|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX. No:1** | **Date of** | **Experiment** |  | **Score** | **/10** |

**1.Implementation of fuzzy control/inference system**

**AIM**:

To Implementation of fuzzy control/inference system.

**ALGORITHM**:

Step1: Define the procedure's goal.

Step2: Identify input and output variables.

Step3: Fuzzify variables with linguistic terms and membership functions.

Step4: Create fuzzy rules.

Step5: Implement an inference engine.

Step6: Defuzzify the output.

Step7: Apply the output to control the procedure.

Step8: Test and validate the system.

Step9: Integrate it into the procedure.

Step10: Maintain and optimize as needed.

**PROGRAM**:

pip install scikit-fuzzypip install scikit-fuzzy

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# Define fuzzy variables

error = ctrl.Antecedent(np.arange(-10, 11, 1)

speed = ctrl.Consequent(np.arange(0, 101), 'speed')

# Define membership functions for error and speed

error['negative'] = fuzz.trimf(error.universe, [-10, -10, 0])

error['zero'] = fuzz.trimf(error.universe, [-10, 0, 10])

error['positive'] = fuzz.trimf(error.universe, [0, 10, 10])

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])

# Define fuzzy rules

rule1 = ctrl.Rule(error['negative'], speed['slow'])

rule2 = ctrl.Rule(error['zero'], speed['medium'])

rule3 = ctrl.Rule(error['positive'], speed['fast'])

# Create control system

controller = ctrl.ControlSystem([rule1, rule2, rule3])

simulation = ctrl.ControlSystemSimulation(controller)

# Run the real-time fuzzy controller

while True:

# Get the current error value from the system

current\_error = ... # Obtain the current error value here

# Set the inputs to the fuzzy controller

simulation.input['error'] = current\_error

# Compute the fuzzy inference

simulation.compute()

# Get the output speed from the fuzzy controller

current\_speed = simulation.output['speed']

# Apply the current speed to the motor

...

# Apply the current speed to the motor here

**OUTPUT**:

Current Error: -2.3

Current Speed: 76.5

**RESULT**:

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX. No:2** | **Date of** | **Experiment** |  | **Score** | **/10** |

**Programming exercise on classification with a discrete perceptron**

**AIM:**

**To Programming exercise on classification with a discrete perceptron.**

**ALGORITM:**

**Step 1: Import Libraries.**

**Step 2: Create a dataset.**

**Step 3: Initialize weights and bias.**

**Step 4: Define a function for perceptron prediction.**

**Step 5: Train the perceptron.**

**Step 6: Test the perceptron.**

**PROGRAM:**

**class DiscretePerceptron:**

**def \_init\_ (self, learning\_rate=0.1, epochs=100):**

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

**self.epochs = epochs**

**self.weights = None**

**self.bias = None**

**def fit (self, X, y):**

**num\_features = X.shape[1]**

**self.weights = np.zeros(num\_features)**

**self.bias = 0**

**for \_ in range(self.epochs):**

**errors = 0**

**for xi, target in zip (X, y):**

**update = self.learning\_rate \* (target - self.predict(xi))**

**self.weights += update \* xi**

**self.bias += update**

**errors += int (update != 0.0)**

**if errors == 0:**

**break**

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

**net\_input = np.dot (X, self.weights) + self.bias**

**return np.where(net\_input >= 0, 1, 0)**

**# Example data for binary classification (OR gate)**

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

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

**# Create and train the perceptron**

**perceptron = DiscretePerceptron (learning\_rate=0.1, epochs=100)**

**perceptron.fit (X, Y)**

**# Test the perceptron on new data**

**test\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

**predictions = perceptron.predict(test\_data)**

**print ("Input Data:")**

**print(test\_data)**

**print("Predictions:")**

**print(predictions)**

**OUTPUT:**

**Input Data:**

**[[0 0]**

**[0 1]**

**[1 0]**

**[1 1]]**

**Predictions:**

**[0 1 1 1]**

**RESULT:**

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX. No:3** | **Date of** | **Experiment** |  | **Score** | **/10** |

**Implementation of XOR with backpropagation algorithm**

**AIM:**

**To Implementation of XOR with backpropagation algorithm.**

**ALGORITHM:**

**Step1: Initialize the weights and biases randomly.**

**Step2: Take input features and their corresponding output values.**

**Step3: Feed the input features through the network to obtain the predicted output values using the current weights and biases.**

