Vel Tech Multi Tech
Dr.Rangarajan Dr.Sakunthala Engineering College
An Autonomous Institution

Approved by AICTE, Affiliated to Anna University, Chennai.
Accredited by NBA (BME, CSE, ECE, EEE, IT & MECH)Accredited by NAAC.
#42, Avadi-Vel Tech Road, Avadi, Chennai- 600062, Tamil Nadu, India.

191CSV78 – SOFT COMPUTING
(INTEGRATED LABORATORY)

NAME :

REGISTER NO :

ROLL NO :

BRANCH : **B.E - Computer Science and Engineering**

YEAR **III**

SEMESTER **VI**

REGULATION **2019**

Department of Computer Science and Engineering

Vision

To emerge as centre for academic excellence in the field of Computer Science and Engineering by exposure to research and industry practices.

Mission

M1 - To provide good teaching and learning environment with conducive research atmosphere in the field of Computer Science and Engineering.
M2 - To propagate lifelong learning.
M3 - To impart the right proportion of knowledge, attitudes and ethics in students to enable them take up positions of responsibility in the society and make significant contributions.

Vel Tech Multi Tech

Dr.Rangarajan Dr.Sakunthala Engineering College

An Autonomous Institution

Approved by AICTE, Affiliated to Anna University, Chennai.
Accredited by NBA (BME, CSE, ECE, EEE, IT & MECH),
Accredited by NAAC
#42, Avadi-Vel Tech Road, Avadi, Chennai- 600062, Tamil Nadu, India.

CERTIFICATE

Name: ……………………………………………………………………………………………………

Year: ……………… Semester: ...., Branch: **B.E – Computer Science and Engineering**

University Register No: ____________ College Roll No: VM

Certified that this is the bonafide record of work done by the above student in the **191CSV78 – SOFT COMPUTING (INTEGRATED LABORATORY)** during the academic year 2023-24.

Signature of Head of the Department

Signature of Course Incharge

Submitted for the University Practical Examination held on ………………... at VEL TECH MULTI TECH Dr.RANGARAJAN Dr.SAKUNTHALA ENGINEERING COLLEGE, #42, AVADI – VEL TECH ROAD, AVADI, CHENNAI- 600062.

Signature of Examiners

Internal Examiner:……………………

External Examiner:………………

Date:………………..

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEOs	PROGRAMME EDUCATIONAL OBJECTIVES
PEO1	Ability to identify, formulate and analyze complex Computer Science and Engineering problems in the areas of hardware, software, theoretical Computer Science and applications to reach significant conclusions by applying Mathematics, Natural sciences, Computer Science and Engineering principles.
PEO2	Apply knowledge of mathematics, natural science, engineering fundamentals and system fundamentals, software development, networking & communication, and information security to the solution of complex engineering problems in computer science and engineering to get benefits in their professional career or higher education and research or technological entrepreneur.
PEO3	Design solutions for complex computer science and engineering problems using state of the art tools and techniques, components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO's	PROGRAMME SPECIFIC OUTCOMES
PSO1	An ability to apply, design and development of application oriented software systems and to test and document in accordance with Computer Science and Engineering.
PSO2	The design techniques, analysis and the building, testing, operation and maintenance of networks, databases, security and computer systems (both hardware and software).
PSO3	An ability to identify, formulate and solve hardware and software problems using sound computer engineering principles.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Engineering Graduates will be able to:

POs	PROGRAMME OUTCOMES
PO1	**Engineering Knowledge:** Apply knowledge of **mathematics, science**, **engineering fundamentals** and an **Engineering Specialization** to the solution of complex engineering problems.
PO2	**Problem Analysis: Identify**, **formulate, review research literature** and **analyze complex engineering** problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design / Development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations.
PO4	**Conduct Investigations of Complex Problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The Engineer and Society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions
PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OBJECTIVES

The student should be made to:

- Build using different compiler writing tools.
- Diagnose how to implement the different Phases of compiler
- Express the familiarizes how to use the control flow and data flow analysis
- Design the simple optimization techniques

COURSE OUTCOMES

At the end of the course, the student should be able to:

CO-PO & PSO Mapping

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	3	3	3	3	2	1	-	-	-	-	1	3	3	2
CO2	3	3	3	3	2	1	-	-	-	-	-	1	3	3	1
CO3	3	3	3	3	2	1	-	-	-	-	-	1	3	3	1
CO4	3	3	3	3	3	1	-	-	1	-	-	1	3	3	1
CO5	3	3	3	3	2	1	1	-	-	-		1	3	3	1
CO	3	3	3	3	2	1	1	-	1	-	-	1	**3**	**3**	**1**

Mapping CO's with PO's and PSO'S

Course Outcome	PO's												PSO's		
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12	PSO 1	PSO 2	PSO 3
CO1	3	3	3	3	3	2	1	-	-	-	-	1	3	3	2
CO2	3	3	3	3	2	1	-	-	-	-	-	1	3	3	1
CO3	3	3	3	3	2	1	-	-	-	-	-	1	3	3	1
CO4	3	3	3	3	3	1	-	-	1	-	-	1	3	3	1
CO5	3	3	3	3	2	1	1	-	-	-	-	1	3	3	1
CO	**3**	**3**	**3**	**3**	**2**	**1**	**1**	**-**	**1**	**-**	**-**	**1**	**3**	**3**	**1**

1 – Low **2 – Medium** **3 – High**

Course Code : 191CSV78

Course Name : SOFT COMPUTING (INTEGRATED LABORATORY)

COURSE PLAN

EX. NO	DATE	LIST OF EXERCISES	CO	PAGE NO	SIGN
1		IMPLEMENTATION OF FUZZY CONTROLLER			
2		PROGRAMMING EXERCISE ON CLASSIFICATION WITH A DISCRETE PERCEPTRON			
3		IMPLEMENTATION OF XOR WITH BACK PROPAGATION ALGORITHM			
4		IMPLEMENTATION OF SELF ORGANIZING MAPS FOR A SPECIFIC APPLICATION			
5		PROGRAMMING EXERCISE ON MAXIMIZNG A FUNCTION USING GENETIC ALGORITHM			
6		IMPLEMENTATION OF TWO INPUT SINE FUNCTION			
7		IMPLEMENTATION OF THREE INPUT NON LINEAR FUNCTION			

Ex No 1: **IMPLEMENTATION OF FUZZY CONTROLLER**

DATE:

AIM:

To Implement a fuzzy controller involves creating a system that makes decisions based on fuzzy logic rules and membership functions.

ALGORITHM:

- Define input and output variables to control and make decision
- For each input and output variable, create membership functions that define their linguistic range.
- Define rules that connect combinations of inputs' membership functions to outputs' membership functions
- Convert crisp inputs (real-world values) into fuzzy sets based on the defined membership functions.
- Use the rules to infer the appropriate output membership functions based on the fuzzified inputs
- Convert the fuzzy output back to a crisp value for the actual control action.

PROGRAM:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Antecedent variables (inputs)
distance = ctrl.Antecedent(np.arange(0, 101, 1), 'distance')
speed = ctrl.Antecedent(np.arange(0, 101, 1), 'speed')

# Consequent variable (output)
acceleration = ctrl.Consequent(np.arange(0, 101, 1), 'acceleration')

# Membership functions for distance
distance['near'] = fuzz.trimf(distance.universe, [0, 0, 50])
distance['medium'] = fuzz.trimf(distance.universe, [0, 50, 100])
distance['far'] = fuzz.trimf(distance.universe, [50, 100, 100])

# Membership functions for speed
speed['slow'] = fuzz.trimf(speed.universe, [0, 0, 50])
speed['medium'] = fuzz.trimf(speed.universe, [0, 50, 100])
speed['fast'] = fuzz.trimf(speed.universe, [50, 100, 100])

# Membership functions for acceleration
acceleration['decelerate'] = fuzz.trimf(acceleration.universe, [0, 0, 50])
acceleration['maintain'] = fuzz.trimf(acceleration.universe, [0, 50, 100])
acceleration['accelerate'] = fuzz.trimf(acceleration.universe, [50, 100, 100])
```

```
# Rules for the fuzzy logic
rule1 = ctrl.Rule(distance['near'] | speed['slow'], acceleration['decelerate'])
rule2 = ctrl.Rule(distance['medium'] | speed['medium'], acceleration['maintain'])
rule3 = ctrl.Rule(distance['far'] | speed['fast'], acceleration['accelerate'])

# Control system
acceleration_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
acceleration_simulation = ctrl.ControlSystemSimulation(acceleration_ctrl)

