handling imprecise inputs. In pattern recognition, fuzzy set operations can aid in feature selection and classification.

**EXPERIMENT 7**

**Aim :** Implementation of Fuzzy Relations(Max-min Composition)

**Theory :**

The max-min composition is a fundamental operation in fuzzy logic and set theory, particularly in fuzzy relations. In fuzzy set theory, membership degrees can range from 0 to 1, highlighting the importance of membership in fuzzy relations. The max operation takes the maximum degree of membership for each pair of elements (x, z) in the resulting relation, emphasizing the most significant degree of relationship between x and z. The min operation takes the minimum degree of membership for each pair of elements (x, z) in the resulting relation, considering the weakest link in the chain. This composition rule is widely used in fuzzy logic to capture both the strongest and weakest connections between elements, providing a balanced approach to composition.

**Python Script :**

import numpy as np

# Max-Min Composition given by Zadeh

def maxMin(x, y):

z = []

for x1 in x:

for y1 in y.T:

z.append(max(np.minimum(x1, y1)))

return np.array(z).reshape((x.shape[0], y.shape[1]))

# 3 arrays for the example

r1 = np.array([[1, 0, .7], [.3, .2, 0], [0, .5, 1]])

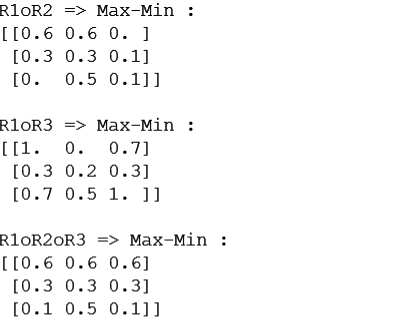
r2 = np.array([[.6, .6, 0], [0, .6, .1], [0, .1, 0]])

r3 = np.array([[1, 0, .7], [0, 1, 0], [.7, 0, 1]])

print ("R1oR2 => Max-Min :\n" + str(maxMin(r1, r2)) + "\n")

print ("R1oR3 => Max-Min :\n" + str(maxMin(r1, r3)) + "\n")

print ("R1oR2oR3 => Max-Min :\n" + str(maxMin(r1, maxMin(r2, r3))) + "\n")

**Output :**

**Conclusion :** Successfully demonstrated the Max-min Composition in Fuzzy relations.

**VIVA VOICE**

**Question 1: What is a fuzzy relation, and how does it differ from a crisp (classical) relation?**

**Answer:** A fuzzy relation extends the concept of a crisp (classical) relation by allowing the degree of association between elements of two sets to be represented by membership grades rather than binary values. In a fuzzy relation, each pair of elements from the two sets has a degree of membership indicating the strength of the relationship between them.

**Question 2: What is the Max-min composition method used for in fuzzy relations?**

**Answer:** The Max-min composition method is used to combine two fuzzy relations to obtain a new fuzzy relation. It calculates the degree of membership for each pair of elements in the resulting fuzzy relation by taking the maximum of the minimum membership grades obtained from the compositions of elements through a common intermediary set.

**Question 3: How do you implement the Max-min composition method in practice?**

**Answer:** To implement the Max-min composition method, you iterate through all possible combinations of elements from the two fuzzy relations being composed. For each combination, you find the minimum membership grade obtained by composing the elements through a common intermediary set. Then, you take the maximum of these minimum membership grades to determine the degree of membership for the pair of elements in the resulting fuzzy relation.

**Question 4: What are some advantages of using fuzzy relations and Max-min composition?**

**Answer:** Fuzzy relations and Max-min composition provide a flexible framework for modeling and reasoning with uncertain or imprecise information. They allow for the representation of complex relationships between elements with varying degrees of association. Additionally, they can be applied in various domains such as decision making, control systems, and pattern recognition.

**Question 5: Can you provide an example application where Max-min composition of fuzzy relations would be useful?**

**Answer:** One example application is in multi-criteria decision making, where various factors contribute to a decision, and each factor has a degree of importance or influence. By representing these factors as fuzzy relations and applying Max-min composition, one can assess the overall impact of different combinations of factors on the decision outcome, considering the varying degrees of importance assigned to each factor.

