Properties of the Sign Gradient Descent Algorithms

Introduction

Gradient-based optimization techniques are the backbone of modern machine learning and optimization problems. While Gradient Descent (GD) uses full gradient information, its variants such as Sign Gradient Descent (SGD) rely only on the sign of gradients, making them computationally efficient and robust to noise. We study the properties and behaviors of SGD, Adaptive Sign Gradient Descent (ASGD), and Hybrid Gradient Descent (HGD) in challenging optimization scenarios, focusing on convergence, stability, and precision.

Optimization Techniques

Gradient Descent (GD)

GD is the simplest optimization method, updating parameters using the gradient:

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

While effective for smooth functions, GD struggles in noisy and non-convex settings due to oscillations.

code:

```
#x_init is the inital point, gamma is the learning rate
def gradient_descent(grad_func, x_init, gamma, iterations):
    x = np.array(x_init, dtype=float)
    #Convert x_init into a NumPy array
    path = [x.copy()]
#path is a list to store the sequence of points visited during
#the optimization process.
   for _ in range(iterations):
        grad = grad_func(x)
        x -= gamma * grad #Update x by moving in the direction
        #opposite to the gradient scaled by gamma for total iterations
        path.append(x.copy())

    return np.array(path) #returning the path array for further references
```

Sign Gradient Descent (SGD)

SGD updates parameters using the sign of the gradient:

$$x_{t+1} = x_t - \eta \operatorname{sign}(\nabla f(x_t))$$

- The sign function is defined as:

$$\operatorname{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{1}$$

code:

```
#x_init is the inital point, gamma is the slearning rate
def sign_gradient_descent(grad_func, x_init, gamma, iterations):
    x = np.array(x_init, dtype=float)
    path = [x.copy()]
    for _ in range(iterations):
        grad = grad_func(x)
        x -= gamma * np.sign(grad)#Update x by moving in the direction
        #opposite to the gradient scaled by gamma for total iterations
        #but the dign of the grad is determined by the above
        #sign function
        path.append(x.copy())

    return np.array(path) #returning the path array for further references
```

Adaptive Sign Gradient Descent (ASGD)

ASGD adjusts step sizes dynamically based on the optimization trajectory:

$$x_{t+1} = x_t - \eta_t \operatorname{sign}(\nabla f(x_t))$$

where η_t is updated adaptively. This balances convergence speed and stability in non-convex landscapes.

code:

```
#x_init is the inital point, gamma is the slearning rate
def adaptive_sign_gradient_descent(grad_func, x_init, gamma_0, decay_rate,
    x = np.array(x_init, dtype=float)
    gamma = gamma_0 #initial learning rate
    path = [x.copy()]
    for _ in range(iterations):
        grad = grad_func(x)
        x -= gamma * np.sign(grad)#choose the next point as that in SGD
```

```
        path.append(x.copy())
        gamma *= decay_rate  # Reduce step size
        #The decay_rate is a factor between 0 and 1 .
        #This gradually decreases the step size, making updates
        #smaller as the algorithm progresses(to refine the solution).
    return np.array(path)
```

Hybrid Gradient Descent (HGD)

HGD combines the strengths of GD and SGD by dynamically switching between the two based on optimization stages. This hybrid approach allows for efficient exploration of the function landscape and precise convergence near minima.

code:

```
#x_init is the inital point, gamma is the slearning rate
def hybrid_gradient_descent(grad_func, x_init, gamma1, gamma2, iterations):
    x = np.array(x_init, dtype=float)
    path = [x.copy()]
    for _ in range(iterations):
        grad = grad_func(x)
        x -= gamma1 * grad + gamma2 * np.sign(grad)
        #The position x is updated using a combination of: GD and SGD
        #gamma1: Scales the standard gradient descent component
        #gamma2: Scales the sign gradient descent component.
        #This combination can balance smooth convergence and robustness
        path.append(x.copy())
    return np.array(path)
```

Benchmark Functions

To analyze the properties of these algorithms, experiments were conducted on the following functions:

- **Trimodal Function:** $f(x) = \sin(x) + \sin(3x) + \sin(5x) + 3$
- **Quadratic Function:** $f(x) = x^2$
- **Cubic Function:** $f(x) = x^3$
- **Quartic Function:** $f(x) = x^4$

Each function presents unique challenges, including multiple local minima, sharp gradients, and varying smoothness.

Experimental Results

The behavior of the algorithms was visualized on the benchmark functions. Figures 1 and 2 illustrate optimization paths on these functions.

