Assignment 2: Python Code for BFGS and GA

AE413: Optimization techniques in engineering

Gaurav Gupta, SC21B026

1 Overview

This report discusses the implementation and testing of two optimization algorithms: **BFGS (Broyden–Fletcher–Goldfarb–Shanno)** and **Genetic Algorithm (GA)** in python. Both algorithms serve different optimization needs, with BFGS being suitable for smooth, differentiable functions and GA being more flexible for complex, non-linear, and non-differentiable problems.

BFGS Algorithm

BFGS is a quasi-Newton method used to find local minima of smooth functions. It approximates the Hessian matrix to iteratively update the search direction, making it efficient for problems where derivatives are available and relatively inexpensive to compute. The BFGS method is particularly well-suited for smooth, unimodal functions and is widely used in various scientific and engineering applications due to its convergence properties and computational efficiency.

```python
import numpy as np
import sympy as sp

def derv(func, variables):

    nVariables = len(variables)

    # Empty array for gradient functions
    df = np.empty(nVariables, dtype=object)

    # Compute the gradient with respect to each input
    ↪  variable
```

```python
    for i in range(nVariables):
        df[i] = sp.diff(func, variables[i])  # Compute gradient w.r.t. i-th variable

    return df

def dervAtPoint(der, var, values):
    val = dict(zip(var,values))
    res = []
    for i in range(len(val)):
        res.append(der[i].subs(val).evalf())

    return res

def backtracking_line_search(fun, x, grad, direction, alpha=1.0, rho=0.8, c=1e-4):
    """
    Perform backtracking line search to find the optimal step size.

    Parameters:
    - fun: objective function
    - x: current point as a numpy array
    - grad: gradient at the current point
    - direction: search direction
    - alpha: initial step size (default is 1)
    - rho: factor to decrease alpha (0 < rho < 1), typically 0.8
    - c: Armijo condition constant, typically 1e-4

    Returns:
    - optimal alpha satisfying the Armijo condition
    """
    fx = fun(x)
    while fun(x + alpha * direction) > fx + c * alpha * np.dot(grad, direction):
        alpha *= rho  # Reduce step size by a factor of rho
    return alpha

def cauchyMethod(fun, x0):
    '''
```

```
    This function implements the cauchy's method.

    1. Input: Function, X0
    2. Compute the number of variables and create symbolic variables
    3. Create a symbolic function
    4. Find the gradient of the function
    5. Determine the step length
    6. Return the value of X1
    '''

    # Find the number of variables
    nVar = len(x0)

    # create an array of symbolic variables
    var = sp.symbols(f'x0:{nVar}')

    # create a symbolic function
    func = fun(var)

    # finding the gradient of the func
    der = derv(func, var)

    derX0 = dervAtPoint(der,var,x0)

    if derX0 == [0]*nVar:
        return x0

    dir = -np.transpose(derX0)
    step =  backtracking_line_search(fun,np.array(x0, dtype=float),derX0, dir)
    x1 = np.transpose(x0) + step*dir
    return x1

def matrixUpdate(B0, g, d):

    # Ensure g and d are column vectors
    g = g.reshape(-1, 1)
    d = d.reshape(-1, 1)

    # Calculate denominator
```

```
    den = np.dot(np.transpose(d), g)[0, 0]

    # Update the matrix using the BFGS formula
    t1 = (1 + np.dot(np.transpose(g), B0) @ g / den) * (d @ np.transpose(d) / den)
    t2 = B0 @ g @ np.transpose(d) / den
    t3 = d @ np.transpose(g) @ B0 / den

    # Update B0
    B1 = B0 + t1 - t2 - t3

    return B1


def check(dx1, eps, x0, x1):
    a1 = np.all(np.abs(dx1) < eps)
    e = 1e-8
    a2 = np.all(np.abs(x1-x0) < e)
    return not (a1 or a2)

def BFGS(fun, x0, eps=1e-15):
    '''
    This function implements the BFGS.

    Input: Function, X0
    Interation loop until gradient = 0

    Intiate the loop using x1, df/dx at x1 from Cauchy's solution

    Within loop:
        1. compute g and d
        2. update B matrix
        3. update the value of x1 and x0 = x1
        4. update the value of df/dx at x1 and x0
    '''
    # Find the number of variables
    nVar = len(x0)

    # create an array of symbolic variables
    var = sp.symbols(f'x0:{nVar}')
```

```

    # create a symbolic function
    func = fun(var)

    # finding the gradient of the func
    der = derv(func, var)

    #find the first solution using cauchy's method
    x1 = cauchyMethod(fun, x0)

    #find the gradients at x0 and x1
    dx0 = dervAtPoint(der,var,x0)
    dx1 = dervAtPoint(der,var,x1)

    B0 = np.array([[1, 0], [0, 1]])

    while check(dx1,eps,x0,x1):

        # find g and d vectors
        g = np.transpose(dx1) - np.transpose(dx0)
        d = np.transpose(x1) - np.transpose(x0)

        # update B matrix
        B1 = matrixUpdate(B0,g,d)

        # compute optimal step length
        dir = -B1@np.transpose(dx1)
        step =  backtracking_line_search(fun, np.array(x1, dtype=float), dx1, dir)

        x0 = x1
        x1 = np.transpose(x1) + step*dir
        dx0 = dx1
        dx1 = dervAtPoint(der,var,x1)
        B0 = B1

    return np.round(np.array(x1,dtype='float'), 5)

