Genetic Algorithm for optimization problem

Genetic algorithms use the process of natural selection and genetics. They, populations of potential individuals is evolved over generations to find best solutions to a problem. The idea is to select the fittest individuals from population, use them to create next generations. This leads to better solutions due to crossover, selections and mutations.

The step for implementation/algorithm:

→ define problem: Create mathematical function that needs to be optimized

→ Initialize population: Generate initial population of potential solutions.

→ Evaluate fitness: Evaluate fitness of each individual in population

→ Selection: Select individuals based on their fitness to reproduce and generate offspring for next generation.

→ Crossover: Process of combining two parent solution to create offspring

→ Mutation: It involves genetic diversity by introducing small changes to the individual's structure.

→ Iteration: Repeat the evaluation, solution, crossover and mutation processes for fixed number of

Applications:

1) Optimization: They are commonly used in optimization problem where we have to maximize/minimize a given objective function value.

2) DNA Analysis: GAs have been used to determine structure of DNA using spectometric data about sample.

3) Neural Networks: GAs are also used to train neural networks, particularly recurrent neural networks.

4) Image Processing: GAs are used for digital processing tasks as well as dense pixel matching

5) Robot Trajectory Generation: They have been used to plan the paths which a robot arm takes by moving from one point to another

Pseudocode

```
Function GeneticAlgorithm
    set population_size = 100
    let mutation_rate = 0.1
    let crossover_rate = 0.7
    let num_generations = 50
```

Page 3

```
let value_range = (-10, 10)

Function InitializePopulation (size, range)
    return randomly generate 'size' individuals
    within 'range'

Function Evaluatefitness(population)
    return [f(x) for x in population]

Function SelectParent (population, fitness_values)
    let total_fitness = sum(fitness_values)
    let selection_probabilities = fitness_values/total_fitness

    return randomly selected two indices based
    on probabilities

function crossover (parent1, parent2)
    if random() < crossover_rate then
        let alpha = Random()
        return (alpha * parent1 + (1-alpha) *
        parent2, alpha * parent2 + (1-alpha)
        * parent1)
    else
        return (parent1, parent2)

function Mutate (offspring)
    if random() < mutation_rate, then
        return randomly generated a new
        value within value range
    else
        return offspring

let population = InitializePopulation (population_size,
                                       value_range)
```

for generation from 1 to num_generations
do

set fitness_values = Evaluate Fitness
(Population)

set best_index = Index of max(fitness_values)
print ("Generation", generation,
"Best solution = ", population(best_index)
", Fitness = ", fitness_values[best_index]

Else

set new_population = []

for i from 1 to (population_size/2) do
set (parent1, parent2) = (selectParents
(population, fitness_values)
set (offspring1, offspring2) = (crossover
(parent1, parent2)
Add Mutate(offspring1) to new_population
Add Mutate(offspring2) to new_population

Set population = new_population

Set fitness_values = Evaluatefitness (population)
set best_index = Index of max(fitness_values)
Print "that Best solution = ", population[best_index], "
fitness = ", fitness_values[best_index]

END function

Date 14, 11, 24
Page 7

3) Ant Colony Optimization for Traveling Saleman Problem

Initialize parameters
n_cities, alpha, beta, rho, Q, iterations, ant_count, initial_pheromone

Generate cities with random coordinates.

Calculate distance matrix: distances[i][j] = Euclidian distance between cities i and j
Initialize pheromone matrix: pheromone[i][j] = initial_pheromone

Define functions:
- calc_distance(city1, city2): Return Euclidian distance
- choose_next_city(current, visited, pheromone, distances):
  - Compute probability for each unvisited city
  - Return city with highest probability.
- simulate_ant(pheromone, distances):
  - for each ant, construct a tour by selecting cities based on pheromone and distance.
  - Calculate tour length and update best tour if shorter
  - Return best tour and best_length
- update_pheromones(pheromone, all_tours, best_tour, best_length, Q, rho):
  - Evaporate pheromones: pheromone *= (1-rho)
  - Deposit pheromone on all tours based on tour quality
  - Reinforce pheromone on best tour

Main loop (ACO algorithm):

Date 14/11/24
Page 8

- Initialize best_tour and best_length.
- for each iteration (1 to iterations):
  - Run simulateAnt() to get best tour and best_length
  - Update best solution if necessary
  - Call update_pheromones()

Return best_tour, best_length
Display best_tour and plot path.