**EXPERIMENT 8**

**Aim :** Implementation of simple neural network (McCulloh-Pitts model).

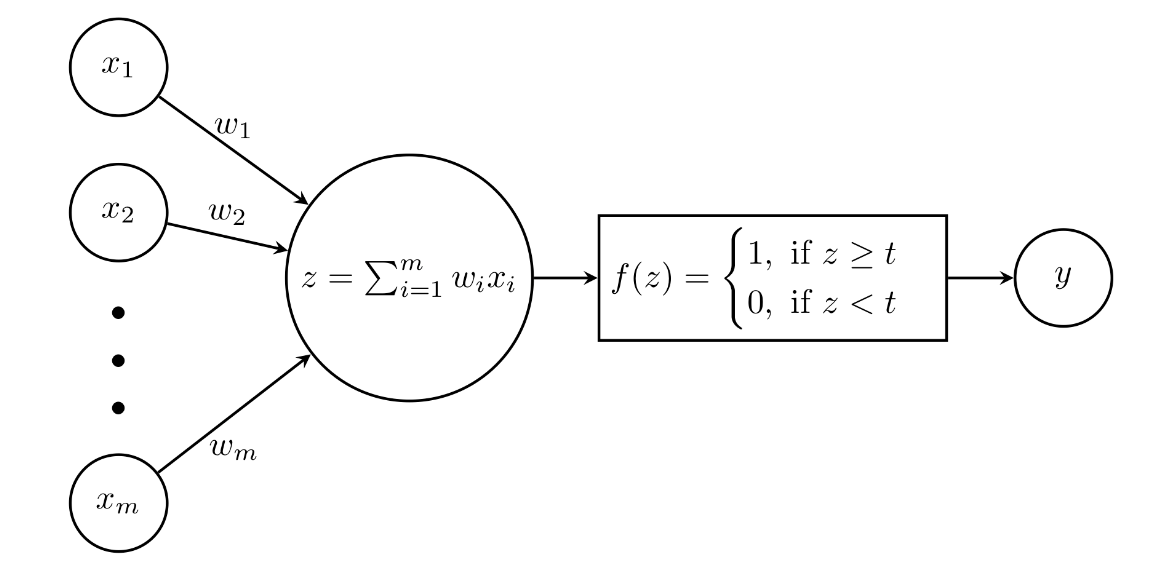
**Theory :**

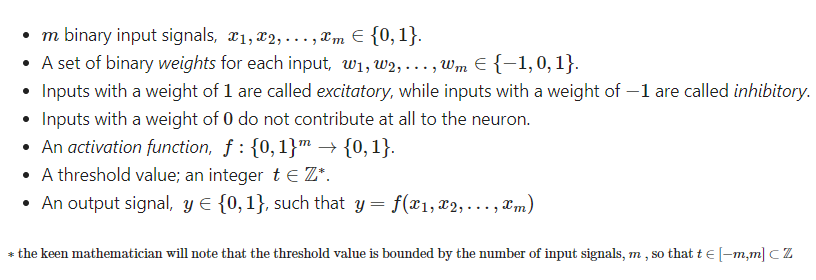
McCulloch-Pitts (MCP) Neuron. The MCP Neuron is a simplified mathematical model of a biological neuron which can be used to construct Boolean logic gates.

Although the MCP neuron is rudimentary by today's standards, it formed an early and important stepping stone in the history of artificial neural networks. Frank Rosenblatt's Perceptron and later artificial neural networks both build on the fundamental ideas of the MCP neuron.

**The MCP Model of an Artificial Neuron**

The McCulloch-Pitts (MCP) model is the earliest mathematical representation of an artificial neuron. It was first proposed in 1943 by the neurophysiologist Walter S. McCulloch and the logician Walter Pitts. The MCP model abstracts the biological notion of a neuron as a mathematical model containing:



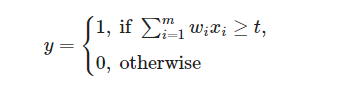


The logic is as follows:

* If the sum of the weighted inputs exceeds the threshold value, then the neuron is said to be activated and the output signal is 1
* Otherwise the neuron is not activated and the output signal is 0

Nowadays there are a variety of activation functions which are used to form a binary classifier. In its original formulation, the activation function took the form of a Heaviside step function. The Heaviside step function matches the logic above by outputting 1 when the neuron is activated; 0 otherwise.

Ie.