# Pass inputs to the controller and compute the output
acceleration_simulation.input['distance'] = 70  # Distance to the obstacle
acceleration_simulation.input['speed'] = 30     # Current speed
acceleration_simulation.compute()

# Display the computed acceleration
print("Computed Acceleration:", acceleration_simulation.output['acceleration'])

# Visualize the membership functions (optional)
distance.view()
speed.view()
acceleration.view()
```

OUTPUT:

```
Computed Acceleration: 50.0
```

RESULT:

Thus the given output was verified for the experiment.

EX NO:2 PROGRAMMING EXERCISE ON CLASSIFICATION WITH A DISCRETE PERCEPTRON

DATE:

AIM:

To Develop a Python program to implement a discrete perceptron for binary classification.

ALGORITHM:

1.Initialize weights (w) and bias (b) randomly or to zero.
2.Iterate through the training dataset for a fixed number of epochs.
3.Input the features (x) of the data point to the perceptron.

- Calculate the weighted sum of inputs: \text{weighted_sum} = \sum_{i=1}^{n} (w_i \times x_i) + b, where n is the number of features.
- Apply Step Function (Discrete Activation): \text{output} = \begin{cases} 1 & \text{if } \text{weighted_sum} \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}
- Update Weights and Bias:

4. If the output doesn't match the expected label:

- Adjust weights: w_i = w_i + \text{learning_rate} \times (expected - output) \times x_i for all i features.
- Adjust bias: b = b + \text{learning_rate} \times (expected - output)

5. Repeat Until Convergence or Maximum Epochs Reached

PROGRAM:

```
class DiscretePerceptron:
    def __init__(self, input_size):
        self.weights = [0] * input_size
        self.bias = 0

    def predict(self, inputs):
        activation = self.bias
        for i in range(len(inputs)):
            activation += self.weights[i] * inputs[i]
        return 1 if activation >= 0 else 0

    def train(self, training_inputs, labels, epochs=10, learning_rate=1):
        for epoch in range(epochs):
            for inputs, label in zip(training_inputs, labels):
```

```
                prediction = self.predict(inputs)
                for i in range(len(self.weights)):
                    self.weights[i] += learning_rate * (label - prediction) * inputs[i]
                self.bias += learning_rate * (label - prediction)
            print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {self.evaluate(training_inputs,
labels)}")

    def evaluate(self, inputs, labels):
        correct = 0
        for i in range(len(inputs)):
            prediction = self.predict(inputs[i])
            if prediction == labels[i]:
                correct += 1
        return correct / len(inputs)


# Training data for AND gate
training_inputs = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]
labels = [0, 0, 0, 1]

# Creating a Discrete Perceptron and training it on AND gate data
perceptron = DiscretePerceptron(input_size=2)
perceptron.train(training_inputs, labels, epochs=10, learning_rate=0.1)

# Testing the trained model
test_inputs = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]
print("\nTesting the model:")
for i, test_input in enumerate(test_inputs):
    prediction = perceptron.predict(test_input)
    print(f"Input: {test_input} Predicted Output: {prediction}")
```

OUTPUT:

```
Epoch 1/10 - Accuracy: 0.5
Epoch 2/10 - Accuracy: 0.5
Epoch 3/10 - Accuracy: 0.5
Epoch 4/10 - Accuracy: 0.75
Epoch 5/10 - Accuracy: 1.0
Epoch 6/10 - Accuracy: 1.0
Epoch 7/10 - Accuracy: 1.0
Epoch 8/10 - Accuracy: 1.0
Epoch 9/10 - Accuracy: 1.0
Epoch 10/10 - Accuracy: 1.0

Testing the model:
Input: [0, 0] Predicted Output: 0
Input: [0, 1] Predicted Output: 0
Input: [1, 0] Predicted Output: 0
Input: [1, 1] Predicted Output: 1
```

RESULT:

The given output is verified for the classification of discrete perceptron

EX NO:3. IMPLEMENTATION OF XOR WITH BACK PROPAGATION ALGORITHM

DATE:

AIM: The goal is to create a neural network capable of learning and predicting the XOR function's outputs based on given inputs.

ALGORITHM:

1.Randomly initialize weights and biases for connections between layers
2. Define the XOR truth table dataset containing input-output pair
3. Input the XOR data values to the neural network.

- Compute the outputs for each input through forward propagation:
- Calculate the weighted sum of inputs and apply activation function for hidden layer(s) and output layer.

4.Update weights and biases using the backpropagation algorithm
5.Adjust weights and biases using backpropagation to minimize errors.

PROGRAM:

```
import numpy as np

class XORNeuralNetwork:
    def __init__(self):
        # Initialize weights and biases for the network
        self.input_size = 2
        self.hidden_size = 4
        self.output_size = 1

        self.hidden_weights = np.random.randn(self.input_size, self.hidden_size)
        self.hidden_bias = np.zeros((1, self.hidden_size))

        self.output_weights = np.random.randn(self.hidden_size, self.output_size)
        self.output_bias = np.zeros((1, self.output_size))

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward_propagation(self, inputs):
        # Forward pass through the network
        self.hidden_layer_activation = np.dot(inputs, self.hidden_weights) + self.hidden_bias
```

```
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_activation)

        self.output_layer_activation = np.dot(self.hidden_layer_output, self.output_weights) +
self.output_bias
        self.predicted_output = self.sigmoid(self.output_layer_activation)

        return self.predicted_output

    def backward_propagation(self, inputs, targets, learning_rate):
        # Backpropagation to update weights and biases
        error = targets - self.predicted_output
        output_delta = error * self.sigmoid_derivative(self.predicted_output)

        hidden_layer_error = output_delta.dot(self.output_weights.T)
        hidden_layer_delta = hidden_layer_error *
self.sigmoid_derivative(self.hidden_layer_output)

        self.output_weights += self.hidden_layer_output.T.dot(output_delta) * learning_rate
        self.output_bias += np.sum(output_delta, axis=0, keepdims=True) * learning_rate

        self.hidden_weights += inputs.T.dot(hidden_layer_delta) * learning_rate
        self.hidden_bias += np.sum(hidden_layer_delta, axis=0, keepdims=True) * learning_rate

    def train(self, training_inputs, training_outputs, epochs, learning_rate):
        for epoch in range(epochs):
            self.forward_propagation(training_inputs)
            self.backward_propagation(training_inputs, training_outputs, learning_rate)

    def predict(self, inputs):
        return self.forward_propagation(inputs)


# Training data for XOR gate
XOR_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
XOR_outputs = np.array([[0], [1], [1], [0]])

# Creating XORNeuralNetwork instance and training the network
xor_nn = XORNeuralNetwork()
xor_nn.train(XOR_inputs, XOR_outputs, epochs=10000, learning_rate=0.1)

# Testing the trained model
print("Predictions after training:")
for i in range(len(XOR_inputs)):
    prediction = xor_nn.predict(XOR_inputs[i])
    print(f"Input: {XOR_inputs[i]} Predicted Output: {prediction}")
```

OUTPUT:

```
Predictions after training:
Input: [0 0] Predicted Output: [[0.01653688]]
Input: [0 1] Predicted Output: [[0.98427626]]
Input: [1 0] Predicted Output: [[0.98412436]]
Input: [1 1] Predicted Output: [[0.01932397]]
```

RESULT:
The given output is verified for the XOR GATE with Back Propagation Algorithm.

EX NO:4. IMPLEMENTATION OF SELF ORGANIZING MAPS FOR A SPECIFIC APPLICATION

DATE:

AIM:

The objective is to create a SOM-based model that effectively clusters and represents complex data in a lower-dimensional space, providing insights and visualization of the data's underlying structure.

ALGORITHM:

1. Initialize weights for each node in the grid with random values or small random samples from the dataset.
2. Define the learning rate (α) and neighborhood radius (r).
3. For each input vector, find the node in the SOM grid whose weights are closest (most similar) to the input vector.
4. Compute the Euclidean distance or another similarity measure to identify the Best-Matching Unit
5. Adjust the weights of the BMU and its neighboring nodes based on the input vector and learning rate
6. Decrease the learning rate (α) and neighborhood radius (r) over time to gradually refine the map.

7. Iterate through the dataset for a defined number of epochs, updating the SOM weights based on input vectors

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from minisom import MiniSom
from PIL import Image

# Load an image and convert it to a NumPy array
image = Image.open('path_to_your_image.jpg')  # Replace with your image path
image = image.resize((100, 100))  # Resize for faster processing
data = np.array(image)
data = data.reshape(-1, 3)  # Reshape to a 1D array of RGB values

# Define SOM parameters
```