Pseudo code

```
Initialize parameters
    n_cities = 10
    alpha : 1.0
    beta = 5.0
    rho = 0.5
    Q = 100
    iterations = 100
    ant_count = 50
    initial_pheromone = 1.0

np.random.seed(42)
cities = np.random.rand(n_cities, 2)

function calc_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0]) ** 2 +
                   (city[1] - city[1] ** 2)

distances = np.zeros((n_cities, n_cities))
for i in range(n_cities):
    for j in range(i+1, n_cities):
        dist = calc_distance(city[i], cities[j])
        distances[i][j] = dist
        distances[j][i] = dist

pheromone = np.ones((n_cities, n_cities)) * initial_pherma

function choose_next_city(current_city, visited_city, pheromone,
                          probabilities[] distance)
    total = 0.0
    for i in range(n_cities):
        if i not in visited:
```

```
pheromone_level = pheromone[current_city][i] ** alpha
distance_heuristic = (1.0 / distances[current_city][i]) ** beta
prob = pheromone_level * distance_heuristic
probabilities.append(prob)
else:
    probabilities.append(0)

return random.choices(range(n_cities), probabilities)[0]

function simulate_ants(pheromone, distances):
    for _ in range(ant_count):
        tour = [random.randint(0, n_cities-1)]
        visited.add(tour[0])

    while len(visited) < n_cities:
        current_city = tour[-1]
        tour.append(next_city)
        visited.add(next_city)

    tour_length = sum(distances[tour][i][tour[i+1]]
                      for i in range(n_cities-1)) +
                  distances[tour[-1]][tour[0]]

    if tour_length < best_length:
        best_length = tour_length
    return best_tour, best_length, all_tours

function update_pheromones(pheromone, all_tours,
                           best_tours, best_length, Q, rho):
```

```
pheromone *= (1 - rho)
for tour, length in all_tours:
    for i in range(len(tour)-1):
        pheromone[tour[i]][tour[i+1]] += Q / length
for i in range(len(best_tour)-1):
    pheromone[best_tour[i]][best_tour[i+1]] += Q / best_length
    pheromone[best_tour[i+1]][best_tour[i]] += Q / best_length.

function aco_tsp():
    for iteration in range(iterations):
        if (best_length < best_overall_length):
            best_overall_length = best_length
            best_overall_tour = best_tour
        update_pheromones(pheromone, all_tours,
        best_tours, best_length, Q, rho)
    return best_overall_tour, best_overall_length

best_tour, best_length = aco_tsp()
x = [cities[city][0] for city in best_tour] +
        [cities[best_tour[0]][0]]
y = [cities[city][1] for city in best_tour] + [cities[best_
        [0][1]]
plt.figure(figsize = (8,6))
plt.plot(x, y, marker = 'o', color = 'b', linestyle = '-', marker
best_tour, best_length = aco_tsp()
```

21/1/24

Particle Swarm Optimization

Initialize swarm
  for each particle i in swarm:
    initialize position (x_i) randomly within problem bounds
    initialize velocity (v_i) randomly
    set personal best position pbest_i = x_i
    set personal best value f(pbest_i)

set global best (gbest)
  set gbest = position of the particle with the best value f(pbest_i)

for each iteration (t=1 to max_iterations):
  for each particle i:
    Evaluate fitness f(x_i) at current position x_i
    if f(x_i) < f(pbest_i):
      update pbest_i = x_i
      update f(pbest_i) = f(x_i)
    if f(pbest_i) < f(gbest):
      update gbest = pbest_i

  for each particle i:
    update velocity v_i using the formula:
    v_i(t+1) = w * v_i(t) + c1 * r1 * (pbest_i - x_i)
        + c2 * r2 * (gbest - x_i)

  where:
    w = inertia weight (controls exploration v/s exploitation)
    c1, c2 = cognitive and social learning factors
    r1, r2 = random values between 0 and 1.
    pbest_i = personal best position of particle i
    gbest = global best position of the swarm.

  for each particle i:

update position x_i using the formula:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

Ensure that x_i stays within bounds (boundary handling)

Output global best position gbest and its corresponding value $f(gbest)$.

27/11/24

4) Cuckoo Search

Date 23/11/24
Page 12

```
def cuckoo_search():
    N=50
    max_iter=100
    pa=0.25
    alpha=0.01

    nests=initialize_nests(N)
    fitness=[evaluate_fitness(nest) for nest in nests]
    best_nest=min(nests, key=lambda nest:
    evaluate_fitness(nest)

    for iteration in range(max_iter):
        new_nests=[]
        for nest in nests:
            new_nest=levy_flight(nest,alpha)
            new_nests.append(new_nest)

        new_fitness=[evaluate_fitness(nest) for nest in
            new_nests]
        for i in range(N)
            if new_fitness[i]<fitness[i]:
                nests[i]=new_nests[i]
                fitness[i]=new_fitness[i]

    if random.random()<pa:
        worst_nests=select_worst_nests(nests,
                                       fitness
        replace_worst_nests(worst_nests)
```

```
best_nest = min(nests, key=lambda nest: evaluate_
    fitness(nest))

print(f"Iteration {iteration + 1}, Best solution:
        {best_nest}")
return best_nest

initialize_nests();

for i to N:
    nest_i = Randomly generate a vector within
             the bounds: [lowerbound, upper_bound]
nests[i] = nest_i
return nests.

evaluate_fitness():
    for a given nest 'nest_i':
    fitness_i = f(nest_i)
    return fitness_i

levy_flight():
    s = Random sample from Gaussian distribution:
        s ~ N(0,1)
    step = alpha * s / |s|^(1/3)
    new_nest_i = nest_i + step
    return new_nest_i

select_worst_nests():
    for given nest and their fitness values:
    sorted_indices = sort(fitness)
    N_worst = floor(N * p_a)
    worst_nests = nests[sorted_indices[N-N_worst
Return worst_nests.
```

Date 21/11/24
Page

replace_worst_nests():

for each worst nest 'worst_nest':
    new_nest = Randomly generate a vector
    with n bacons
    nests[worst_nest] = new_nest
return nests

21/11/24

Date 28/11/24
Page 16

3) Grey Wolf Optimizer (GWO)

Initialize Parameters:
- Set N (number of wolves), D (dimensions), Max_Iter (max iterations), bounds.