Using this model, McCulloch and Pitts showed (using some impressive logical calculus) that is was possible to construct the three basic Boolean logic gates: OR, AND and NOT

**The MCP Neuron in Python**

Each logic gate is completely determined by its weights and its threshold value. So the idea is to define a class which takes in the weight and threshold value as inputs, from which we can define each type of gate. We can then define a simple process which takes in input signals, applies an activation function (using the input weights) and then uses the threshold value to decide if the output of that signal is a 1 or a 0.

To keep things simple, the decide function below, which decides if a set of inputs returns a 1 or a 0, operates only on 1-D arrays (list) of signals; as opposed to a truth table which is a 2-D matrix. Thus we can define the concept of a message:

* a message is an m--dimensional vector of binary input signals, 

Then, ignoring the last column of the truth table (which is the output signal), each row of the truth table is a message. Finally we can abstract the truth table (ignoring the last column) as a set of n messages, where n is the number of rows of the truth table.

The procedure for generating truth tables is then to make a decision on each row and append the output signals to the table (which is a 2-D array).

**Python Script:**

import numpy as np

import pandas as pd

class MCPNeuron(object):

def \_\_init\_\_(self, w = [1,1], t = 1):

self.w = np.array(w)

self.t = t

def decide(self, message):

x = message # consistency with function notation above

sum\_ = np.inner(self.w,x)

if sum\_ >= self.t:

return 1

else:

return 0

def TruthTable(self, in\_signals, in\_labels, out\_label):

table = pd.DataFrame(in\_signals, columns = in\_labels)

out\_signals = []

for row in in\_signals:

signal = self.decide(message = row)

out\_signals.append(signal)

table[out\_label] = pd.Series(out\_signals)

return table

in\_signals = np.array([[0,0], [0,1], [1,0], [1,1]])

in\_labels = ['x1', 'x2']

out\_label = 'y'

# instantiate OR gate as an MCP Neuron class

OR = MCPNeuron(w = [1,1], t = 1)

OR\_table = OR.TruthTable(in\_signals, in\_labels = in\_labels, out\_label = out\_label)

print(OR\_table)

in\_signals = np.array([[0,0], [0,1], [1,0], [1,1]])

# instantiate AND gate as an MCP Neuron class

AND = MCPNeuron(w = [1,1], t = 2)

AND\_table = AND.TruthTable(in\_signals, in\_labels = in\_labels, out\_label = out\_label)

print(AND\_table)

NOT\_signals = np.array([[0], [1]])

# instantiate NOT gate as an MCP Neuron class

NOT = MCPNeuron(w = [-1], t = 0)

NOT\_table = NOT.TruthTable(NOT\_signals, in\_labels = ['x1'], out\_label = 'y')

print(NOT\_table)

in\_signals = np.array([[0,0], [0,1], [1,0], [1,1]])

# instantiate AND gate as an MCP Neuron class

NAND = MCPNeuron(w = [-1,-1], t = -1)

NAND\_table = NAND.TruthTable(in\_signals, in\_labels = in\_labels, out\_label = out\_label)

print(NAND\_table)

in\_signals = np.array([[0,0], [0,1], [1,0], [1,1]])

# instantiate AND gate as an MCP Neuron class

NOR = MCPNeuron(w = [-1,-1], t = 0)

NOR\_table = NOR.TruthTable(in\_signals, in\_labels = in\_labels, out\_label = out\_label)

print(NOR\_table)

**Output:**

**OR Gate:**

x1 x2 y

0 0 0 0X

1 0 1 1

2 1 0 1

3 1 1 1

**AND Gate:**

x1 x2 y

0 0 0 0

1 0 1 0

2 1 0 0

3 1 1 1

**NOT Gate**:

x1 y

0 0 1

1 1 0

**NAND Gate**:

x1 x2 y

0 0 0 1

1 0 1 1

2 1 0 1

3 1 1 0

**NOR Gate**:

x1 x2 y

0 0 0 1

1 0 1 0

2 1 0 0

3 1 1 0

\*\* Process exited - Return Code: 0 \*\*

Press Enter to exit terminal

**Conclusion:** Successfully implemented simple neural network (McCulloh-Pitts model).

**VIVA VOICE**

**Question 1: What is the McCulloh-Pitts model?**

**Answer 1:** The McCulloh-Pitts model is one of the earliest proposed neural network models, introduced by Warren McCulloh and Walter Pitts in 1943. It describes a simplified mathematical model of a biological neuron and serves as the foundation for modern artificial neural networks.

