Lab-3

Implementation of Genetic Algorithms using the objective Function [f(x) = x^2]

```
import numpy as np
import random

population size = 100
num_generations = 50
gene-range = (-10,10)
crossover_rate = (0.7)
mutation_rate = 0.01

def fitness (x):
    return x ** 2

def initialize-population (size, gene-range):
  return np.random.uniform(gene-range[0],
       gene range[1], size)

def selection (population):
    fitness_values = fitness(population)
    probabilities = fitness-values / np.sum (fitness
                              -values)
return population[np.random.choice(len(popula
tion), size=2, p=probabilities)]

def crossover-and-mutate (parent 1, parent 2):
    if random.random() < crossover rate:
        alpha = random.random()
        offspring 1 = alpha * parent1 + (1-alpha)
                        * parent 2
```

offspring 2 = (1- alpha) * parent 1 + alpha * parent 2

else:
    offspring 1, offspring 2 = parent 1, parent 2

if random. random() < mutation_rate:
    offspring 1 = np. random uniform (gene_range[0],
        gene_range[1])
if random random () < mutation_rate
    offspring 2 = np. random. uniform (gene_range[0],
        gene_range[1])
return offspring 1, offspring 2

def genetic_algorithm():
    population = initialize_population (population
        size, gene_range).
    for _ in range (num_generations):
        new_population = []
        for _ in range (population_size // 2):
            parent1, parent 2 = selection (population
            offspring 1, offspring 2 = crossover
            _mutate (parent1, parent 2)
            new_population. extend ([offspring 1,
                offspring 2])

population = np. array (new_population)
best_fitness = np. max (fitness (population))
print (f"Best fitness = { best_fitness }")

Date
Page 9

o/p:-
Best Solution : 0.40866
fitness: 0.167004

Page 2 10.2.1

LAB - 2

Particle Swarm optimization for Function Optim

particle Swarm optimization [PSO] is inspired by the social behaviour of birds flocking or fish schooling. PSO is used to find optimal solution by iteratively improving a candidate solution with regard to a given measure of quality. Implementing the PSO algorithm using Python to optimize a mathematical function.

Applications of pso :

Particle Swarm optimization has a wide range of Application across various fields, Including :-

1) Engineering Design :- optimization of structural designs, component layouts and resource allocation.
2) Artificial Intelligence :- Feature selection, training of neural networks and optimization of algorithms
3) Robotics :- path planning and multi - robot coordination
4) Finance : Portfolio optimization and risk assessment
5) Image processing : Edge detection and Image Segmentation

Telecommunication : Network design and frequency allocation.

Healthcare : Intelligent diagonisis, disease detection and classification, Medical Image segmentation

Page 3

Algorithm steps :

Step 1 :- Define the problem [Mathematical Function] :

-> Identify a mathematical function f(x) that you want to optimize (eg, minimize or maximize)

-> common choices include functions like Rastrigin function or the sphere function

Step 2 : Initialize parameters :

-> Set parameters such as :

(•) Number of particles 'N'

(•) Inertia weight w (controls exploration vs exploitation)

(•) Cognitive coefficient c_1 (influences how much a particle is attracted to its own best position)

(•) Social coefficient c_2 (influences how much a particle is attracted to the global best position)

Step 3 :- Initialize particles :

(•) generate an initial population of particles with random positions and velocities in the solution space

Step 4 :- Evaluate fitness :- calculate the fitness of each particles with random position and velocities in the solution space

Step 5 :- Update Velocities and positions :

-> update the velocity v_i and position x_i of each particle using the formula : $v_i = w \cdot v_i + c_1 \cdot r_1 \cdot (pbest_i - x_i) + c_2 \cdot r_2 (gbest - x_i)$ $x_i = x_i + v_i$

-> Here, pbest$_i$ is the personal best solution of particle i, gbest is the global best position among all

particles and r_1 are random numbers betw[een]
0 and 1.

6. Iterate:- Repeat the evaluation, updating & position adjustment for a set of number iter[ations] or until a convergence criterion is met (eg. [no] significant improvement in the best solution).

7. O/P the best solution:-
→ After completing the iterations, output the best solution found, including its position and fitness value.