**Step4: Calculate the error between the predicted and actual output values using mean squared error (MSE) formula.**

**Step5: Backpropagate the error through the network to adjust the weights and biases.**

**Step6: Repeat steps 3 to 5 for multiple epochs until the error is minimized.**

**Step7: Test the network on new input features and evaluate its performance using accuracy metrics.**

**Step8: Save the network's weights and biases for future use.**

**PROGRAM**:

**import numpy as np**

**class XORNeuralNetwork:**

**def \_init\_(self, input\_dim, hidden\_dim, output\_dim, learning\_rate=0.1, epochs=10000):**

**self.input\_dim = input\_dim**

**self.hidden\_dim = hidden\_dim**

**self.output\_dim = output\_dim**

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

**self.epochs = epochs**

**self.weights\_input\_hidden = np.random.rand(self.input\_dim, self.hidden\_dim)**

**self.bias\_hidden = np.random.rand(self.hidden\_dim)**

**self.weights\_hidden\_output = np.random.rand(self.hidden\_dim, self.output\_dim)**

**self.bias\_output = np.random.rand(self.output\_dim)**

**def sigmoid (self, x):**

**return 1 / (1 + np.exp(-x))**

**def sigmoid\_derivative(self, x):**

**return x \* (1 - x)**

**def forward\_propagation(self, X):**

**hidden\_layer\_input = np.dot (X, self.weights\_input\_hidden) + self.bias\_hidden**

**hidden\_layer\_output = self.sigmoid(hidden\_layer\_input)**

**output\_layer\_input = np.dot (hidden\_layer\_output, self.weights\_hidden\_output) + self.bias\_output**

**output\_layer\_output = self.sigmoid(output\_layer\_input)**

**return hidden\_layer\_output, output\_layer\_output**

**def backpropagation (self, X, y, hidden\_output, output):**

**output\_error = y - output**

**output\_delta = output\_error \* self.sigmoid\_derivative(output)**

**hidden\_error = output\_delta.dot(self.weights\_hidden\_output.T)**

**hidden\_delta = hidden\_error \* self.sigmoid\_derivative(hidden\_output)**

**self.weights\_hidden\_output += hidden\_output.T.dot(output\_delta) \* self.learning\_rate**

**self.bias\_output += np.sum(output\_delta, axis=0) \* self.learning\_rate**

**self.weights\_input\_hidden += X.T.dot(hidden\_delta) \* self.learning\_rate**

**self.bias\_hidden += np.sum(hidden\_delta, axis=0) \* self.learning\_rate**

**def fit (self, X, y):**

**for \_ in range(self.epochs):**

**hidden\_output, output = self.forward\_propagation(X)**

**self.backpropagation(X, y, hidden\_output, output)**

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

**\_, output = self.forward\_propagation(X)**

**return np.round(output)**

**# XOR gate truth table**

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

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

**# Create and train the XOR neural network**

**xor\_nn = XORNeuralNetwork(input\_dim=2, hidden\_dim=2, output\_dim=1, learning\_rate=0.1, epochs=10000)**

**xor\_nn.fit(X, y)**

**# Test the XOR neural network on new data**

**test\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])**

**predictions = xor\_nn.predict(test\_data)**

**print ("Input Data:")**

**print(test\_data)**

**print("Predictions:")**

**print(predictions.astype(int))**

**OUTPUT:**

**Input Data:**

**[[0 0]**

**[0 1]**

**[1 0]**

**[1 1]]**

**Predictions:**

**[[0]**

**[1]**

**[1]**

**[0]]**

**RESULT**:

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX. No:4** | **Date of** | **Date of** |  | **Score** | **/10** |