**Question 2: How does the McCulloh-Pitts neuron model work?**

**Answer 2:** In the McCulloh-Pitts model, each neuron receives binary inputs from other neurons or external sources. It computes a weighted sum of these inputs and applies a threshold function to determine its output. If the weighted sum exceeds a certain threshold, the neuron fires (outputs 1); otherwise, it remains inactive (outputs 0).

**Question 3: What are the key components of the McCulloh-Pitts neuron model?**

**Answer 3:** The key components include:

* Inputs: Binary signals representing the inputs to the neuron.
* Weights: Numeric values representing the strengths of connections between inputs and the neuron.
* Threshold: A threshold value that determines the neuron's activation level.
* Threshold Function: A step function that compares the weighted sum of inputs to the threshold and determines the neuron's output.

**Question 4: How is learning achieved in the McCulloh-Pitts model?**

**Answer 4:** The original McCulloh-Pitts model does not incorporate learning mechanisms. However, variations and extensions of the model, such as the perceptron learning rule developed by Frank Rosenblatt, introduced learning capabilities. The perceptron learning rule adjusts the weights of connections between neurons based on the error between the desired and actual outputs, allowing the network to learn from training data.

**Question 5: What are some limitations of the McCulloh-Pitts model?**

**Answer 5:** The McCulloh-Pitts model has several limitations, including:

* It only supports binary inputs and outputs, limiting its ability to represent and process continuous or real-valued data.
* It relies on a threshold function, which results in binary output states and discontinuous decision boundaries.
* It does not account for the nonlinearities present in biological neurons, such as synaptic delays and dendritic integration processes.
* It does not incorporate mechanisms for learning or adaptation, making it less suitable for tasks requiring complex learning processes.

**EXPERIMENT 9**

**Aim:** Study of ANFIS Architecture.

**Theory:**

An adaptive neuro-fuzzy inference system or adaptive network-based fuzzy inference system (ANFIS) is a kind of artificial neural network that is based on Takagi–Sugeno fuzzy inference system. Since it integrates both neural networks and fuzzy logic principles, it has potential to capture the benefits of both in a single framework. Its inference system corresponds to a set of fuzzy IF–THEN rules that have learning capability to approximate nonlinear functions. Hence, ANFIS is considered to be a universal estimator. For using the ANFIS in a more efficient and optimal way, one can use the best parameters obtained by genetic algorithm.

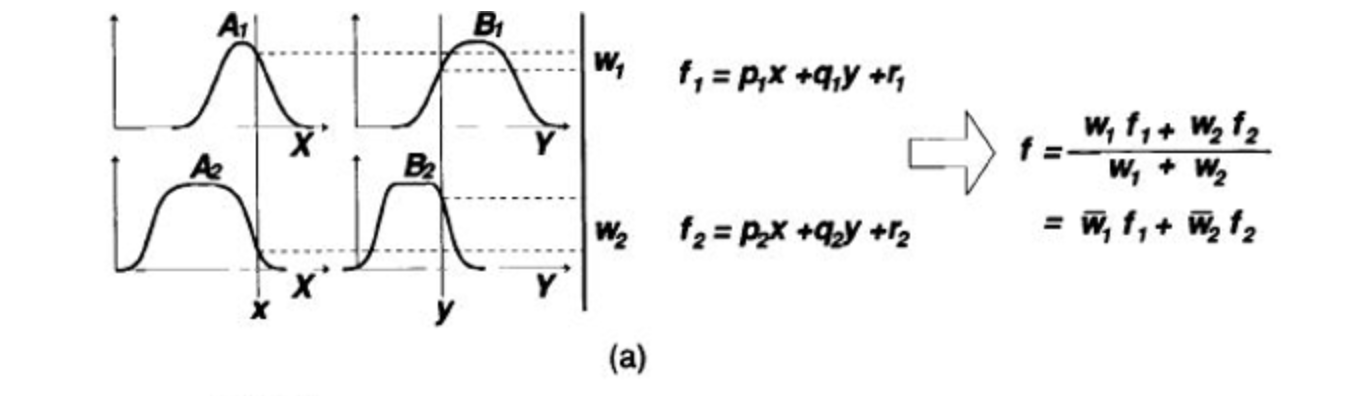
**ANFIS Architecture:**

**1. Representing Takagi-Sugeno Fuzzy Model**

For simplicity, we assume that the fuzzy inference sytem under consideration has two inputs x and y and one output z. For a first-order Takagi-Sugeno fuzzy model, a common rule set with two fuzzy if-then rules is the following:

Rule 1: If x is A1 and y is B1, then f1=p1x+q1y+r1;

Rule 2: If x is A2 and y is B2, then f2=p2x+q2y+r2;



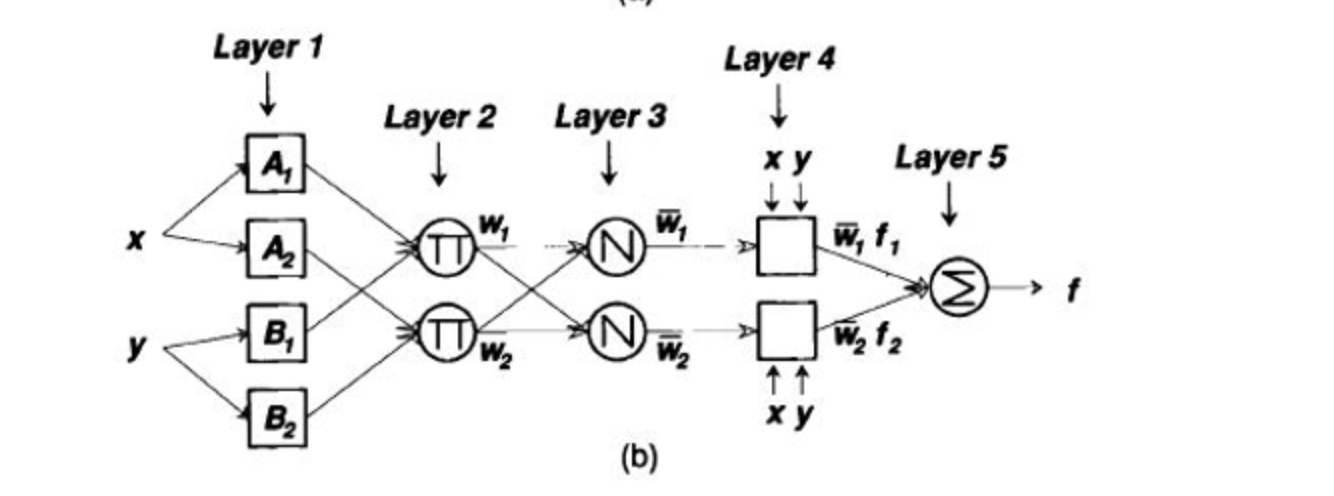
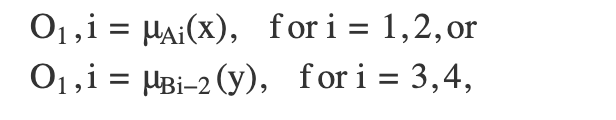


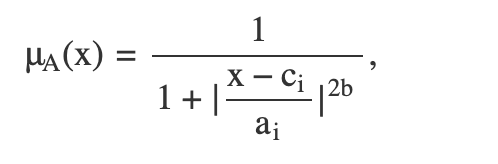
Figure 1: (a) A two inputs first order Takagi-Sugeno fuzzy model with two rules; (b) The equivalent ANFIS architecture.

Figure 1(a) illustrates the reasoning mechanism for this Takagi-Sugeno model; the corresponding equivalent ANFIS architecture is as shown in Figure 1(b), where nodes of the same layer have similar functions, as described next. (Here we denote the output of the ith node in layer l as Ol,i )

**Layer 1** Every node i in this layer is an adaptive node with a node function

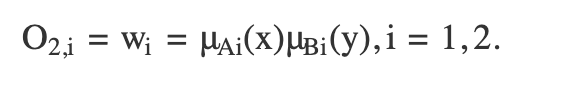


where x (or y) is the input to node i and Ai (or Bi-2) is a linguistic label (such as "small" or "large") associated with this node. In other words, O1,i is the membership grade of a fuzzy set A ( =A1 , A2 , B1 or B2 ) and it specifies the degree to which the given input x (or y) satisfies the quantifier A.



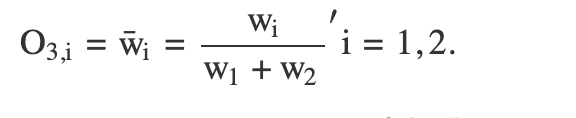
where {ai, bi, ci} is the parameter set. As the values of these parameters change, the bell-shaped function varies accordingly, thus exhibiting various forms of membership function for fuzzy set A. Parameters in this layer are referred to as premise parameters.

