1BM22CS241

Sanjeet Prajwal Pandit

Partical_Swarm_Optimization

```
import numpy as np
import matplotlib.pyplot as plt

# Rastrigin Function (Objective Function)
def rastrigin(x):
    return 10 * len(x) + sum(xi**2 - 10 * np.cos(2 * np.pi * xi) for xi in
x)

# PSO Algorithm
class PSO:
    def __init__(self, objective_function, n_particles, n_dimensions,
max_iter, bounds):
        self.objective_function = objective_function  # The objective
function to minimize
        self.n_particles = n_particles  # Number of particles in the swarm
        self.n_dimensions = n_dimensions  # Number of dimensions (features)
        self.max_iter = max_iter  # Maximum number of iterations
        self.bounds = bounds  # Bounds for the search space (min, max)

        # Initialize the swarm's positions and velocities
        self.positions = np.random.uniform(bounds[0], bounds[1],
(n_particles, n_dimensions))
        self.velocities = np.random.uniform(-1, 1, (n_particles,
n_dimensions))

        # Initialize personal best positions and values
        self.pbest_positions = np.copy(self.positions)
        self.pbest_values = np.apply_along_axis(self.objective_function, 1,
self.positions)

        # Initialize global best position and value
        self.gbest_position =
self.pbest_positions[np.argmin(self.pbest_values)]
        self.gbest_value = np.min(self.pbest_values)

        # PSO parameters (hyperparameters)
        self.w = 0.5  # Inertia weight
        self.c1 = 1.5  # Cognitive coefficient
        self.c2 = 1.5  # Social coefficient

    def update_velocity(self, i):
        # Update velocity for particle i
        r1, r2 = np.random.rand(2)  # Random numbers for the update rule
        inertia = self.w * self.velocities[i]
        cognitive = self.c1 * r1 * (self.pbest_positions[i] -
self.positions[i])
        social = self.c2 * r2 * (self.gbest_position - self.positions[i])
        self.velocities[i] = inertia + cognitive + social

    def update_position(self, i):
        # Update position for particle i
        self.positions[i] += self.velocities[i]
```

```
        # Ensure particles are within bounds
        self.positions[i] = np.clip(self.positions[i], self.bounds[0],
self.bounds[1])

    def optimize(self):
        # Perform optimization over a number of iterations
        for t in range(self.max_iter):
            for i in range(self.n_particles):
                # Update particle velocity and position
                self.update_velocity(i)
                self.update_position(i)

                # Evaluate fitness (objective function value)
                fitness = self.objective_function(self.positions[i])

                # Update personal best if necessary
                if fitness < self.pbest_values[i]:
                    self.pbest_values[i] = fitness
                    self.pbest_positions[i] = self.positions[i]

            # Update global best if necessary
            min_pbest_value = np.min(self.pbest_values)
            if min_pbest_value < self.gbest_value:
                self.gbest_value = min_pbest_value
                self.gbest_position =
self.pbest_positions[np.argmin(self.pbest_values)]

            # Print the best result at each iteration (optional)
            print(f"Iteration {t+1}/{self.max_iter}, Best Value:
{self.gbest_value}")

        return self.gbest_position, self.gbest_value


# Set parameters
n_particles = 30  # Number of particles in the swarm
n_dimensions = 2  # Number of dimensions (for Rastrigin, 2D)
max_iter = 100  # Maximum number of iterations
bounds = (-5.12, 5.12)  # Bounds for Rastrigin function

# Initialize and run PSO
pso = PSO(objective_function=rastrigin,
          n_particles=n_particles,
          n_dimensions=n_dimensions,
          max_iter=max_iter,
          bounds=bounds)

best_position, best_value = pso.optimize()
print("\nOptimal solution:", best_position)
print("Objective function value at optimal solution:", best_value)

# Plot the convergence of the objective function
iterations = np.arange(1, max_iter + 1)
best_values = [pso.gbest_value for _ in range(max_iter)]  # For
illustration, same best value per iteration

plt.plot(iterations, best_values, label='Best Value Over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Best Objective Value')
plt.title('PSO Convergence')
```

```
plt.legend()
plt.show()
```

```
Optimal solution: [-2.04997664e-09  7.71996126e-10]
Objective function value at optimal solution: 0.0
```