Er. PERUMAL MANIMEKALAI COLLEGE OF ENGINEERING

ACCREDITED BY NBA & NAAC WITH 'A' GRADE

Koneripalli, HOSUR - 635 117.

NAME………………………………………………. **BRANCH………………………………….**

YEAR………………………………………………. **SEM…………………………………….**

PROGRAMME………………………………………………………………………………………………………

REGISTER NO:

Certified that this bonafide record of the work done by the above student of the --- laboratory during the year 20 - 20

STAFF IN CHARGE **HEAD OF THE DEPARTMENT**

Submitted for the examination held on ………………………………………………… at ER. Perumal Manimekalai College of Engineering

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

Sl.no	Date	Experiment	Page	Marks	Remarks

LIST OF EXPERIMENTS

EX.NO	NAME OF THE EXPERIMENTS	CO'S Covered	PO-PSO's Covered
1.	Implementation of fuzzy control/ inference system	CO1	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
2.	Programming exercise on classification with a discrete perceptron	CO2	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
3.	Implementation of XOR with backpropagation algorithm	CO2	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
4.	Implementation of self organizing maps for a specific application	CO1,CO2	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
5.	Programming exercises on maximizing a function using Genetic algorithm	CO3	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
6.	Implementation of two input sine function	CO4	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
7.	Implementation of three input non linear function	CO4	PO1,2,3,4,5,9,10,11,12, PSO1,2,3
8.	Case studies on Real Time Project using Neural Network	CO5	PO1,2,3,4,5,9,10,11,12, PSO1,2,3

EX NO 1: Implementation of fuzzy controller(Washing Machine)

AIM:

To implement fuzzy control /inference system for washing machine.

ALGORITHM:

1.Define Input and Output Variables.

2.Fuzzification: Map crisp input values to fuzzy sets. For example, "small," "medium," and "large" for laundry load size

3. Apply membership functions to determine the degree of membership of each input value in the fuzzy sets.

4. **Rule Base:**

- Define a set of rules that express the relationship between inputs and outputs.

5.Inference Engine:

- Apply fuzzy logic operators (like AND, OR) to combine the fuzzy sets according to the defined rules.
- Use inference methods like Mamdani or Sugeno to determine the fuzzy output sets

6. **Defuzzification:**

- Convert fuzzy output sets back into crisp values.
- Use methods like centroid, maximum membership, or weighted average to calculate the crisp output.

Washing Machine Controller:

To design a system using fuzzy logic, input & output is necessary part of the system.

Main function of the washing machine is to clean cloth without damaging the cloth. In order to achieve it, the output parameters of fuzzy logic, which are the washing parameters, must be given more importance. The identified input & output parameters are:

Input: 1. Degree of dirt

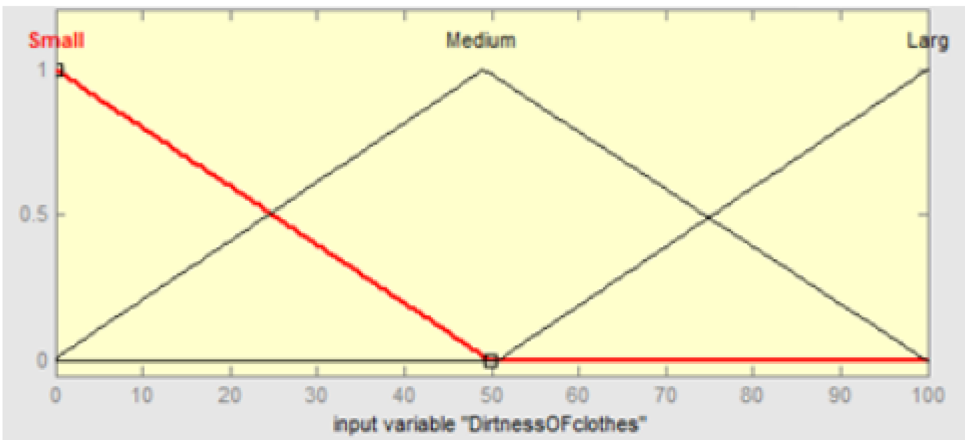
2. Type of dirt

Output: Wash time

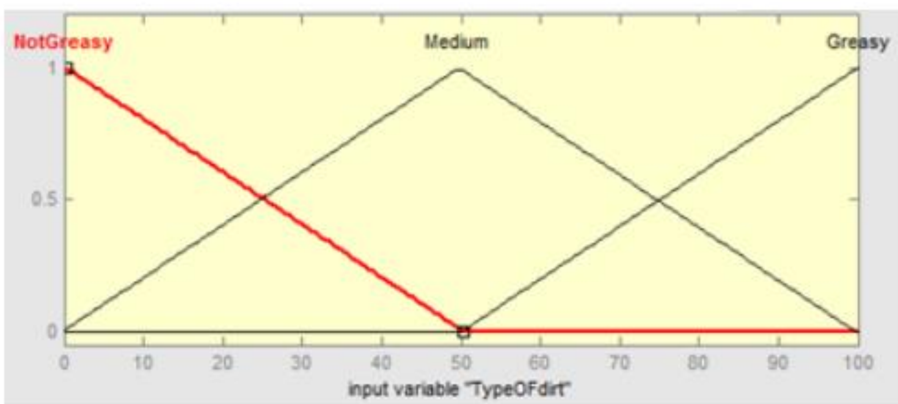
Fuzzy sets:

The fuzzy sets which characterize the inputs & output are given as follows:

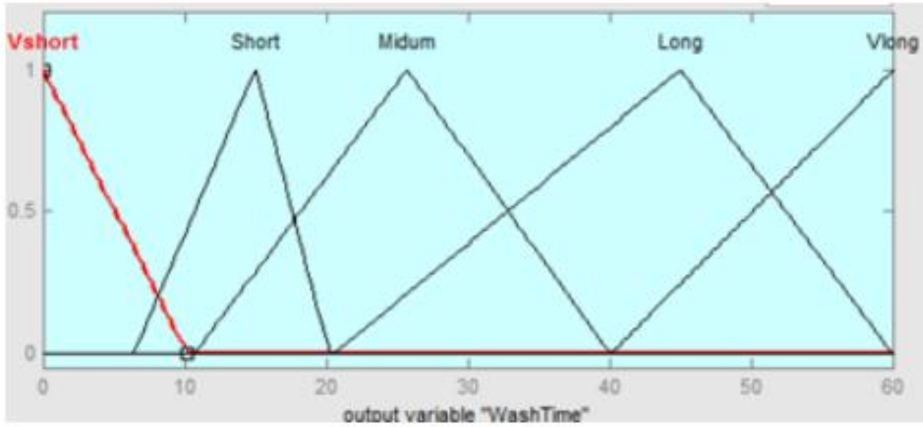
1. Dirtness of clothes



Type of dirt



3. Wash time



Procedure:

Step1: Fuzzification of inputs

For the fuzzification of inputs, that is, to compute the membership for the antecedents, the formula used is,

Step 2: Defining set of rules

	S	M	L
NG	VS	S	M
M	M	M	L
G	L	L	VL

1. If Dirtness of clothes is Large and Type of dirt is Greasy then Wash Time is Very Long;
2. If Dirtness of clothes is Medium and Type of dirt is Greasy then Wash Time is Long;

3. If Dirtness of clothes is Small and Type of dirt is Greasy then Wash Time is Long;
4. If Dirtness of clothes is Large and Type of dirt is Medium then Wash Time is Long;
5. If Dirtness of clothes is Medium and Type of dirt is Medium then Wash Time is Medium
6. If Dirtness of clothes is Small and Type of dirt is Medium then Wash Time is Medium;
7. If Dirtness of clothes is Large and Type of dirt is Not Greasy then Wash Time is Medium;
8. If Dirtness of clothes is Medium and Type of dirt is Not Greasy then Wash Time is Short;
9. If Dirtness of clothes is Small and Type of dirt is Not Greasy then Wash Time is Very Short;

RESULT:

The above fuzzy inference /controller system for washing machine has been implemented successfully.

EX NO :2 Programming exercise on classification with a discrete perceptron

AIM:

TO write the python code for implementation on classification with a discrete perceptron.

ALGORITHM:

1. Define the `DiscretePerceptron` class with necessary attributes and methods.Initialize the weights of the perceptron to zeros, including the bias term.
2. Implement the `predict` method within the `DiscretePerceptron` class.
3. Implement the `train` method within the `DiscretePerceptron` class.
4. Define a simple dataset for binary classification, consisting of input features and corresponding labels.
5. Create an instance of the `DiscretePerceptron` class.
6. Define a set of test inputs
7. Display the predicted outputs for the test inputs along with the true labels

PROGRAM:

```
import numpy as np


class DiscretePerceptron:
  def __init__(self, num_features):
    self.weights = np.zeros(num_features + 1)  # Add one for the bias term


  def predict(self, inputs):
    summation = np.dot(inputs, self.weights[1:]) + self.weights[0]  # Add bias
    if summation > 0:
      activation = 1
```

```
        else:
            activation = 0
        return activation

    def train(self, training_inputs, labels, learning_rate=0.1, epochs=100):
        for _ in range(epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                self.weights[1:] += learning_rate * (label - prediction) * inputs
                self.weights[0] += learning_rate * (label - prediction)

# Example dataset for binary classification
training_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Labels for the dataset
labels = np.array([0, 1, 1, 1])

# Create a DiscretePerceptron instance
perceptron = DiscretePerceptron(num_features=2)
```

```
# Train the perceptron
perceptron.train(training_inputs, labels)

# Test the perceptron
test_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

print("Predictions:")
for inputs in test_inputs:
    prediction = perceptron.predict(inputs)
    print(f"Inputs: {inputs}, Prediction: {prediction}")
```