3/10/24

THURSDAY

Date 14.11.24
Page 10

LAB-5

Ant colony Optimization for Travelling Salesman problem

```
import numpy as np

cities = np.array([[0,0],[1,5],[5,3],[6,1],
                   [3,6], [7,7]])
num_cities = len(cities)
distances = np.zeros((num_cities),(num_cities))
for i in range(num_cities):
    for j in range(num_cities):
        distances[i][j] = np.linalg.norm(cities[i]
                          - cities[j])

num_ants = 10
num_iterations = 100
alpha = 1.0
beta = 2.0
rho = 0.5
initial_pheromone = 1.0
pheromone = np.ones((num_cities, num_cities))
                    * initial_pheromone

def calculate_probabilities(ant_position, visited):
    probabilities = []
    for city in range(num_cities):
        if city not in visited:
            prob = (pheromone[ant_position]
    [city] ** alpha) * ((1/ distances[ant_position]
        [city]) ** beta)
            probabilities.append(0)
    probabilities /= np.sum(probabilities)
    return probabilities

best_path = None
best_path_length = float('inf')

for _ in range(num_ants):
    visited = []
    path_length = 0
    ant_position = np.random.randint(num_cities)
    visited.append(ant_position)
    for _ in range(num_cities - 1):
        probabilities = calculate_probabilities(ant
                        _position, visited)
        next_city = np.random.choice(range(num_ci
                    ties), p=probabilities)
        path_length += distances[ant_position]
                       [next_city]
        ant_position = next_city
        visited.append(next_city)

path_length += distances[ant_position][visited[0]]
all_paths.append(visited)
path_lengths.append(path_length)

    if path_length < best_path_length:
        best_path_length = path_length
        best_path = visited

pheromone *= (1 - rho) #Evaporate pheromones
for path, length in zip(all_paths, path_lengths):
```

Date
Page

```
for i in range(len(path) -1) :
        phermone[path[i]][path[i+1]] += 1.
phermone[path[-1]][path[0]] += 1.0 / length

# output the best solution
print ("best path found :", best_path)
print ("shortest path length :", best_path_length
```

o/p :

best_path found : [5, 2, 3, 0, 1, 4]
Shortest path length : 24.2491 //..

THURSDAY

Date 21/11/24
Page 13

LAB-6

Cuckoo Search Algorithm

```
import numpy as np

def levy_flight(lambda, d):
    sigma_u = (np.math.gamma(1+ lambda) *
               np.sin(np.pi * lambda / 2) /
               np.math.gamma((1+ lambda) / 2)
               * lambda * 2 ** ((lambda - 1) / 2)))
               ** (1 / lambda)
    u = np.random.normal(0, sigma_u, size = d)
    v = np.random.normal(0, 1, size = d)
    step = u / np.abs(v) ** (1 / lambda)
    return step

def cuckoo_search(fun, dim, bounds, num_nests=25,
                  max_iteration, pa=0.25, alpha
                  = 0.01, beta = 1.5):
    nests = np.random.uniform(bounds[0],
                              bounds[1], (num_
                              nests, dim))
    fitness = np.array([func(nest) for nest in
                        nests])

    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        for i in range(num_nests):
            step = alpha * levy_flight(beta, dim)
            num_nest = nests[i] + step * (nests[i]
                       - best_nest)
```

```
new_nest = np.clip(new_nest, bounds[0],
                   bounds[1])
new_fitness = func(new_nest)
if new_fitness < fitness[i]:
    nests[i] = new_nest
    fitness[i] = new_fitness

worst_nests_idx = np.argsort(fitness)[-int(pa *
      num_nests):]
nests[worst_nests_idx] = np.random.uniform
    (bounds[0], bounds[1], (len(worst_nests_idx),
      dim))
fitness[worst_nests_idx] = np.array([func
    (nest) for nest in nests[worst_nests_idx]])

min_idx = np.argmin(fitness)
if fitness[min_idx] < best_fitness:
    best_fitness = fitness[min_idx]
    best_nest = nests[min_idx]
print(f"Iteration {iteration + 1}/{max_it
    , Best Fitness: {best_fitness}")
return best_nest, best_fitness

dim = 5
bounds = [-5.12, 5.12]
num_nests = 25
max_iteration = 100
pa = 0.25

print(" Best Solution found : ", best_Solution)
print("Best fitness (objective value): ",
       best_fitness)
```

O/P:

Iteration 1/50, Best Fitness : 154.9275
Iteration 11/50, Best Fitness : 94.9635
Iteration 21/50, Best Fitness : 50.3612
Iteration 31/50, Best Fitness : 33.2598
Iteration 41/50, Best Fitness : 19.23426
Iteration 50/50, Best Fitness : 19.23426

Best Solution found : [0.0052 -1.2056 0.2438
-0.875 1.0059]
Best fitness (objective value) : 19.23426

04.11.

LAB - 7

Grey Wolf Optimization Algorithm :-

```
import numpy as np

def objective_function(x):
    return sum(x**2)

def gwo(obj_func, dim, n_wolves, max_iter,
        lower_bound, upper_bound):
    wolves = np.random.uniform(lower_bound,
             upper_bound, (n_wolves, dim))

    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)
    alpha_score = float('inf')
    beta_score = float('inf')
    delta_score = float('inf')

for iter in range(max_iter):
    for i in range(n_wolves):
        fitness = obj_func(wolves[i])
        if fitness < alpha_score:
            delta_score, delta_pos = beta_score,
                beta_pos.copy()
            beta_score, beta_pos = alpha_score,
                alpha_pos.copy()
            alpha_score, alpha_pos = fitness,
                wolves[i].copy()
        elif fitness < alpha_score:
            delta_score, delta_pos = beta_score,
                beta_pos.copy()
            beta_score, beta_pos = fitness, wolves[i].copy()
        elif fitness < delta_score:
            delta_score, delta_pos = fitness, wolves[i].copy()

    a = 2 - 2 * (iter / max_iter)
    for i in range(n_wolves):
        for j in range(dim):
            r1, r2 = np.random.random(), np.random.
                random()
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] -
                wolves[i][j])
            X1 = alpha_pos[j] - A1 * D_alpha

            r1, r2 = np.random.random(), np.random.random()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_pos[j] - wolves[i][j])
            X2 = beta_pos[j] - A2 * D_beta

            r1, r2 = np.random.random(), np.random.
                random()
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - wolves[i]
            X3 = delta_pos[j] - A3 * D_delta

            wolves[i][j] = (X1 + X2 + X3) / 3

        wolves[i] = np.clip(wolves[i], lower_bound
                    upper_bound)
return alpha_score, alpha_pos
```

