

Comparative Analysis of Quasi-Newton and Polak-Ribière Methods in Optimization Problems

Yufeng Liu (yufengl2), Ziyang Xie (ziyang8), Jiangwei Yu (jy79)

1. ABSTRACT

This study explores the efficiency of the Quasi-Newton method, BFGS, and the Conjugate Gradient method using Polak-Ribière in solving optimization problems of varying complexity and size. The investigation focuses on two fundamental scenarios: a dual form of ridge regression and a multilayer perceptron for complex function fitting. Our results show that BFGS works better in problems involving high order derivatives or full-rank hessians, while CG works better in quadratic problems and overdetermined systems with rank-deficient hessians.

Our Code is available here: <https://github.com/ZiYang-xie/CS544-24Spring/tree/main/mp1>

2. INTRODUCTION

Optimization algorithms are pivotal in computational mathematics and data science, with applications spanning various domains. This report addresses a critical question: when is a Quasi-Newton method, implemented using conjugate gradients, preferable over the Polak-Ribière method, or vice versa? Traditional wisdom suggests starting with Quasi-Newton methods, but this study seeks a more experimental validation. We assess the performance of these methods in two distinct scenarios - small-scale (10s-100s of variables) and large-scale problems (1000s of variables) - and consider the impact of large third derivatives and the effectiveness of the Polak-Ribière restart strategy.

3. BACKGROUND

In the gradient direction calculation, CG and BFGS incorporate second order information in different ways. In short, CG approximates the effect of the hessian by incorporating the difference between the current gradient and the previous gradient. BFGS on the other hand maintains a representation of a reconstructed hessian. Thus CG is commonly regarded as a first order method, and BFGS as a second order method.

4. PROJECT DESCRIPTION

In the next two sections we will discuss our experiments on 1. Dual Ridge Regression, and 2. Multilayer Perceptron Function Fitting. The selection of these two tasks is strategic, aiming to evaluate these optimization algorithms across a spectrum of problem complexities.

Dual Ridge Regression provides a well-defined, quadratic, and convex optimization problem, enabling a clear comparison of the algorithms in terms of efficiency, accuracy, and scalability, with the advantage of having a known closed-form solution for benchmarking.

Multilayer Perceptron Function Fitting introduces a non-linear and complex optimization challenge, allowing us to assess algorithm performance in higher-order derivatives and non-convex optimization.

This setup allows for a comprehensive assessment of CG and BFGS, highlighting their strengths and weaknesses in handling problems with varying degrees of complexity, from quadratic and convex to non-linear and complex optimization landscapes.

4.1 Dual Ridge Regression

Motivation

There are several good properties of dual ridge regression. First, its primal form can be solved in a closed-form way. Second, its object function is quadratic by construction, and has a positive semi-definite hessian. Third, we can manipulate the number of variables to arbitrary amounts while keeping interpretability. Interpretation is done by transforming the dual back to the primal in low dimension and applying direct visualization.

Derivation

Ridge Regression primal problem:

$$\min_w \frac{1}{2} \|Xw - y\|^2 + \frac{\lambda}{2} \|w\|^2$$

We impose an extra constraint: $Xw - y = 0$ so that we can write down its lagrangian and form a dual problem:

$$L(w, a) = \frac{1}{2} \|Xw - y\|^2 + \frac{\lambda}{2} \|w\|^2 - a^T (Xw - y)$$

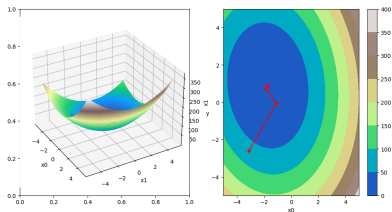
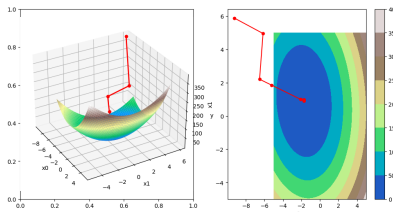
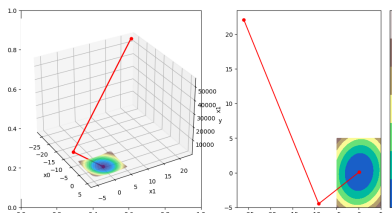
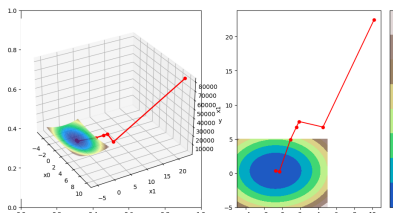
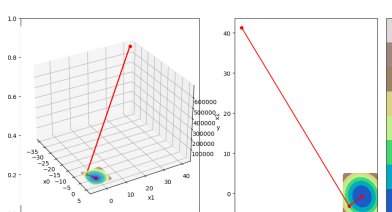
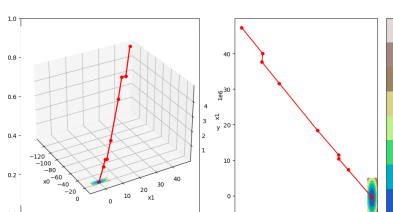
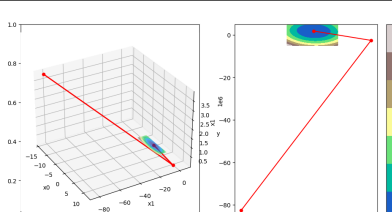
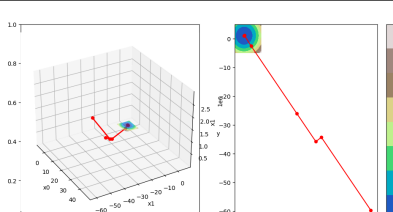
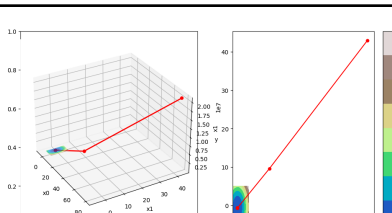
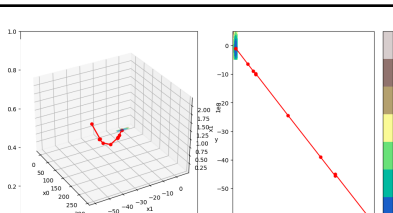
$$\max_a -\frac{1}{2} a^T K (K + \lambda I)^{-1} a + a^T Ky, \quad \text{where } K = X^T X$$