OUTPUT:

```
Predictions:
Inputs: [0 0], Prediction: 0
Inputs: [0 1], Prediction: 1
Inputs: [1 0], Prediction: 1
Inputs: [1 1], Prediction: 1
```

RESULT:

The above python code for implementation on classification with a discrete perceptron has been executed successfully.

EX NO :3 Implementation of XOR with backpropagation algorithm

AIM:

To write python code for implementing XOR with backpropagation algorithm.

ALGORITHM:

1. **Initialize the Neural Network:**Define the number of input neurons, hidden neurons, and output neurons.
2. **Define Activation Functions:**Define a sigmoid activation function and its derivative.
3. **Forward Propagation:**
   - Compute the weighted sum of inputs and biases for the hidden layer.
   - Apply the activation function to the hidden layer.
4. **Calculate Error:**
   - Calculate the error between the predicted output and the actual output.
5. **Backpropagation:**
   - Compute the derivative of the error with respect to the output layer.
   - Compute the derivative of the error with respect to the hidden layer.
   - Update the weights and biases of both layers using gradient descent.
6. **Repeat:**
   - Repeat steps 3 to 5 for a predefined number of epochs or until convergence.
7. **Training:**
   - Train the neural network by providing the XOR dataset inputs and their corresponding labels.
8. **Testing:**
   - Test the trained neural network on the XOR dataset to see how well it performs.

PROGRAM:

```
import numpy as np

# Define the sigmoid activation function and its derivative
```

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

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

# Define the XOR dataset
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])

# Initialize the neural network parameters
input_size = 2
hidden_size = 2
output_size = 1
learning_rate = 0.1
epochs = 10000

# Initialize weights and biases with random values
np.random.seed(1)
weights_input_hidden = np.random.randn(input_size, hidden_size)
bias_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.randn(hidden_size, output_size)
bias_output = np.zeros((1, output_size))
```

```
# Training the neural network
for epoch in range(epochs):
    # Forward propagation
    hidden_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)
    output = np.dot(hidden_output, weights_hidden_output) + bias_output
    output = sigmoid(output)

    # Backpropagation
    error = y - output
    d_output = error * sigmoid_derivative(output)
    error_hidden = d_output.dot(weights_hidden_output.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    # Update weights and biases
    weights_hidden_output += hidden_output.T.dot(d_output) * learning_rate
    bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden) * learning_rate
    bias_hidden += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

    if epoch % 1000 == 0:
        print(f'Error: {np.mean(np.abs(error))}')

# Testing the trained neural network
```

```
hidden_input = np.dot(X, weights_input_hidden) + bias_hidden
hidden_output = sigmoid(hidden_input)
output = np.dot(hidden_output, weights_hidden_output) + bias_output
output = sigmoid(output)
print("Output After Training:")
print(output)
```

OUTPUT:

```
Error: 0.5049636626294205
Error: 0.49273906275161916
Error: 0.46405350098168774
Error: 0.41015018889002763
Error: 0.38388830097983007
Error: 0.37166661091541
Error: 0.36466799345972645
Error: 0.3601159068862657
Error: 0.35690249763775406
Error: 0.3545020126098154
Output After Training:
[[0.06327997]
 [0.66197348]
 [0.66198146]
 [0.67120806]]
```

RESULT:

The above python code for implementing XOR with backpropagation algorithm has been executed successfully.

EX NO:4.Implementation of self organizing maps for a specific application

AIM:

To write code for implementing self organizing maps for a specific application.

Self Organizing Map (or Kohonen Map or SOM) is a type of Artificial Neural Network which is also inspired by biological models of neural systems from the 1970s. It follows an unsupervised learning approach and trained its network through a competitive learning algorithm. SOM is used for clustering and mapping (or dimensionality reduction) techniques to map multidimensional data onto lower-dimensional which allows people to reduce complex problems for easy interpretation. SOM has two layers, one is the Input layer and the other one is the Output layer.

The architecture of the Self Organizing Map with two clusters and n input features of any sample is given below:

Let's say an input data of size (m, n) where m is the number of training examples and n is the number of features in each example. First, it initializes the weights of size (n, C) where C is the number of clusters. Then iterating over the input data, for each training example, it updates the winning vector (weight vector with the shortest distance (e.g Euclidean distance) from training example). Weight updation rule is given by :

$w_{ij} = w_{ij}(old) + alpha(t) * (x_i^k - w_{ij}(old))$

where alpha is a learning rate at time t, j denotes the winning vector, i denotes the i^{th} feature of training example and k denotes the k^{th} training example from the input data. After training the SOM network, trained weights are used for clustering new examples. A new example falls in the cluster of winning vectors.

Algorithm

Training:

Step 1: Initialize the weights w_{ij} random value may be assumed. Initialize the learning rate α.

Step 2: Calculate squared Euclidean distance.

$$D(j) = \Sigma (wij - xi)^2 \quad \text{where } i=1 \text{ to } n \text{ and } j=1 \text{ to } m$$

Step 3: Find index J, when D(j) is minimum that will be considered as winning index.

Step 4: For each j within a specific neighborhood of j and for all i, calculate the new weight.

$$wij(new)=wij(old) + \alpha[xi - wij(old)]$$

Step 5: Update the learning rule by using :

$$\alpha(t+1) = 0.5 * t$$

Step 6: Test the Stopping Condition.

Below is the implementation of the above approach:

```
import math
class SOM:

    # Function here computes the winning vector
    # by Euclidean distance
    def winner(self, weights, sample):

        D0 = 0
        D1 = 0

        for i in range(len(sample)):

            D0 = D0 + math.pow((sample[i] - weights[0][i]), 2)
            D1 = D1 + math.pow((sample[i] - weights[1][i]), 2)

        # Selecting the cluster with smallest distance as winning cluster

        if D0 < D1:
            return 0
        else:
            return 1

    # Function here updates the winning vector
    def update(self, weights, sample, J, alpha):
        # Here iterating over the weights of winning cluster and modifying them
        for i in range(len(weights[0])):
```

```
                weights[J][i] = weights[J][i] + alpha * (sample[i] - weights[J][i])

        return weights

# Driver code
def main():
    # Training Examples ( m, n )
    T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]
    m, n = len(T), len(T[0])
    # weight initialization ( n, C )
    weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
    # training
    ob = SOM()
    epochs = 3
    alpha = 0.5
    for i in range(epochs):
        for j in range(m):

            # training sample
            sample = T[j]

            # Compute winner vector
            J = ob.winner(weights, sample)

            # Update winning vector
            weights = ob.update(weights, sample, J, alpha)
# classify test sample
    s = [0, 0, 0, 1]
    J = ob.winner(weights, s)

    print("Test Sample s belongs to Cluster : ", J)
    print("Trained weights : ", weights)

if __name__ == "__main__":
    main()
```

Output:

Test Sample s belongs to Cluster : 0

Trained weights : [[0.6000000000000001, 0.8, 0.5, 0.9], [0.3333984375, 0.0666015625, 0.7, 0.3]]

RESULT:

The above code for implementing self organizing maps for a specific application.

Ex.No.5. Programming exercises on maximizing a function using Genetic algorithm

AIM:

To write code for implementing maximize function using genetic algorithm.

Implementation of Genetic Algorithm

The Genetic Algorithm is a stochastic global search optimization algorithm.

It is inspired by the biological theory of evolution by means of natural selection. Specifically, the new synthesis that combines an understanding of genetics with the theory.

Genetic algorithms borrow inspiration from biological evolution, where fitter individuals are more likely to pass on their genes to the next generation.

The algorithm uses analogs of a genetic representation (bitstrings), fitness (function evaluations), genetic recombination (crossover of bitstrings), and mutation (flipping bits).

Parents are used as the basis for generating the next generation of candidate points and one parent for each position in the population is required.

Parents are then taken in pairs and used to create two children. Recombination is performed using a crossover operator. This involves selecting a random split point on the bit string, then creating a child with the bits up to the split point from the first parent and from the split point to the end of the string from the second parent. This process is then inverted for the second child.

For example the two parents:

parent1 = 00000

parent2 = 11111

May result in two cross-over children:

child1 = 00011

child2 = 11100

This is called one point crossover, and there are many other variations of the operator.

Crossover is applied probabilistically for each pair of parents, meaning that in some cases, copies of the parents are taken as the children instead of the recombination operator. Crossover is controlled by a hyperparameter set to a large value, such as 80 percent or 90 percent.

Mutation involves flipping bits in created children candidate solutions. Typically, the mutation rate is set to *1/L*, where *L* is the length of the bitstring.

For example, if a problem used a bitstring with 20 bits, then a good default mutation rate would be (1/20) = 0.05 or a probability of 5 percent.

This defines the simple genetic algorithm procedure. It is a large field of study, and there are many extensions to the algorithm.

Now that we are familiar with the simple genetic algorithm procedure, let's look at how we might implement it from scratch.