```

Genetic Algorithm (GA)

GA is an evolutionary algorithm inspired by the principles of natural selection and genetics. This implementation includes:

- **Elitism-based selection**: Ensures the fittest individuals are carried over to the next generation.
- **Simulated Binary Crossover (SBX)**: Combines pairs of parents to produce offspring with a controlled level of diversity.
- **Normally Disturbed Mutation**: Introduces small variations in offspring to enhance exploration of the search space.

GA is particularly effective for global optimization, where the objective function may be non-linear, multi-modal, or non-differentiable.

```python
import numpy as np
import sympy as sp
import copy

class solution:

    # Parameters
    DNA = [0]   # DNA of the solution
    fitness = 0 # How good the solution is ?
    var = [] # Variables

    # Methods
    def eval_fitness(self, func):
        fun = func(self.var) # create a symbolic function
        vals = dict(zip(self.var, self.DNA)) # dictonary of
        ↪   var and values
        self.fitness = fun.subs(vals).evalf()

    def __init__(self, nDim, bounds, func):
        self.DNA = np.random.uniform(bounds[0], bounds[1],
        ↪   nDim)
        self.var = sp.symbols(f'x0:{nDim}')
        self.eval_fitness(func)

    def copy(self):
        new_sol = solution.__new__(solution)
```

```python

        # Manually copy attributes
        new_sol.DNA = copy.deepcopy(self.DNA)
        new_sol.fitness = self.fitness
        new_sol.var = self.var
        return new_sol

    def show(self):
        print(f"DNA: {self.DNA}")
        print(f"Fitness: {self.fitness}")


class generation:

    #Parameters
    id = 0 # id^th generation
    nsol = 0 # Number of solutions in the generation
    population = [] # Array of solution objects
    parents = [] # Array of parents in current generation
    children = [] # Array of children of current generation
    fun = 0 # Objective Function

    #Methods

    def __init__(self, gen, members, func, ndim, bounds):

        if(members<0):
            raise ValueError("Number of members should be positive.")
        elif(members%2!=0):
            raise ValueError("Number of members in a generation should be even")

        self.nsol = members
        self.id = gen
        self.population = [solution(ndim,bounds,func) for i in range(self.nsol)]
        self.fun = func

    def evaluate(self):
        for child in self.children:
```

```python
            child.eval_fitness(self.fun)

    def elitism(self):

        '''
        Function for Elitism based selection of parents

        Input: Self

        Output: Array of solution selected from the population

        Algorithm: Constant population size with Elitism

            1. Create a sorted list of elements based on fitness (minimum is the best)
            2. The top 10% are members of next generation directly.
            3. Rest 90% are parents for next generation.
        '''

        sortedPopulation = sorted(self.population, key=lambda obj: obj.fitness)

        elite_count = int(0.1 * self.nsol)
        self.children = sortedPopulation[:elite_count]
        self.parents = sortedPopulation[elite_count:]

        pass

    def crossover(self):

        '''
        Function to perform Simulated Binary Crossover for a given array of parents.

        Input: Array of parents in current generation

        Output: Array of children of current generation

        Algorithm:
```

```
        1. Select two parents x1 and x2
        2. Find U and Calculate Beta,
            Beta = (2U)^1/(etac + 1) if U<0.5
            Beta = (1/(2U-1))^1/(etac + 1) if U>0.5
        3. x1_new = 0.5*((1+Beta)*x1 + (1-Beta)*x2)
           x2_new = 0.5*((1-Beta)*x1 + (1+Beta)*x2)

           eta = 5
        '''
