Steepest Descent Optimization

Optimization Techniques

CS348

Ragheed Almasry - 412117664
Anas Alarbied - 411116054
Mustafa Sallat - 412117552
Instructor: Dr.Ali Mustafa

May 12, 2024

Contents

1 Introduction

Steepest descent optimization is a versatile technique used to find the minimum of a function. By iteratively adjusting the parameters of the function, steepest descent aims to converge to the optimal point where the function reaches its minimum value. In this report, we explore the implementation and analysis of steepest descent optimization, focusing on dynamically computing the optimal point by defining only the objective function and the initial point.

2 Objective

The objective of this report is to demonstrate the effectiveness of steepest descent optimization in dynamically computing optimal points without the need for manual intervention. We aim to showcase the simplicity and versatility of the approach by allowing users to define the objective function and the initial point, enabling seamless optimization across various scenarios and problem domains.

3 Implementation

We implement steepest descent optimization using Python and the Autograd library. The implementation involves several key steps:

3.1 Defining The Objective Function and The Initial Point

```
def f(x):
    return (2*x[0] - 4*x[1]) ** 2 + (x[1] + 6) ** 2 - 1.75 * (x[0] - 3*x[1]) * (x[1] + 8)
```

Figure 1: Objective Function *f(x)*

```
# Initial point
initial_point = np.array([4.0, -7.0], dtype=float)
```

Figure 2: Initial Point

3.2 Computing The Gradient and Hessian Matrix Using Autograd

```
# Compute the gradient of the objective function
gradient_f = grad(f)
```

Figure 3: Gradient of The Objective Function $\nabla f(x)$

```
def steepest_descent(initial_point):
    gradient = gradient_f(initial_point)
    hessian_func = hessian(f)
    hessian_matrix = hessian_func(initial_point)
```

Figure 4: Gradient of Initial Point $\nabla f(x^0)$ *and The Hessian*

3.3 Calculating c and k for the standard form of the quadratic function

```
# Calculate c and k
c = gradient_f(np.zeros_like(initial_point))
k = f(np.zeros_like(initial_point))
```

Figure 5: Calculating The Linear Part and The Constant

3.4 Computing The Gradient Length, Learning Rate

```
# Calculate the length of the gradient
gradient_length = np.linalg.norm(gradient)
```

Figure 6: Gradient Length

```
def calculate_learning_rate(grad, hessian):
    learning_rate = np.dot(grad, grad) / np.dot(grad, np.dot(hessian, grad))
    return learning_rate
```

Figure 7: Learning Rate Function

```
learning_rate = calculate_learning_rate(gradient, hessian_matrix)
```

Figure 8: Learning Rate

3.5 Checking Function Convexity

```
# Check function convexity
is_convex = np.all(np.linalg.eigvals(hessian_matrix) >= 0)
```

Figure 9: Check Function Convexity

3.6 Finding Critical Point Using The Hessian Matrix.

```
# critical point using the Hessian Matrix
critical_point = -np.linalg.inv(hessian).dot(c)
```

Figure 10: Critical Point Using The Hessian

3.7 Computing The optimal Points Using Steepest Descent.

```
# Steepest Descent Optimization Method
x_next = initial_point - learning_rate * gradient
```

Figure 11: Enter Caption

This comprehensive approach enables us to efficiently optimize the objective function without manual intervention, making it a powerful tool for various optimization tasks.