```
# initial population of random bitstring

pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]



# tournament selection

def selection(pop, scores, k=3):

# first random selection

selection_ix = randint(len(pop))

for ix in randint(0, len(pop), k-1):

# check if better (e.g. perform a tournament)

if scores[ix] < scores[selection_ix]:

selection_ix = ix

return pop[selection_ix]
```

We can then call this function one time for each position in the population to create a list of parents.

```
# select parents

selected = [selection(pop, scores) for _ in range(n_pop)]
```

The crossover() function below implements crossover using a draw of a random number in the range [0,1] to determine if crossover is performed, then selecting a valid split point if crossover is to be performed.

```
# crossover two parents to create two children
```

```
def crossover(p1, p2, r_cross):
# children are copies of parents by default
c1, c2 = p1.copy(), p2.copy()
# check for recombination
if rand() < r_cross:
# select crossover point that is not on the end of the string
pt = randint(1, len(p1)-2)
# perform crossover
c1 = p1[:pt] + p2[pt:]
c2 = p2[:pt] + p1[pt:]
return [c1, c2]
```

We also need a function to perform mutation.

This procedure simply flips bits with a low probability controlled by the “r_mut” hyperparameter.

```
# mutation operator
def mutation(bitstring, r_mut):
for i in range(len(bitstring)):
# check for a mutation
if rand() < r_mut:
# flip the bit
bitstring[i] = 1 - bitstring[i]
```

We can then loop over the list of parents and create a list of children to be used as the next generation, calling the crossover and mutation functions as needed.

```
# create the next generation
children = list()
for i in range(0, n_pop, 2):
# get selected parents in pair
```

```
p1, p2 = selected[i], selected[i+1]
# crossover and mutation
for c in crossover(p1, p2, r_cross):
# mutation
mutation(c, r_mut)
# store for next generation
children.append(c)
# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
# initial population of random bitstring
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
# keep track of best solution
best, best_eval = 0, objective(pop[0])
# enumerate generations
for gen in range(n_iter):
# evaluate all candidates in the population
scores = [objective(c) for c in pop]
# check for new best solution
for i in range(n_pop):
if scores[i] < best_eval:
best, best_eval = pop[i], scores[i]
print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
# select parents
```

```
selected = [selection(pop, scores) for _ in range(n_pop)]

# create the next generation

children = list()

for i in range(0, n_pop, 2):

# get selected parents in pairs

p1, p2 = selected[i], selected[i+1]

# crossover and mutation

for c in crossover(p1, p2, r_cross):

# mutation

mutation(c, r_mut)

# store for next generation

children.append(c)

# replace population

pop = children

 return [best, best_eval]
```

Genetic Algorithm for OneMax

The onemax() function below implements this and takes a bitstring of integer values as input and returns the negative sum of the values.

```
Def onemax(x):
```

```
return -sum(x)


# define the total iterations
n_iter = 100
# bits
n_bits = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
```

The search can then be called and the best result reported.

```
# perform the genetic algorithm search
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Tying this together, the complete example of applying the genetic algorithm to the OneMax objective function is listed below.

```
# genetic algorithm search of the one max optimization problem
```

```
from numpy.random import randint

from numpy.random import rand


# objective function

def onemax(x):

return -sum(x)


# tournament selection

def selection(pop, scores, k=3):

# first random selection

selection_ix = randint(len(pop))

for ix in randint(0, len(pop), k-1):

# check if better (e.g. perform a tournament)

if scores[ix] < scores[selection_ix]:

selection_ix = ix

return pop[selection_ix]




# crossover two parents to create two children

def crossover(p1, p2, r_cross):

# children are copies of parents by default

c1, c2 = p1.copy(), p2.copy()

# check for recombination

if rand() < r_cross:
```

```
# select crossover point that is not on the end of the string
pt = randint(1, len(p1)-2)
# perform crossover
c1 = p1[:pt] + p2[pt:]
c2 = p2[:pt] + p1[pt:]
return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
for i in range(len(bitstring)):
# check for a mutation
if rand() < r_mut:
# flip the bit
bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
# initial population of random bitstring
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
# keep track of best solution
best, best_eval = 0, objective(pop[0])
# enumerate generations
for gen in range(n_iter):
# evaluate all candidates in the population
scores = [objective(c) for c in pop]
```

```
# check for new best solution
for i in range(n_pop):
if scores[i] < best_eval:
best, best_eval = pop[i], scores[i]
print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
# select parents
selected = [selection(pop, scores) for _ in range(n_pop)]
# create the next generation
children = list()
for i in range(0, n_pop, 2):
# get selected parents in pairs
p1, p2 = selected[i], selected[i+1]
# crossover and mutation
for c in crossover(p1, p2, r_cross):
# mutation
mutation(c, r_mut)



# store for next generation
children.append(c)
# replace population
pop = children
return [best, best_eval]
 # define the total iterations
```

```
n_iter = 100

# bits

n_bits = 20

# define the population size

n_pop = 100

# crossover rate

r_cross = 0.9

# mutation rate

r_mut = 1.0 / float(n_bits)

# perform the genetic algorithm search

best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)

print('Done!')

print('f(%s) = %f' % (best, score))
```