        np.random.shuffle(self.parents)
        for i in range(0,len(self.parents),2):
            p1 = self.parents[i]
            p2 = self.parents[i+1]
            U = np.random.rand(1)
            eta = 5
            if (U<0.5):
                Beta = (2*U)**(1/(eta+1))
            else:
                Beta = (1/(2*U-1))**(1/(eta+1))

            # Make a copy of parents
            c1 = p1.copy()
            c2 = p2.copy()

            #Update the DNA from crossover
            c1.DNA = 0.5*(1+Beta)*p1.DNA + 0.5*(1-Beta)*p2.DNA
            c2.DNA = 0.5*(1-Beta)*p1.DNA + 0.5*(1+Beta)*p2.DNA

            #Update the fitness value
            c1.eval_fitness(self.fun)
            c2.eval_fitness(self.fun)

            #Append to the children array
            self.children.append(c1)
            self.children.append(c2)

    def mutation(self, mutationRate = 0.05, sigma = 0.5):

        '''
        Function to perform mutation in children of current generation
```

```

        Input: Array of children for current generation

        Output: Array of children with mutation

        Algorithm: Normally Disturbed mutation

        if prob < MutationRate:
            x_new = x + N(0, sigma)
        '''
        for sol in self.children:
            prob = np.random.rand()
            if prob <= mutationRate:
                sol.DNA += np.random.normal(0, sigma,
                ↪  size=len(sol.DNA))

    def show(self):
        print(f"Generation Number: {self.id}")
        print(f"Population: {[i.DNA for i in
        ↪  self.population]}")
        print(f"Selected Parents: {[i.DNA for i in
        ↪  self.parents]}")
        print(f"Children: {[i.DNA for i in self.children]}")

    def reset(self):
        self.population = []
        self.parents = []
        self.children = []

def GA(nGen, nSol, func, ndim=2, bounds=[-10,10]):
    '''
    Function to implement Genetic Algorithm

    Input:
    1. Number of Generations
    2. Number of Solutions in each generation
    3. Objective Function
    4. Dimension of solution
    5. Bound of the solution

    Output: Optimal Solution
```

```

    Algorithm:
    1. Initiate first generation
    Inside loop:
        1. Elitism() of nth gen
        2. Crossover() of nth gen
        3. Mutation() of nth gen
        4. population of n+1th gen = children of nth gen
        5. Check most optimal solution of nth generation
        ↪ similar to most optimal solution of n+1th gen, if
        ↪ yes stop the algorithm.
        6. If not continue till all generations and return
        ↪ the most optimal solution.

    '''
    eps = 1e-2
    gen0 = generation(0, nSol, func, ndim, bounds)

    bestSol = 0
    for i in range(1, nGen):
        gen0.elitism()
        gen0.crossover()
        gen0.mutation(mutationRate=0.075, sigma=0.2)
        gen0.evaluate()

        gen1 = generation(i, nSol, func, ndim, bounds)
        gen1.reset()
        gen1.population = copy.deepcopy(gen0.children)

        gen1.elitism()
        gen1.crossover()
        gen1.mutation(mutationRate=0.1, sigma=0.1)
        gen1.evaluate()

        bestgen1 = sorted(gen1.children, key=lambda obj:
        ↪ obj.fitness)[0]

        if i==1:
            bestSol = bestgen1
        elif bestgen1.fitness < bestSol.fitness:
            bestSol = bestgen1
```

```
        # print(f"Generation {i}, Best Fitness:
        ↪  {bestgen1.fitness}")

        gen0.reset()
        gen0.population = copy.deepcopy(gen1.children)

    return bestSol
```

2 Test Functions and Results

Two benchmark functions were used to evaluate the performance of BFGS and GA:

- **Bohachevsky Function**:

  $$f(x,y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7$$

  This unimodal function tests the algorithms' ability to converge to a global minimum in a smooth landscape.