4 Source Code

```python
import autograd.numpy as np
from autograd import grad, hessian

def f(x):
    return (2*x[0] - 4*x[1]) ** 2 + (x[1] + 6) ** 2 - 1.75 * (x[0] - 3*x[1]) * (x[1] + 8)

# compute the gradient of the objective function
gradient_f = grad(f)

def calculate_learning_rate(grad, hessian):
    learning_rate = np.dot(grad, grad) / np.dot(grad, np.dot(hessian, grad))
    return learning_rate

def steepest_descent(initial_point):
    gradient = gradient_f(initial_point)
    hessian_func = hessian(f)
    hessian_matrix = hessian_func(initial_point)

    learning_rate = calculate_learning_rate(gradient, hessian_matrix)

    # Steepest Descent Optimization Method
    x_next = initial_point - learning_rate * gradient

    # Check function convexity
    is_convex = np.all(np.linalg.eigvals(hessian_matrix) >= 0)

    return x_next, f(x_next), learning_rate, gradient, hessian_matrix, is_convex

# Initial point
initial_point = np.array([4.0, -7.0], dtype=float)

# Perform steepest descent optimization
optimal_point, min_value, learning_rate, gradient, hessian, is_convex = steepest_descent(initial_point)

# Calculate c and k
c = gradient_f(np.zeros_like(initial_point))
k = f(np.zeros_like(initial_point))

# Calculate the length of the gradient
gradient_length = np.linalg.norm(gradient)

# critical point using the Hessian Matrix
critical_point = -np.linalg.inv(hessian).dot(c)

print("Constant k:", k)
print("Linear part c:", c)
print("Hessian matrix:")
print(hessian)
if is_convex:
    print("The function is convex.")
else:
    print("The function is not convex.")
print("Critical point using the Hessian:", critical_point)
print("Gradient at initial point:", gradient)
print("Gradient length:", gradient_length)
print("Learning rate:", learning_rate)
print("Optimal point using Steepest Descent:", optimal_point)
print("Minimum value:", min_value)
```


Figure 12: Source Code

5 Examples

5.1 $f(x) = (x_1 + 1)^2 + (x_2 + 6)^2 - 1.75(x_1 - 2)(x_2 + 4)$

Defining The Objective Function and The Initial Point:

```
def f(x):
    return (x[0] + 1) ** 2 + (x[1] + 6) ** 2 - 1.75 * (x[0] - 2) * (x[1] + 4)
```

Figure 13: Objective Function

```
# Initial point
initial_point = np.array([5, 7], dtype=float)
```

Figure 14: Initial Point

Results

```
Constant k: 51.0
Linear part c: [-5.  15.5]
Hessian matrix:
[[ 2.   -1.75]
 [-1.75  2.  ]]
The function is convex.
Critical point using the Hessian: [-18.26666667 -23.73333333]
Gradient at initial point: [-7.25 20.75]
Gradient length: 21.980104640333266
Learning rate: 0.32364085494776945
Optimal point using Steepest Descent: [7.3463962  0.28445226]
Minimum value: 69.07050597667944
```

codesnap.dev

Figure 15: Enter Caption

5.2 $f(x) = (2x_1 - 4x_2)^2 + (x_2 + 6)^2 - 1.75(x_1 - 3x_2)(x_2 + 8)$

Defining The Objective Function and The Initial Point:

```python
def f(x):
    return (2*x[0] - 4*x[1]) ** 2 + (x[1] + 6) ** 2 - 1.75 * (x[0] - 3*x[1]) * (x[1] + 8)
```

Figure 16: Objective Function

```python
# Initial point
initial_point = np.array([4.0, -7.0], dtype=float)
```

Figure 17: Initial Point

Results:

```
Constant k: 36.0
Linear part c: [-14.  54.]
Hessian matrix:
[[  8.   -17.75]
 [-17.75  44.5 ]]
The function is convex.
Critical point using Hessian: [-8.19541985 -4.48244275]
Gradient at initial point: [ 142.25 -328.5 ]
Gradient length: 357.97669267705123
Learning rate: 0.01934924201406845
Optimal point using Steepest descent: [ 1.24757032 -0.643774  ]
Minimum value: 13.473318492520413
```

codesnap.dev

Figure 18: Results

6 Conclusion

We conduct experiments with different objective functions and initial points to evaluate the performance of the dynamically computed optimal points. The results demonstrate the versatility and effectiveness of steepest descent optimization in dynamically computing optimal points across various problem domains, providing valuable insights for practical applications in optimization tasks.