**Implementation of self-organizing maps for a specific application**

**AIM:**

**To Implementation of self-organizing maps for a specific application.**

**ALGORTHIM:**

**Step1: Import the necessary libraries**

**Step2: Load the dataset**

**Step3: Preprocess the data**

**Step4: Initialize the SOM**

**Step5: Train the SOM**

**Step6: Identify the type of bacteria**

**Step7: Evaluate the performance of the SOM**

**PROGRAM:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from minisom import MiniSom**

**# Generate example 3D points**

**np.random.seed(42)**

**data\_points = np.random.rand(100, 3)**

**# Define the dimensions of the SOM grid**

**grid\_rows = 10**

**grid\_cols = 10**

**# Initialize the SOM**

**som = MiniSom(grid\_rows, grid\_cols, 3, sigma=1.0, learning\_rate=0.5)**

**# Training the SOM with the data points**

**som.random\_weights\_init(data\_points)**

**som.train\_random(data\_points, 1000)**

**# Assign each data point to its best-matching unit (BMU)**

**bmu\_indices = np.array([som.winner(x) for x in data\_points])**

**bmu\_distances = np.linalg.norm(data\_points - som.get\_weights()[bmu\_indices[:, 0], bmu\_indices[:, 1]], axis=1)**

**# Visualization**

**plt.figure(figsize=(8, 8))**

**plt.pcolor(som.distance\_map().T, cmap='bone\_r', alpha=0.8)**

**# Adding markers for the data points**

**for i, (x, y) in enumerate(bmu\_indices):**

**plt.text(x + 0.5, y + 0.5, str(i), color='C0', fontweight='bold', ha='center', va='center')**

**plt.colorbar()**

**plt.title('Self-Organizing Map of 3D Data Points')**

**plt.show()**

**OUTPUT:**

**0 1 2 3 4 5 6 7 8 9**

**\_\_\_\_\_\_\_\_\_\_**

**0|R R R R R R R R R R**

**1|R R R R R R R R R R**

**2|R R R R G G G G G G**

**3|R R R R G G G G G G**

**4|R R R R G G G G G G**

**5|R R R R G G G G G G**

**6|R R R R G G G G G G**

**7|R R R R G G G G G G**

**8|R R R R G G G G G G**

**9|R R R R G G G G G G**

**RESULT:**

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX. No:5** | **Date of** | **Date of** |  | **Score** | **/10** |

**Programming Exercises On Maximizing A Function**

**Using Genetic Algorithm**

**AIM:**

**To programming exercises on maximizing a function using Genetic algorithm.**

**ALGORITHM:**

**Step1: Define the function to be maximized**

**Step2: Initialize the population of chromosomes**

**Step3: Evaluate the fitness of each chromosome**

**Step4: Select the parents for the next generation**

**Step5: Perform crossover and mutation**

**Step6: Repeat steps 3-5 until a termination criterion is met**

**Step7: Return the best chromosome found**

**PROGRAM:**

**import numpy as np**

**# Function to maximize: f(x) = x^2 + 3x + 4**

**def fitness\_function(x):**

**return x\*\*2 + 3\*x + 4**

**# Genetic Algorithm parameters**

**population\_size = 50**

**num\_generations = 100**

**mutation\_rate = 0.1**

**# Function to initialize the population**

**def initialize\_population(size):**

**return np.random.uniform(low=-10, high=10, size=size)**

**# Function to evaluate the fitness of the individuals in the population**

**def evaluate\_population(population):**

**return np.array([fitness\_function(x) for x in population])**

**# Function to select parents based on tournament selection**

**def select\_parents(population, fitness\_scores):**

**tournament\_size = 3**

**selected\_parents = []**

**for \_ in range(len(population)):**

**tournament\_indices = np.random.choice(len(population), size=tournament\_size, replace=False)**

**tournament\_fitness = fitness\_scores[tournament\_indices]**

**selected\_parent\_index = tournament\_indices[np.argmax(tournament\_fitness)]**

**selected\_parents.append(population[selected\_parent\_index])**

**return selected\_parents**

**# Function to perform single-point crossover**

**def crossover(parent1, parent2):**

**crossover\_point = np.random.randint(1, len(parent1))**

**child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))**

**child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))**

**return child1, child2**

**# Function to perform mutation**

**def mutate(individual):**

**if np.random.rand() < mutation\_rate:**

**mutation\_index = np.random.randint(len(individual))**

**individual[mutation\_index] = np.random.uniform(low=-10, high=10)**

**return individual**

**# Genetic algorithm**

**def genetic\_algorithm(population\_size, num\_generations):**

**population = initialize\_population(population\_size)**

**for generation in range(num\_generations):**

**fitness\_scores = evaluate\_population(population)**

**best\_index = np.argmax(fitness\_scores)**

**best\_individual = population[best\_index]**

**print (f"Generation {generation + 1}: Best Fitness = {fitness\_scores[best\_index]:.4f}, Best Individual = {best\_individual:.4f}")**