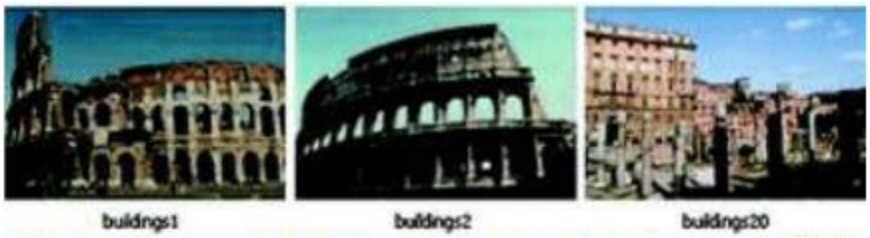
```
width = 10
height = 10
input_len = data.shape[1]
sigma = 1.0
learning_rate = 0.5
iterations = 10000

# Initialize SOM
som = MiniSom(width, height, input_len, sigma=sigma, learning_rate=learning_rate)
som.random_weights_init(data)
print("Training SOM...")
som.train_random(data, iterations)

# Get the SOM's weights and map input data to their closest neurons
mapped = som.win_map(data)

# Create a new image based on the SOM's clusters
mapped_image = np.zeros((width * height, 3))
for i, x in enumerate(mapped):
    mapped_image[i] = np.mean(x, axis=0)

mapped_image = mapped_image.reshape(width, height, 3).astype(np.uint8)

# Display the original and mapped images
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image)
ax[0].set_title('Original Image')
ax[0].axis('off')

ax[1].imshow(mapped_image)
ax[1].set_title('SOM Mapped Image')
ax[1].axis('off')

plt.show()
```

OUTPUT:

FIG 1:

QUERY IMAGE

RESULTANT IMAGE:



RESULT:

The Output for Given Image is Classified for the given Experiment.

EX NO:5. PROGRAMMING EXERCISE ON MAXIMIZNG A FUNCTION USING GENETIC ALGORITHM

DATE:

AIM:

The objective is to create an evolutionary optimization technique capable of finding the global maximum of a predefined function by evolving a population of potential solutions.

ALGORITHMS:

1. The objective is to create an evolutionary optimization technique capable of finding the global maximum of a predefined function by evolving a population of potential solutions.
2. Define a fitness function that evaluates the fitness (objective value) of each individual based on the given function to be maximized.
3. valuate the fitness of each individual in the population using the defined fitness function
4. Select individuals from the population for reproduction (mating pool) based on their fitness
5. Perform crossover or recombination between selected individuals to create offspring.
6. Apply mutation to some of the offspring individuals with a low probability to introduce diversity
7. Update the population

PROGRAM:

```
import random

# Define the function to be maximized
def fitness_function(x):
    return x**2 + 6*x + 5

# Genetic Algorithm parameters
population_size = 100
mutation_rate = 0.1
num_generations = 100

# Define the range for x values
min_x = -10
max_x = 10

# Function to create an initial population
def create_initial_population(population_size):
    return [random.uniform(min_x, max_x) for _ in range(population_size)]

# Function to calculate fitness scores for the population
```

```
def calculate_fitness(population):
    return [fitness_function(x) for x in population]

# Function for tournament selection
def tournament_selection(population, fitness_scores):
    selected = []
    for _ in range(len(population)):
        idx1, idx2 = random.sample(range(len(population)), 2)
        if fitness_scores[idx1] > fitness_scores[idx2]:
            selected.append(population[idx1])
        else:
            selected.append(population[idx2])
    return selected

# Function for single-point crossover
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

# Function for mutation
def mutate(individual):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.uniform(min_x, max_x)
    return individual

# Main genetic algorithm
population = create_initial_population(population_size)

for generation in range(num_generations):
    fitness_scores = calculate_fitness(population)

    # Select parents
    selected_parents = tournament_selection(population, fitness_scores)

    # Perform crossover
    new_population = []
    for i in range(0, len(selected_parents), 2):
        child1, child2 = crossover(selected_parents[i], selected_parents[i + 1])
        new_population.extend([child1, child2])

    # Mutate
    population = [mutate(individual) for individual in new_population]
```

```
# Find the best individual in the final population
best_fitness_scores = calculate_fitness(population)
best_individual_idx = best_fitness_scores.index(max(best_fitness_scores))
best_individual = population[best_individual_idx]
best_x = best_individual

print(f"The value of x that maximizes the function is: {best_x}")
print(f"The maximum value of the function is: {fitness_function(best_x)}")
```

OUTPUT:

```
The value of x that maximizes the function is: -3.0
The maximum value of the function is: 2.0
```

RESULT:

Thus the given Output for the given experiment is verified.