Initialize Population:
- Randomly initialize the positions of N wolves in D-dimensional space.

Evaluate fitness
- for each wolf, calculate the fitness using the objective function.

Identify Alpha, Beta, Delta wolves:
- sort wolves by fitness.
- Alpha ← best wolf
- Beta ← second best wolf
- Delta ← third best wolf

Main optimization Loop (for t = 1 to Max_Iter):
- Calculate the decreasing factor a = 2 - t*(2/Max_Iter)
- for each wolf i:
  - Calculate the fitness of wolf i
  - If fitness of wolf i < fitness of alpha
    - Delta ← Beta
    - Beta ← Alpha
    - Alpha ← wolf i (update alpha position and fitness)
- Update position of Each wolf:
  for each wolf i:
  - Calculate A = 2*r1 - a
  - Calculate C = 2*r2

Date 29/11/24
Page 16

- Calculate D = |C * Alpha_position of wolf_i]
- Update position of wolf i : position[i] =
  Alpha - A * D
- Ensure position[i] stays within bounds

Check convergence (optional):
- If stopping criteria are met, stop early.

Output Best Solution:
- The position of the Alpha wolf is the Best solution find

Objective function:

return np.sum(x**2)

6/12

Activate Windows

Date 28.11.24
Page 16

3) Grey Wolf Optimizer (GWO)

Initialize Parameters:
- Set N (number of wolves), D (dimensions), Max_Iter (max iterations), bounds.

Initialize Population:
- Randomly initialize the positions of N wolves in D-dimensional space.

Evaluate fitness
- for each wolf, calculate the fitness using the objective function.

Identify Alpha, Beta, Delta wolves:
- sort wolves by fitness.
- Alpha ← best wolf
- Beta ← second best wolf
- Delta ← third best wolf

Main optimization Loop (for t = 1 to Max_Iter):
- Calculate the decreasing factor a = 2 - t*(2/Max_Iter)
- for each wolf i:
  - Calculate the fitness of wolf i
  - If fitness of wolf i < fitness of alpha
    - Delta ← Beta
    - Beta ← Alpha
    - Alpha ← wolf i (update alpha position and fitness)
- Update position of Each wolf:
  for each wolf i:
  - Calculate A = 2*r1 - a
  - Calculate C = 2*r2

Date 29/11/24
Page 15

- Calculate D = |C * Alpha_position of wolf_i|
- Update position of wolf i: position[i] = Alpha - A * D
- Ensure position[i] stays within bounds

Check convergence (optional):
- If stopping criteria are met, stop early.

Output Best solution:
- The position of the Alpha wolf is the best solution find

Objective function:

return np.sum(x**2)

6/6

18/12/24

Date: 5, 11, 24
Page: 13

6) Parallel Cellular Algorithm

```
def update_cell(cell_index, grid, size):
    x, y = cell_index
    neighbours = [ ((x-1)%size, y), ((x+1)%size, y)
                   (x, (y-1)%size), (x, (y+1)%size)
    ]
    new_state = sum(grid[n[0], n[1]] for n in
                    neighbours)%2
    return (x, y, new_state)

def parallel_update(grid, size, num_iterations):
    pool = Pool(processes=4)
    for iteration in range(num_iterations):
        print(f"Iteration {iteration+1}:")
        indices = [(x,y) for x in range(size)
                   for y in range(size)]
        result = pool.starmap(update_cell
                 [(i, grid, size) for i in indices])
        for x, y, new_state in result:
            grid[x,y] = new_state
    return grid
gridsize = 100
grid = np.random.randint(2, size=(grid_size, g
num_iterations = 2
updated_grid = parallel_update(grid, gri
                               num_iterations)
```

Date 5/17/24
Page 18

7) Gene Expression Algorithm

Initialize population with random chromosomes

for generation = 1 TO max_generations DO
    FITNESS_LIST = [ ]
    for each chromosomes IN Population DO
        fitness = COMPUTE_FITNESS (chromosome)
        Add fitness to fitness_list
END FOR

NEW_POPULATION = [ ]
FOR i = 1 to population size/2 DO
    parent1 = SELECT_PARENT (population, FITNESS_LIST)
    parent2 = SELECT PARENT (population, FITNESS_LIST)
    (child1, child2) = CROSSOVER (parent1, parent2)
    child1 = Mutate (child1)
    child2 = Mutate (child2)
    Add child1 to new_population
    Add child2 to new_population
END FOR

population = NEW_POPULATION
END FOR

best_chromosome = find_best (population, fitness_list)
Output best_chromosome, fitness (best_chromosome)

6/12/24