**Layer 2** Every node in this layer is a fixed node labeled anfis, whose output is the product of all the incoming signals:



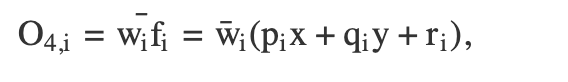
Each node output represents the firing strength of a rule. In general, any other T-norm operators that perform fuzzy AND can be used as the node function in this layer.

**Layer 3** Every node in this layer is a fixed node labeled N. The ith node calculates the ratio of the ith rule's firing strength to the sum of all rules' firing strengths:



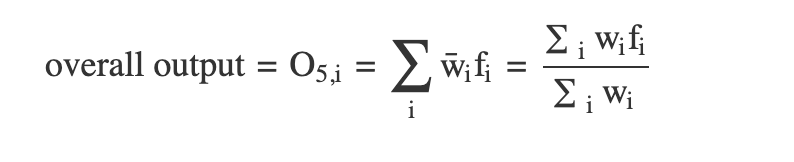
For convenience, outputs of this layer are called normalized firing strengths.

**Layer 4** Every node i in this layer is an adaptive node with a node function:



where anfis is a normalized firing strength from layer 3 and {pi, qi, ri} is the parameter set of this node. Parameters in this layer are referred to as consequent parameters.

**Layer 5** The single node in this layer is a fixed node labeled anfis, which computes the overall output as the summation of all incoming singals:



Thus we have constructed an adaptive network that is functionally equivalent to a Sugeno fuzzy model.

**2. Representing Tsukamoto Fuzzy ModelS:**

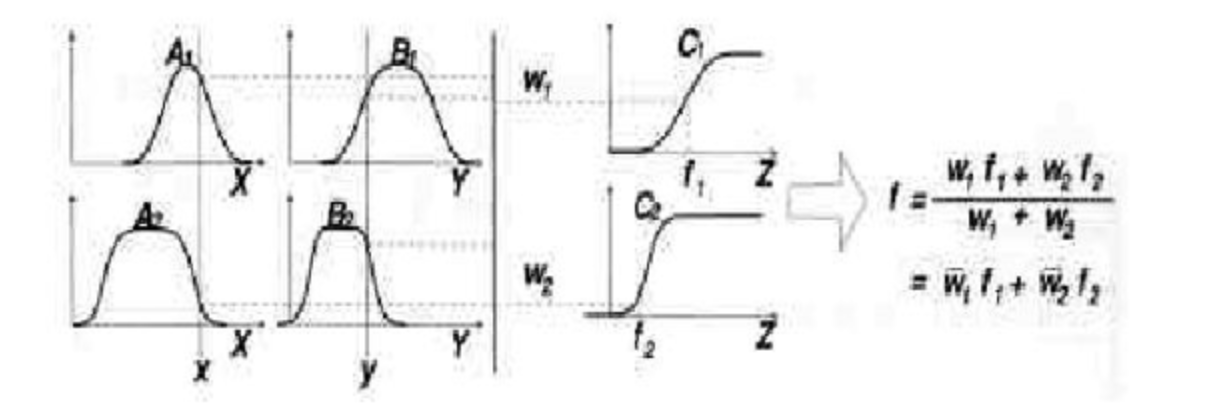


Figure 2(a)