EX NO:6. IMPLEMENTATION OF TWO INPUT SINE FUNCTION

DATE:

AIM:

The objective is to create a neural network model that can learn and predict the sine function based on two input variables.

ALGORITHM:

- Initialize Neural Network Weights and Biases
- Split Dataset into Training and Validation Sets
- Shuffle and iterate over the training dataset in batches.
- Calculate loss/error between predicted and actual outputs.
- Back propagate the error to update weights using optimization algorithms like gradient descent or Adam.
- Validate the model's performance on the validation set to monitor for overfitting.
- Stop Training Based on Convergence Criteria

PROGRAM:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model

# Generate random data for training
np.random.seed(42)
num_samples = 1000
input_data = np.random.uniform(low=0, high=2*np.pi, size=(num_samples, 2))
output_data = np.sin(input_data[:, 0]) * np.sin(input_data[:, 1])

# Define the neural network architecture
inputs = tf.keras.Input(shape=(2,))
hidden = layers.Dense(32, activation='relu')(inputs)
output = layers.Dense(1)(hidden)

# Create the model
model = Model(inputs=inputs, outputs=output)
```

```
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(input_data, output_data, epochs=50, batch_size=32, validation_split=0.2)

# Test the model
test_input = np.array([[0.5, 1.5], [1.0, 2.0]])  # Example test input
predictions = model.predict(test_input)

print("Predictions for test input:")
for i in range(len(test_input)):
    print(f"Input: {test_input[i]}, Predicted Output: {predictions[i][0]}")
```

OUT PUT:

```
Predictions for test input:
Input: [0.5 1.5], Predicted Output: 0.3694686598777771
Input: [1. 2.], Predicted Output: 0.375812023878097 53
```

RESULT:

Thus the given Output is Verified for this experiment

EX NO:7: IMPLEMENTATION OF THREE INPUT NON LINEAR FUNCTION

DATE:

AIM:

The objective is to create a neural network model that can learn and predict the sine function based on two input variables.

ALGORITHM:

1. Initialize weights and biases in the neural network (random initialization or predefined values).
2. Split the generated dataset into training and validation sets for model evaluation.
3. Shuffle and iterate over the training dataset in batches.
4. Forward propagate input through the network to get predictions.
5. Calculate loss/error between predicted and actual outputs.
6. Backpropagate the error to update weights using optimization algorithms like gradient descent or Adam.
7. Validate the model's performance on the validation set to monitor for overfitting.
8. Terminate training based on convergence criteria

PROGRAM:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model

# Generate random data for training
np.random.seed(42)
num_samples = 1000
input_data = np.random.uniform(low=-2*np.pi, high=2*np.pi, size=(num_samples, 3))
output_data = np.sin(input_data[:, 0]) * np.cos(input_data[:, 1]) + 1 / (1 + np.exp(-input_data[:,
2]))

# Define the neural network architecture
inputs = tf.keras.Input(shape=(3,))
hidden1 = layers.Dense(64, activation='relu')(inputs)
hidden2 = layers.Dense(32, activation='relu')(hidden1)
output = layers.Dense(1)(hidden2)

# Create the model
model = Model(inputs=inputs, outputs=output)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
```

```
model.fit(input_data, output_data, epochs=50, batch_size=32, validation_split=0.2)

# Test the model
test_input = np.array([[0.5, 1.0, -1.5], [1.0, 2.0, 0.5]])  # Example test input
predictions = model.predict(test_input)

print("Predictions for test input:")
for i in range(len(test_input)):
    print(f"Input: {test_input[i]}, Predicted Output: {predictions[i][0]}")
```

OUTPUT:

```
Predictions for test input:
Input: [ 0.5  1.  -1.5], Predicted Output: -0.15037643969058
Input: [ 1.   2.   0.5], Predicted Output: 0.378538250684738
```

RESULT:

The Given Result is verified the particular Experiment.