Date / /
Page

```
dimension = 5
num_wolves = 10
max_iterations = 100
lower = -10
upper = 10

best_score, best_position = gwo(objective_function, dimension, num_wolves, max_iteration, lower, upper)

print("Best solution (position):", best_position)
print("Best objective value:", best_score)
```

O/P:- Best Solution (position): [-5.5723 -5.8920 -6.1442 -5.8431]
Best objective value: 1.7742e-12

28.11

Date __/__/__
Page ____

~~Lab~~ Program 6
Parallel Cellular Algorithm

```
import numpy as np

def fitness_function(x, y):
    return -x**2 - y**2 + 10*x + 8*y

grid_size=(5,5)
num_Iteration= 5
search_space= (10,10)
neighborhood_size=1

population = np.random.uniform(search_space[0],
  search_space[1], (*grid_size, 2))

def get_best_neighbor(population, x, y, neighborhood_size):
    x_min, x_max= max(0, x-neighborhood_size,
                  min(grid[0], x+neighborhood_size+1)
    y_min, y_max = max(0, y-neighborhood),
                  min(grid[1], y+neighborhood_size+1)
    neighborhood= population[x_min: x_max, y_min:
                  y_max]. Reshape(-1,2)
    return max(neighborhood, key=lambda ind:
       fitness_function(ind[0], ind[1]))

for iteration in range(num_iteration):
    new population = np.copy(population)
    for x in range(grid_size[0]):
        for y in range(grid_size[1]):
            best neighbor = get best neighbor(population,
                        x, y, neighborhood_size)
```

Date ___
Page ___

```
new_population[x, y] = np.clip(best_neighbour +
                        mutation, search_space[0],
                        search_space[1])
population = new_population

    fitness = np.array([[fitness_function(
                ind[0], ind[1]) for ind in row]
                for row in population])
    best_idx = np.unravel_index(fitness.argmax(),
                fitness.shape)
    print(f"Iteration {iteration+1}: Best fitness=
            {fitness[best_idx]:.2f}, Best Cell=
            {population[best_idx]}")

# output final best solution
best_idx = np.unravel_index(fitness.argmax(),
                fitness.shape)
print(f"\nBest solution: {x, y} = {population[best
        _idx]}, fitness[best_idx]:.2f}")
```

output

Iteration 1: Best fitness= 40.60, Best cell =
[4.78053407 4.59665822]
Iteration 2: Best fitness= 40.99, Best Cell=
[5.070018 3.90911838]
Iteration 3: Best fitness = 40.98, Best cell
[4.86435292 3.93221728]
Iteration 4: Best fitness = 40.97, Best cell
: [5.05377386 3.84260916]
Iteration 5: Best fitness = 40.98,
Best cell=[5.12205527 4.05919827]

Expression

Program Genetic Algorithm

```
import numpy as np

def fitness_function(x):
    return -x**2 + 5*x+6


population_size =10
mutation_rate =0.1
crossover_rate =0.8
num_generations =50
search_space = (-10, 10)

population= np.random.uniform (search_space
[0], search_space[1], population_size)

for generation in range(num_generations):
    fitness= np.array([fitness_function(ind)
            for ind in population])
    adjusted_fitness= fitness- fitness.min() +1
    probabilities = adjusted_fitness / adjusted_
                fitness.sum()
    selected = np.random.choice (population
    size = population_size, p=probabilities)

    next_population= []
    for i in range (0, population_size, 2):
        parent1 = selected[i]
        parent2 = selected[(i+1) % population_size]
        if np.random.rand() < crossover_rate:
            offspring1 = (parent1 + parent2) /2
            offspring2 = (parent2+parent1) /2
        else
```

Date ___/___/___
Page ______

```
offspring += np.random.uniform(-1,1)
if np.random.rand() < mutation_rate
else 0
offspring += np.random.uniform(-1,1)
if np.random.rand() < mutation_rate
else 0

best_idx = fitness.argmax()
print(f"Generation {generation+1}:
    Best fitness = {fitness[best_idx]:.2f}
    Best Individual = {population[best_idx]
    :.2f}")

final_fitness = np.array([fitness_function(ind)
                 for ind in population])
best_idx = final_fitness.argmax()
print(f"\n Best Solution: x = {population[
    best_idx]:.2f}, fitness = {final_fitness
    [best_idx]:.2f}")
```

output

Best solution x = 2.48, fitness = 12.20