- **Ackley Function**:

  $$f(x,y) = -20\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right) - \exp\left(0.5(\cos(2\pi x) + \cos(2\pi y))\right) + 20 + \exp(1)$$

  Known for its numerous local minima, the Ackley function challenges the algorithms with a rugged, multimodal landscape.

```
from BFGS import BFGS
from GA import GA
import numpy as np
import sympy as sp

# Sample Test Problem for BFGS
def testFunc(arr):
    return arr[0] - arr[1] + 2*arr[0]**2 + 2*arr[0]*arr[1] +
    ↪  arr[1]**2

# Unimodal Benchmark problem
def unimodalBenchmark(arr):
    # Bohachevsky Unimodal function
```

```
    # f(x,y) = x^2 + 2*y^2 - 0.3*cos(3*pi*x) - 0.4*cos(4*pi*y) + 0.7
    # Minimum at (0,0)
    return arr[0]**2 + 2*(arr[1]**2) - 0.3*sp.cos(3*sp.pi*arr[0]) - 0.4*sp.cos(4*sp.pi*arr[1]) + 0.7

# Multimodal Benchmark problem
def multimodalBenchmark(arr):
    # Ackley Function
    # f(x,y) = -20exp(-0.2*sqrt(0.5*(x^2 + y^2))) - exp(0.5*(cos(2pix) + cos(2piy))) + 20 + exp(1)
    # Global Minimum at (0,0)
    return -20*sp.exp(-0.2*sp.sqrt(0.5*(arr[0]**2 + arr[1]**2))) - sp.exp(0.5*(sp.cos(2*sp.pi*arr[0]) + sp.cos(2*sp.pi*arr[1]))) + 20 + sp.exp(1)

# #BFGS Test
x0 = [0, 0]
sol1 = BFGS(testFunc,x0)
assert(np.array_equal(sol1, np.array([-1, 1.5])))

# Test for the unimodal function with global minima at [0,0]
x0 = [-5,5]
sol2 = BFGS(unimodalBenchmark,x0)
assert(np.array_equal(sol2, np.array([0,0])))

# Test for the multimodal function with global minima at [0,0]
x0 = [-5,5]
sol3 = BFGS(multimodalBenchmark,x0)
assert(not np.array_equal(sol3, np.array([0,0]))) # Converges to a local maxima

x0 = [-0.1,0.1]
sol4 = BFGS(multimodalBenchmark,x0)
assert(np.array_equal(sol4, np.array([0,0]))) # Converges to the global maxima

# GA Test
solGA1 = GA(60, 100, testFunc)
```

```python
print(f'Solution from BFGS: {sol1} and Solution from GA: {solGA1.DNA}')

# Test for the unimodal function with global minima at [0,0]
solGA2 = GA(60, 100, unimodalBenchmark)
print(f'Solution from BFGS: {sol2} and Solution from GA: {solGA2.DNA}')

# Test for the multimodal function with global minima at [0,0]
solGA4 = GA(60, 100, multimodalBenchmark)
print(f'Solution from BFGS: {sol4} and Solution from GA: {solGA4.DNA}')
```

3 Results from the Code

The following are sample outputs from running the BFGS and GA algorithms on the test functions:

```
Solution from BFGS: [-1.   1.5] and Solution from GA: [-0.99964371  1.65817322]
Solution from BFGS: [-0.  0.] and Solution from GA: [-0.06193411  0.02544361]
Solution from BFGS: [ 0. -0.] and Solution from GA: [0.01887624 0.95234389]
```

(Results may vary with different parameters)

4 Code Availability

The code for the BFGS and Genetic Algorithm implementations, including tests on the Bohachevsky and Ackley functions, is available on GitHub at: `https://github.com/airwarriorg91/Optimization-Techniques/tree/main/Assignment-2`

5 Conclusion

The BFGS algorithm is effective for smooth, unimodal functions like the Bohachevsky function, achieving convergence with relatively few iterations. The Genetic Algorithm, with elitism-based selection, SBX, and normally disturbed mutation, provides robust performance across both unimodal and multimodal functions, particularly with the challenging Ackley function. These results demonstrate the complementary strengths of BFGS and GA in solving diverse optimization problems.