Procedure

1. Generate an arbitrary N number of low dimensional input data ($N \gg d$)
2. Solve for the high dimensional (N) dual variables
3. Get back the primal variables in dimension d
4. Plot the primal function and primal variables in low dimension.

Experiments

The following table shows the results with 2d input data ($N \gg d$). The surface is the primal function. The path is the transformed primal variables (from solving the dual problem)

2d input data time in seconds	CG	BFGS
N=10	 <p>restart=0, iteration=3, time=0</p>	 <p>restart=0, iteration=3, time=0</p>
N=100	 <p>restart=0, iteration=2, time=0</p>	 <p>restart=1, iteration=6, time=0</p>
N=500	 <p>restart=0, iteration=2, time=0</p>	 <p>restart=2, iteration=10, time=0.1s</p>
N=1000	 <p>restart=0, iteration=2, time=0</p>	 <p>restart=1, iteration=6, time=0.4s</p>
N=5000	 <p>restart=0, iteration=3, time=1.2</p>	 <p>restart=2, iteration=10, time=50</p>

The following table shows the results with high dimension input data ($N \sim d$), we random add

	CG	BFGS
N=5, d=5	restart=0, iteration=8	restart=0, iteration=5
N=10, d=10	restart=3, iteration=161	restart=0, iteration=15
N=50, d=50	restart=19, iteration=4247, time=0.4	restart=0, iteration=60, time=0
N=100, d=100	restart=50, iteration=20462, time=2.6	restart=0, iteration=108, time=0
N=500, d=500	Timeout	restart=0, iteration=503, time=5.9
N=500, d=450	restart=16, iteration=16666, time=21.6	restart=0, iteration=456, time=5.6
N=1000, d=900	restart=7, iteration=15610, time=1m15s	restart=0, iteration=908, time=50s

We also compare the two methods' memory usage:

	CG	BFGS	L-BFGS
N=2000, d=5	~0.1 MB	64.9 MB	1.0 MB
N=2000, d=50	~0.1 MB	94.1 MB	0.6 MB
N=2000, d=500	~0.1 MB	124.0 MB	1.0 MB
N=10000, d=50	1.1 MB	time out	3.2 MB
N=10000, d=500	1.1 MB	time out	2.1 MB

Analysis

Our results show that both methods converge in a relatively smooth path, without zig-zag behaviors. This is as expected because both methods incorporate second order information. However, CG is significantly more efficient in cases of $N \gg d$, while BFGS more efficient in cases of $N \sim d$. We reason that in cases $N \gg d$, the hessian ($X^T X$) is highly rank deficient ($\text{rank} \leq d$). In theory CG should still take N steps, but it benefits from generating independent directions in each step, thus avoiding the trivial dimensions. BFGS suffers from a higher overhead. On the other hand, in cases of $N \sim d$, the hessian is almost full rank. BFGS can better capture the second order behavior than CG, thus converging faster. Another way to think about it is that a highly rank-deficient objective function is dominated by its first-order terms, thus a second order approach (BFGS) is not very helpful.

For memory usage, cg consumes the least amount of memory, followed by L-BFGS. BFGS has to keep

track of steps. Thus its memory usage depends on the number of iterations, which directly depends on the objective function complexity.

4.2 Multilayer Perceptron Function Fitting

Motivation

This problem is suitable to experiment with objective functions with large 3rd derivatives.

Implementation

We experiment MLP with different layers, which indicates different optimization difficulty. Adding more layers to an MLP increases the optimization problem's complexity due to the higher-dimensional parameter space and the potential for more complex error landscapes, including deeper and more numerous local minima.