OUTPUT:

```
>0, new best f([1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]) = -14.000

>0, new best f([1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0]) = -15.000

>1, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1]) = -16.000

>2, new best f([0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -17.000

>2, new best f([1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000

>8, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000

f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
```

RESULT:

The above code for implementing maximize function using genetic algorithm has been executed successfully.

EX NO 6. Implementation of two input sine function

AIM:

To write a code for implementing two input sine function**.**

ALGORITHM:

1. **Input Values:**Take two input values, typically denoted as xx and yy, representing the inputs to the sine function.
2. **Compute Product:**Compute the product of the two input values: $z=x\times yz=x\times y$.
3. **Compute Sine:**Compute the sine of the product: $\text{sine}=\sin(z)\text{sine}=\sin(z)$.
4. **Output Sine Value:**Return the sine value as the output of the two-input sine function.

Program:

```
# sin function
import math;

# Function for calculating sin value
def cal_sin(n):

    accuracy = 0.0001;

    # Converting degrees to radian
    n = n * (3.142 / 180.0);

    x1 = n;

    # maps the sum along the series
    sinx = n;

    # holds the actual value of sin(n)
    sinval = math.sin(n);
    i = 1;
    while(True):
```

```
        denominator = 2 * i * (2 * i + 1);
        x1 = -x1 * n * n / denominator;
        sinx = sinx + x1;
        i = i + 1;
        if(accuracy <= abs(sinval - sinx)):
            break;

    print(round(sinx));

# Driver Code
n = 90;
cal_sin(n);
```

Output

Input : 30
Output : 0.86602

RESULT:

The above code for implementing two input sine function has been executed successfully.

EX NO:7 Implementation of three input non linear function.

AIM:

To write a code for implementing three input non linear function.

THEORY:

Non-linear regression algorithms work by iteratively adjusting the parameters of a non-linear function to minimize the error between the predicted values of the dependent variable and the actual values. The specific function used depends on the nature of the relationship between the variables, and there are many different types of non-linear functions that can be used.

Here we are implementing Non-Linear Regression using Python:

Step-1: Importing libraries

Importing all the necessary libraries:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

Step-2: Import Dataset

Importing and reading the dataset: [Dataset Link]

```
# Read the CSV file

df = pd.read_csv('/content/gdp.csv')

# Display the first few rows of the dataframe

print(df.head())
```

```
Year         Value
0  1960  5.918412e+10
1  1961  4.955705e+10
2  1962  4.668518e+10
```

```
3   1963   5.009730e+10
4   1964   5.906225e+10
```

Plot the Original Gdp

Creates a scatter plot of the `Year` (independent variable) vs. `Value` (dependent variable).

plt.figure(figsize=(8, 5))

x_original, y_original = df["Year"].values, df["Value"].values

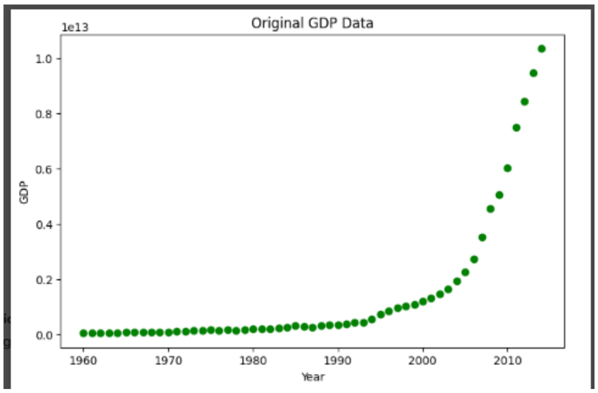
plt.plot(x_original, y_original, 'bo')

plt.ylabel('GDP')

plt.xlabel('Year')

plt.title('Original GDP Data')

plt.show()



Simple logistic model curve

Representing a simple logistic model curve over a range of independent variable values.