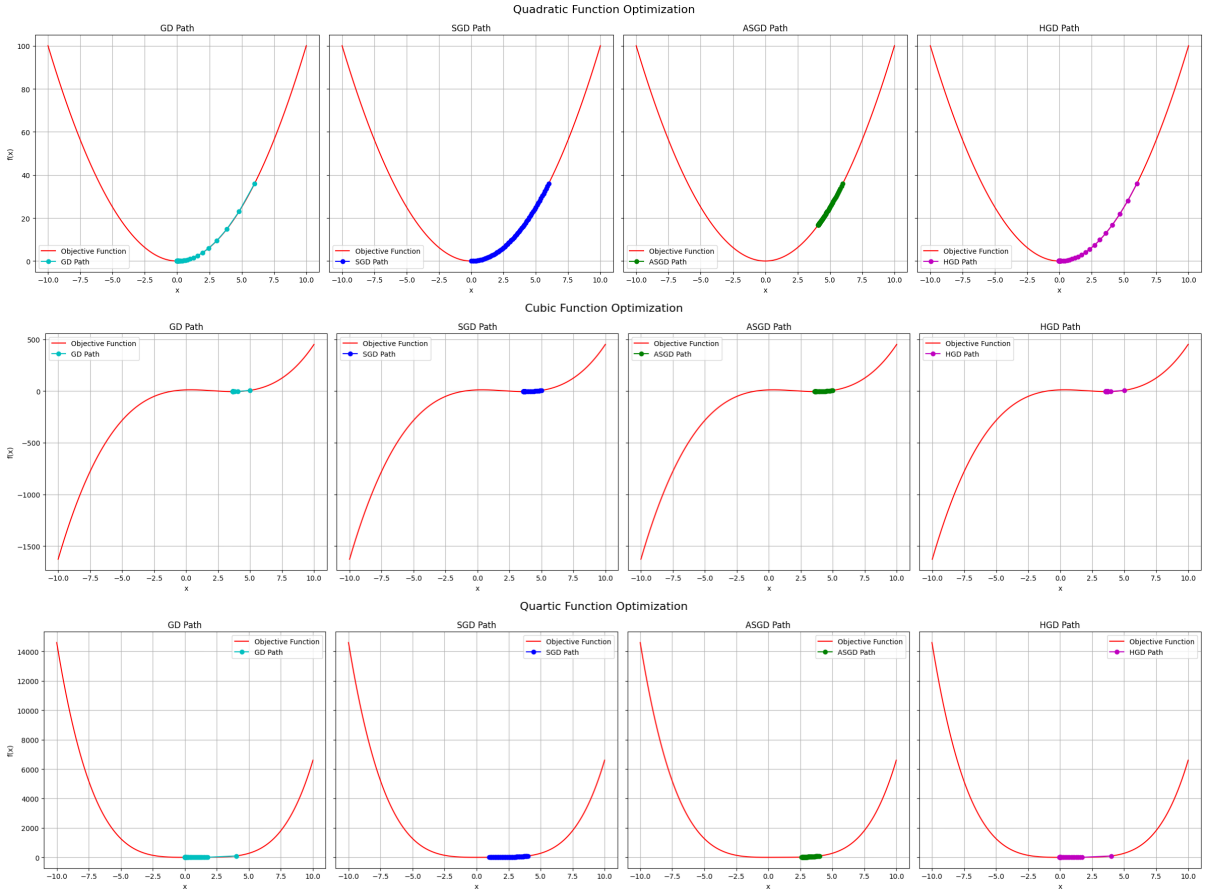
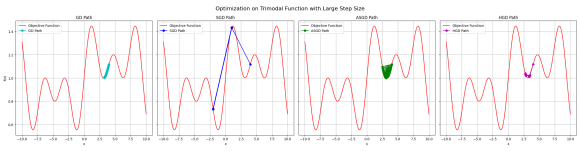
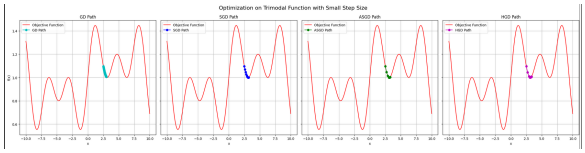


Figure 1: Overview of benchmark functions: Trimodal, Quadratic, Cubic, and Quartic.



(a) Optimization on the Trimodal function (large learning rate).



(b) Optimization on the Trimodal function (small learning rate).

Figure 2: Optimization paths on the Trimodal function.

Discussion

The analysis reveals several key properties:

- **GD:** Performs well on smooth, convex functions but struggles with oscillations in non-convex scenarios.

- **SGD:** Reduces oscillations and noise sensitivity but lacks precision near minima due to fixed step size.
- **ASGD:** Offers a balance of speed and precision by adapting the step size based on the optimization landscape.
- **HGD:** Effectively combines the strengths of GD and SGD, achieving robust performance across diverse function types.

Conclusion

The study highlights the strengths and weaknesses of different gradient descent algorithms. SGD and its variants demonstrate robustness to noise and non-convexity, while HGD provides a hybrid solution for efficient and precise optimization.

References

- Properties of the Sign Gradient Descent Algorithms, Emmanuel Moulay, Vincent Léchappé, Franck Plestan