It's important to note that despite utilizing PyTorch for implementing our Multilayer Perceptron (MLP), we opted not to employ Stochastic Gradient Descent (SGD) for optimizing the MLP parameters.

Instead, we utilized the Conjugate Gradient (CG) and Quasi-Newton methods to directly optimize MLP's parameters, leveraging PyTorch's `torch.autograd` to compute their gradient.

```
class MLP_fitter:
    def __init__(self, X, Y, input_dim, hidden_dim, output_dim=1, layer_num=3):
        self.X = torch.tensor(X, dtype=torch.float32)
        self.Y = torch.tensor(Y, dtype=torch.float32)
        self.model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            *[
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU()
            ]*(layer_num-1),
            nn.Linear(hidden_dim, output_dim)
        )
```

Experiments

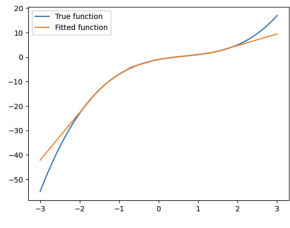
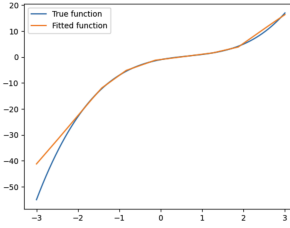
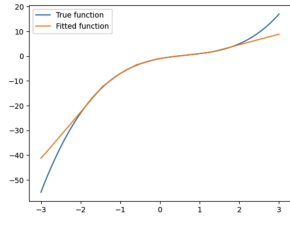
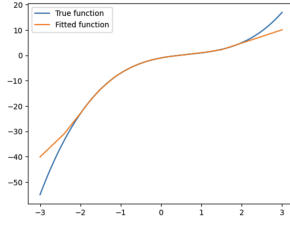
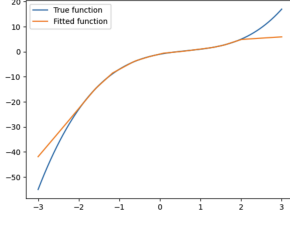
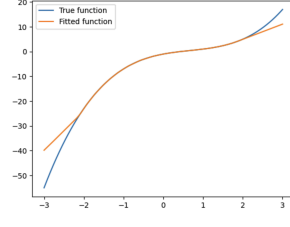
Optimization on relative 'Smooth' Surface

We conducted experiments with both the Conjugate Gradient (CG) and Quasi-Newton methods on optimizing a Multilayer Perceptron (MLP) for function fitting. Our target function is $x^3 - 2x^2 + 3x - 1$. We generated 5k data points range from (-2 to 2), and visualize its fitting results from (-3 to 3)

It is important to clarify that, although our target function involves third derivatives, the optimization challenge pertains to the MLP's parameter space rather than the intrinsic properties of the target function

itself. The results of CG and BFGS methods are as follows

(Each algorithm was tested 5 times to ensure consistency in the results.).

MSE	CG	CG (Results)	BFGS	BFGS (Results)
Layer=1	0.011		0.0090	
Layer=3	0.0067		0.0023	
Layer=5	0.0090		9.3e-4	

The time cost is listed below:

Time	CG	BFGS
Layer=1	1.81s	0.65s
Layer=3	10.24s	4.47s
Layer=5	87.29s	24.83s

Optimization in cases with large third derivatives (or ever higher order)

We further design the following experiment to test problems with large third derivatives. To realize that, we initialize the MLP parameter with random variables using a zero-mean distribution and a high standard deviation ($\sigma=10$). This initialization poses a significant challenge due to the potential for starting points to be far from the function's critical points, leading to high derivatives.

MSE	CG	BFGS
Layer=1	0.07	0.04
Layer=3	3990.07	0.07
Layer=5	3999880350.4	51193.84

The time cost is listed below

Time	CG	BFGS
Layer=1	1.84 s	0.25 s
Layer=3	11.4 s	5.18 s
Layer=5	25.31 s	27.87 s

Explanation: When the third derivative is extremely large, there will be a rapid change in the curvature of the function, the CG method may have difficulty in finding a descent direction. The descent direction in CG is influenced by the second derivative and the previous descent direction. If the second derivative changes dramatically, the assumption that the Hessian matrix remains unchanged during search is no longer valid, which makes CG's error large. As a result, the CG method might struggle to converge efficiently.

5. CONCLUSION

We experimented CG and BFGS in 2 fundamental optimization problems. Our results validated their theoretical performance in time and space consumption. We also show that BFGS works generally better than CG. In particular, BFGS works well in functions involving high order dimensions. One case where CG performs better than BFGS is when the hessian is highly rank-deficient.

6. FUTURE WORK

There are still questions left unanswered:

1. The behavior of the two methods when the objective function is infinitely differentiable.
2. The reason why CG is able to avoid trivial dimensions in rank deficient problems
3. The effect of tuning the restart criteria (max iterations allowed before a forced restart)
4. Generalize to larger and more realistic problems