#Plot a simple logistic model curve

X_logistic = np.arange(-5.0, 5.0, 0.1)

Y_logistic = 1.0 / (1.0 + np.exp(-X_logistic))

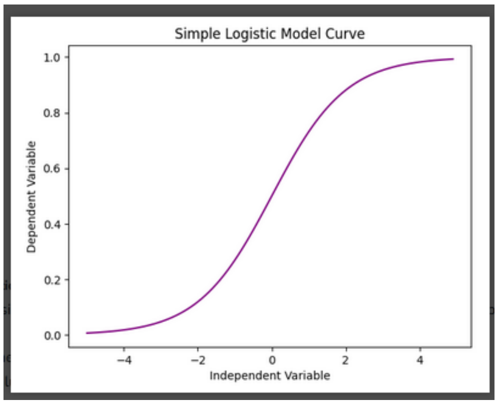
plt.plot(X_logistic, Y_logistic, color='green') # Orange color for the logistic curve

plt.ylabel('Dependent Variable')

plt.xlabel('Independent Variable')

plt.title('Simple Logistic Model Curve')

plt.show()



Define Sigmoid Function

- Implements the sigmoid function (logistic function) that maps any real number to a value between 0 and 1.
- Takes two parameters: `Beta_1` (slope) and `Beta_2` (intercept).
- Assigns initial values for `Beta_1` and `Beta_2` based on intuition or estimation.
- Applies the sigmoid function to the original data with the initial parameter values.

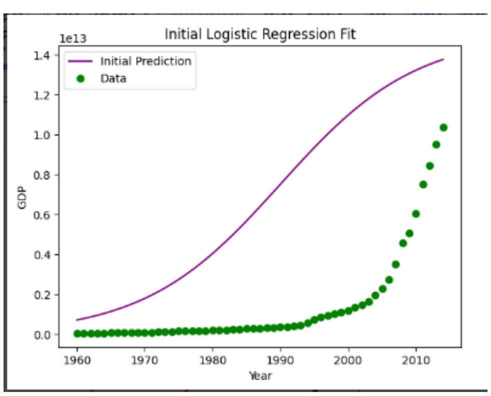
```
#Define the sigmoid function

def sigmoid(x, Beta_1, Beta_2):

 y = 1 / (1 + np.exp(-Beta_1 * (x - Beta_2)))

 return y

# Set initial values for logistic function parameters
```

```
beta_1_initial = 0.10

beta_2_initial = 1990.0

#Apply logistic function to the data

Y_pred_logistic = sigmoid(x_original, beta_1_initial, beta_2_initial)
```

Plot the initial prediction against datapoints

Plots the predicted values (scaled up by 15000000000000 for better visibility) compared to the actual data points.

```
plt.plot(x_original, Y_pred_logistic * 15000000000000., color='purple', label='Initial Prediction')

plt.plot(x_original, y_original, 'go', label='Data')

plt.ylabel('GDP')

plt.xlabel('Year')

plt.legend()

plt.title('Initial Logistic Regression Fit')

plt.show()
```



Normalizing Data

- Divides both `Year` and `Value` by their respective maximum values to scale them between 0 and 1.

```
# Normalize the data

x_normalized = x_original / max(x_original)

y_normalized = y_original / max(y_original)
```

Fitting sigmoid function to normalized data

- Uses the `curve_fit` function from `scipy.optimize` to find the best-fitting parameters for the sigmoid function based on the normalized data.
- Returns the optimal parameters (`popt`) and their covariance matrix (`pcov`).
- Prints the optimized values of `Beta_1` and `Beta_2` found by the curve fitting.

```
from scipy.optimize import curve_fit

# Fit the sigmoid function to the normalized data

popt, pcov = curve_fit(sigmoid, x_normalized, y_normalized)


# Print the final parameters

print("Beta_1 = %f, Beta_2 = %f" % (popt[0], popt[1]))
```

Output:

Beta_1 = 690.451709, Beta_2 = 0.997207

```
#Split data into train/test sets
random_mask = np.random.rand(len(df)) < 0.8
train_x = x_normalized[random_mask]
test_x = x_normalized[~random_mask]
train_y = y_normalized[random_mask]
test_y = y_normalized[~random_mask]

#Build the model using the train set
popt_train, pcov_train = curve_fit(sigmoid, train_x, train_y)

#Predict using the test set
y_hat_test = sigmoid(test_x, *popt_train)

#Evaluate the model
mae = mean_absolute_error(test_y, y_hat_test)
mse = np.mean((y_hat_test - test_y) ** 2)
r2 = r2_score(y_hat_test, test_y)

#Print the evaluation metrics
print("Mean Absolute Error: %.2f" % mae)
print("Mean Squared Error: %.2f" % mse)
print("R2-score: %.2f" % r2)
```

Output:

Mean absolute error: 0.05
Residual sum of squares (MSE): 0.00
R2-score: 0.88

RESULT:

The above code for implementing three input non linear function has been executed successfully

EX NO:8 Case studies on Real Time Project using Neural Network

AIM:

To make a case study on real time project using neural network.