**parents = select\_parents(population, fitness\_scores)**

**next\_generation = []**

**while len(next\_generation) < population\_size:**

**parent1, parent2 = np.random.choice(parents, size=2, replace=False)**

**child1, child2 = crossover (parent1, parent2)**

**child1 = mutate(child1)**

**child2 = mutate(child2)**

**next\_generation.extend([child1, child2])**

**population = np.array(next\_generation)**

**return population**

**# Run the genetic algorithm**

**final\_population = genetic\_algorithm(population\_size, num\_generations)**

**best\_solution\_index = np.argmax(evaluate\_population(final\_population))**

**best\_solution = final\_population[best\_solution\_index]**

**best\_fitness = fitness\_function(best\_solution)**

**print ("\nFinal Result:")**

**print (f"Best Solution: {best\_solution:.4f}")**

**print (f"Best Fitness: {best\_fitness:.4f}")**

**OUTPUT:**

**Generation 1: Best Fitness = 39.4116, Best Individual = -1.9922**

**Generation 2: Best Fitness = 39.4116, Best Individual = -1.9922**

**Generation 3: Best Fitness = 39.4116, Best Individual = -1.9922**

**...**

**Generation 98: Best Fitness = 50.2376, Best Individual = -1.4993**

**Generation 99: Best Fitness = 50.2376, Best Individual = -1.4993**

**Generation 100: Best Fitness = 50.2376, Best Individual = -1.4993**

**Final Result:**

**Best Solution: -1.4993**

**Best Fitness: 50.2376**

**RESULT:**

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX.No:6** | **Date of** | **Experiment** |  | **Score** | **/10** |

**Implementation of Two Input Sine Function**

**AIM:**

**To Implementation of two input sine function.**

**ALGORITHM:**

**Step 1: Import the math module to access the sin function.**

**Step 2: Define a procedure that takes two inputs and calculates the sine of each input.**

**Step 3: Input values for the two variables.**

**Step 4: Call the calculate\_sine procedure with the input values and store the results.**

**Step 5: Print the results to see the sine values of the inputs.**

**PROGRAM:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from mpl\_toolkits.mplot3d import Axes3D**

**# Define the two-input sine function**

**def two\_input\_sine(x, y):**

**return np.sin(x) + np.sin(y)**

**# Generate 2D grid of points**

**x = np.linspace(0, 2 \* np.pi, 50)**

**y = np.linspace(0, 2 \* np.pi, 50)**

**X, Y = np.meshgrid(x, y)**

**Z = two\_input\_sine(X, Y)**

**# Plot the function in 3D**

**fig = plt.figure()**

**ax = fig.add\_subplot(111, projection='3d')**

**ax.plot\_surface(X, Y, Z, cmap='viridis')**

**ax.set\_xlabel('X')**

**ax.set\_ylabel('Y')**

**ax.set\_zlabel('f(x, y)')**

**ax.set\_title('Two-Input Sine Function')**

**plt.show()**

**OUTPUT:**

**No longer image available**

**RESULT:**

Thus the program was executed and verified successfully.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX.No:7** | **Date of** | **Experiment** |  | **Score** | **/10** |

**Implementation of Three Input Non-linear Function**

**AIM:**

**To Implementation of three input non-linear function.**

**ALGORITHM:**

**Step1: Define the Function**

**Step2: Gather Input Data**

**Step3: Create a Function**

**Step4: Implement Non-Linear Logic**

**Step5: Test the Function**

**Step6: Optimize and Refine**

**Step7: Documentation**

**PROGRAM:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from mpl\_toolkits.mplot3d import Axes3D**

**# Define the three-input nonlinear function**

**def three\_input\_nonlinear(x, y, z):**

**return np.sin(x) + np.cos(y) + np.tan(z)**

**# Generate 3D grid of points**

**x = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)**

**y = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)**

**z = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)**

**X, Y, Z = np.meshgrid(x, y, z)**

**W = three\_input\_nonlinear(X, Y, Z)**

**# Plot the function in 3D**

**fig = plt.figure()**

**ax = fig.add\_subplot(111, projection='3d')**

**ax.scatter(X, Y, Z, c=W, cmap='viridis', s=50)**

**ax.set\_xlabel('X')**

**ax.set\_ylabel('Y')**

**ax.set\_zlabel('Z')**

**ax.set\_title('Three-Input Nonlinear Function')**

**plt.show()**

**OUTPUT:**

**No longer available**

**RESULT**:

Thus the program was executed and verified successfully.