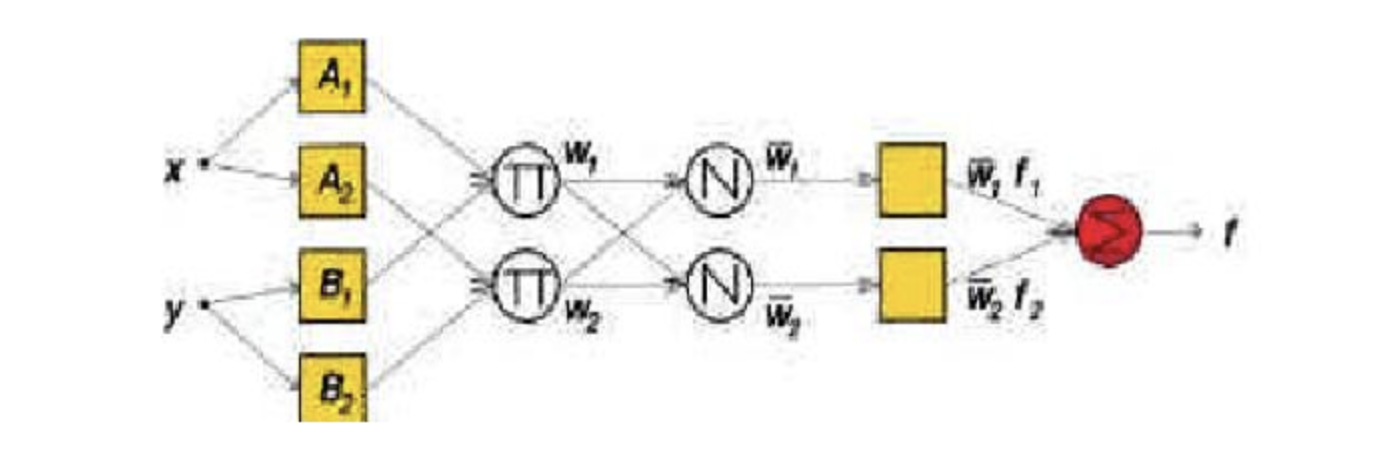


Figure 2: (b)

Figure 2: (a) A two-rule Tsukamoto fuzzy model; (b) The equivalent ANFIS architecture

The extension from TS ANFIS to Tsukamoto ANFIS is straightforward, as show in Figure 2, where the output of each rule (fi, i=1, 2) is induced jointly by a consequent membership function and a firing strength.

**3. Representing Mamdani Fuzzy Model**

For the Mamdani fuzzy inference system with max-min composition, a corresponding ANFIS can be constructed if discrete approximations are used to replace the integrals in the centroid defuzzification scheme introduced in here. However, the resulting ANFIS is much more complicated than either TS ANFIS or Tsukamoto ANFIS. The extra complexity in structure and computation of Mamdani ANFIS with max-min composition does not necessarily imply better learning capability or approximation power. If we adopt sum-product composition and centroid defuzzification for a Mamdani fuzzy model, a corresponding ANFIS can be constructed easily based on Theorem directly without using any approximation at all.

**Conclusion:** Successfully studied and represented ANFIS architecture.

**VIVA VOICE**

**Question 1: What does ANFIS stand for, and what is its purpose?**

**Answer 1:** ANFIS stands for Adaptive Neuro-Fuzzy Inference System. Its purpose is to combine the adaptability of neural networks with the interpretability of fuzzy logic systems to model complex systems, particularly for tasks involving uncertain or imprecise data.

**Question 2: Describe the architecture of an ANFIS.**

**Answer 2:** An ANFIS typically consists of five layers:

* Input Layer: Receives input variables.
* Fuzzification Layer: Converts crisp input variables into fuzzy sets using membership functions.
* Rule Layer: Computes the firing strengths of each rule by combining the fuzzy membership grades.
* Normalization Layer: Normalizes the firing strengths to ensure proper weighting.
* Output Layer: Combines the outputs of all rules to produce the final output.

**Question 3: How are the parameters of an ANFIS model trained?**

**Answer 3:** The parameters of an ANFIS model, including the membership functions and the rule parameters, are typically trained using a hybrid learning algorithm. This algorithm combines gradient descent methods, such as backpropagation, with least squares estimation to update the parameters and optimize the model's performance.

**Question 4: What are the advantages of using ANFIS over traditional neural networks or fuzzy logic systems?**

**Answer 4:** ANFIS offers several advantages, including:

* Interpretability: ANFIS models are easier to interpret and understand compared to black-box neural networks.
* Adaptability: ANFIS can adapt to changing environments and data patterns, similar to neural networks.
* Robustness: ANFIS combines the robustness of fuzzy logic systems with the learning capabilities of neural networks, making it suitable for complex and uncertain systems.
* Generalization: ANFIS can generalize well to unseen data, making it suitable for prediction and classification tasks.

**Question 5: Can you provide an example application where ANFIS could be used effectively?**

**Answer 5:** ANFIS can be used in various applications, such as time series prediction, financial forecasting, and control systems. For example, in financial forecasting, ANFIS can model the relationships between economic indicators and stock prices to predict future market trends. In control systems, ANFIS can adaptively adjust control parameters based on real-time sensor data to optimize system performance.