THOERY:

Real-time projects involving neural networks, such as the handwritten digit recognition system described above, demonstrate the practical applications of artificial intelligence in solving real-world problems. By leveraging advanced machine learning techniques, businesses can develop innovative solutions that improve efficiency, accuracy, and user experience in various domains.

Single neuron with 3 inputs example

```
input=[2.1,3.2,3.5]
weights=[2.2,4.3,2.1]
bias=3
output=input[0]*weights[0]+input[1]*weights[1]+input[2]*weights[2]+bias
print(output)
28.730000000000004
```

A single neuron with 4 inputs(informal)

```
input=[2.1,3.2,3.5,3.6]
weights=[2.2,4.3,2.1,3.7]
bias=3
output=input[0]*weights[0]+input[1]*weights[1]+input[2]*weights[2]+input[3]*weights[3]+bia
s
print(output)
42.050000000000004
```

A single neuron with 4 inputs(formal)

```
input=[2,3,5,6]
weights=[0.2,0.3,-0.1,-0.7]
bias=2
output=input[0]*weights[0]+input[1]*weights[1]+input[2]*weights[2]+input[3]*weights[3]+bia
s
print(output)
-1.3999999999999995
```

3 neuron layer with 4 inputs

```
input=[2,3,5,6]
weights1=[0.2,0.3,0.1,0.7]
weights2=[0.3,0.1,0.6,0.4]
weights3=[0.1,0.2,0.1,0.3]
bias1=2
bias2=3
bias3=3
output1=[input[0]*weights1[0]+input[1]*weights1[1]+input[2]*weights1[2]+input[3]*weights1[
3
input[0]*weights2[0]+input[1]*weights2[1]+input[2]*weights2[2]+input[3]*weights2[
input[0]*weights3[0]+input[1]*weights3[1]+input[2]*weights3[2]+input[3]*weights3[
print(output1)
[7.999999999999999, 9.3, 6.1]
```

Why Bias , Weights

```
value= 0.6
weight=-0.3
bias=-0.2
print(value*weight+bias)
print(value*bias)
-0.38
-0.12
value= 0.6
weight=-0.3
bias=0.2
print(value*weight)
print(value*bias)
-0.18
```

```
input=[2,3,5,6]
weights=[[0.2,0.3,0.1,0.7],[0.3,0.1,0.6,0.4],[0.1,0.2,0.1,0.3]]
bias1=[2,3,3]
output1=[input[0]*weights[0][0]+input[1]*weights[0][1]+input[2]*weights[0][2]+input[3]*weig
hts[
input[0]*weights[1][0]+input[1]*weights[1][1]+input[2]*weights[1][2]+input[3]*weights[
input[0]*weights[2][0]+input[1]*weights[2][1]+input[2]*weights[2][2]+input[3]*weights[
print(output1)
[7.999999999999999, 9.3, 6.1]
```

Arrays and their shapes

Dot product

```
import numpy as np
input=[2.1,3.2,3.5,3.6]
weights=[2.2,4.3,2.1,3.7]
bias=3
```

output=np.dot(input,weights)+bias

Layers

```
import numpy as np
input=[[2.1,3.2,3.5,3.6],
[-0.1,1.2,-3.5,1.6],
[0.1,1.2,-0.5,-1.3]]
weights=[[0.2,0.3,0.1,0.7],
[-0.1,-0.1,0.02,-0.2],
[0.22,0.31,0.2,-0.2]]
bias=[2,3,0.7]
weights1=[[-0.1,-0.2,-0.3,0.5],
[-0.4,-0.2,0.12,-1.2],
[0.22,-0.1,0.1,-2.2]]
bias1=[-1,2,-0.7]
Layer1_output=np.dot(input,np.array(weights).T)+bias
Layer2_output=np.dot(Layer1_output,np.array(weights1).T)+bias1
print(Layer2_output)
```

softmax function

y=e**x

- Eulers Number E=2.718281828459045

```
import math
layer_output=[4.8,2.12,3.865]
E=math.e
exp_values=[]
for output in layer_output:
exp_values.append(E**output)
print(exp_values)
[121.51041751873483, 8.331137487687691, 47.703272432688706]
```

Normalization

```
import math
#layer_output=[4.8,2.12,3.865]
layer_output=[1.8342,4.2222,6.1111]
E=math.e
exp_values=[]
for output in layer_output:
exp_values.append(E**output)
print(exp_values)
normal_base=sum(exp_values)
normal_values=[]
for values in exp_values:
normal_values.append(values/normal_base)
print(normal_values)
```

print(sum(normal_values))
[6.260124042427774, 68.18332271669155, 450.83436030915556]
[0.011917739447945289, 0.1298043088803658, 0.8582779516716889]
1.0
Softmax function
Input--> Exponent values--->normalization-->output

RESULT:

The above project using neural networks has been executed successfully.