Project 1: Optimization

Mathew Thiel, William Sykes

March 4, 2025

Abstract

A quick paper on our process and findings while tackling 10 benchmark optimization problems.

1 Introduction

In this paper, we attempt to minimize the given 10 functions using an empirical gradient descent. Will handled the first 5 functions, and Mathew handled the last 5. This problem begins with the empirical gradient descent, which has the form

$$\begin{gathered}\text{Minimize: } \vec{x} \leftarrow \vec{x} - \alpha \nabla f(\vec{x}) \\ \text{where } (\nabla f(\vec{x}) = f(x+\alpha, x_2, ..., x_n) - f(\vec{x}), ..., f(x_1, ..., x_n + \alpha) - f(\vec{x}))\end{gathered} \tag{1}$$

and differs from a regular gradient descent as it takes the finite difference between two points. The step size α, also called the learning rate, is a user-defined value that tunes the gradient descent algorithm. A larger step size increases the likelihood of escaping a local minima, but in turn, increases the likelihood of skipping over a minimum entirely. Smaller step sizes are able to get closer to the minimum but will get stuck in a local minimum and take longer to compute. In this case, the likelihood of reaching the global minimum is entirely determined by the step size and our starting point. For our starting point, x_0, we randomly choose a point within the bounds of the problem using a Pseudorandom Number Generator (PRNG). The PRNG creates a random array with the same dimension as the problem space, which becomes our initial guess. We can then generate as many initial vectors as per the number of iterations we are willing to perform. In the context of a bounded optimization problem, we chose a low iteration number of 100, as higher numbers of iterations may showcase random luck instead of the skill of the optimization method. Through extensive testing for problems with large space size, a regular, linear gradient descent

$$f(x) = x^2$$

The gradient of the objective function is given by:

$$\nabla f(x) = 2x$$

The gradient descent update rule is given by:

$$x_{\text{new}} = x_{\text{old}} - \text{learning\_rate} \times \nabla f(x_{\text{old}})$$

was chosen as it seemed to get closer to a minimum with fewer iterations. We found this especially true with problems involving quadratics like Rosenbrock's Valley, known to have large flats on descents, and Rastrigin's function.

2 PRNG Selection

In the work *A Search for Good Pseudo-random Number Generators*, authors Bhattacharjee et. al. empirically rank various PRNG based on various statistical tests, including the TestU01 library and the NIST statistical test suite [1]. Multiple PRNGs have notable ratings, and for the sake of validation, we have implemented SFMT, Xoroshiro128, Xorshift32 in code, PCG64DXSM, PHilox, and SFC64

using the randomgen Python library, an extension of NumPy's random library [4]. The xorshift prng was chosen as it is very computationally efficient, thus it is fast and easy to implement without using the built pythons built in prng (mersenne twister). We were not sure if memory would be an issue going into the project and xorshift requires little memory, while still having a long period, provided adequate randomness for testing each function. Since the main quality that matters in this context is computation time, aside from the qualities provided in Bhattacharjee et. al., we time and run each PRNG with 10,000 generations. After execution, SFC64 was found to be the fastest by about 0.01 seconds, beating Philox by a large approximate margin of 0.51 seconds.

3 Results

makecell pdflscape

Function	Min.	Time (sec.)	Avg.	Std.	Range	Median
Schwefel's	10265.8913	.0020	-30.9828	291.5430	(-468.5557,486.3077)	-94.1110
1st De Jong	4.0323	.0010	4.1747	1.0815	(-7.6588, 9.9805)	6.2287
Rosenbrock	107.7214	0.1849	-0.0321	0.1456	(-0.2463, 0.2309)	-0.0450
Rastrigin	5829.0313	0.0010	0.8806	13.5895	(-24.9817, 25.4374)	1.4279
Griewangk	13.2840	0.001	7.2725	39.8118	(-58.8178, 63.6148)	18.5353
Sine Envelope Sine Wave	-25.8599	0.3052	-21.2906	1.4335	(-25.8599, -17.76915)	-21.1507
Stretched V Sine Wave	68.9698	0.2599	92.8064	8.9695	(68.9698, 119.3192)	91.3486
Ackley's One	404.4462	0.4173	569.6600	54.9215	(404.4462, 702.2022)	567.6614
Ackley's Two	532.1990	0.3811	576.8136	14.6465	(532.1990, 604.3604)	579.5307
Egg Holder	-3367.1601	0.2347	-304.4381	1578.9610	(-3367.1602, 3360.3457)	-508.9778

Table 1: A table of each optimization function's results for 100 iterations, rounded up to the 4th decimal place

Each function was tested with 100 different initial vectors to start, randomly generated by the previously described PRNG. The Sine Envelope Sine Wave and Stretched V Sine Wave are the most notable since they are fairly close to the global minima provided in the project description. The execution time for the Egg Holder Function is also notable. The results for the Egg Holder function are also very volatile, since the standard deviation is fairly large, even for the function's range. The Ackley's Two and Sine Envelope Sine Wave seem to have low standard deviations for their ranges, and averages close to their minimum estimate.

4 Analysis

Especially since this is a bounded optimization problem with known global minima, these results are fairly underwhelming. For convex problems, especially ones with many troughs like the Foxhole problem, gradient descent seems to be a poor choice for accurate optimization. From the results of this project, it seems like gradient descent works best when there are low numbers of peaks and troughs, or if the optimization problem cannot be analytically solved and it is used as a sort of "best guess" method. We found as different seeds were tested for different sets of starting vectors, often times functions like Rosenbrock's Valley and Rastrigin's function needed a considerable amount of iterations to even begin advancing towards a minimum, Rastrigin's function as well as Schwefels would often get stuck altogether in a local minimum regardless the number of iterations and type of gradient descent function assigned, sometimes straying far outside of the designated bounds given the empirical gradient descent. Testing many differently seeded numbers we felt like this was based on getting lucky with a starting point. To improve our results, it is likely that the step size or number of iterations need to be tweaked. In the case of the number of iterations, with enough iterations, the global minimum can be hit with pure luck, since our starting point is random. However, for functions such as Ackley's One and Two, the optimization seemed to get somewhat stuck around about 400 and 500 respectively. This could mean either there is an error in our functions, or the algorithm is continually getting stuck

in the same troughs. The latter is more likely, as each function was validated by hand. Increasing the step size also had almost no effect, which was somewhat perplexing.

5 Improvements

After completing this project, a few improvements come to mind. Sticking to the constraints of needing to use a PRNG and Empirical Gradient Descent, there are a few important parameters to consider. These parameters include the number of iterations, step size, and starting point. Technically, parameters can be added, as we could possibly separate alpha into two parameters; the step size that increments the gradient, and a separate parameter for the alpha that is added to each x value in the gradient. Separating these parameters would allow us to create a more dynamic gradient, although it is hard to speculate on the effect without testing it. In terms of the step size, we want it to be larger the further we are away from the minimum, and smaller the closer we are to the minimum. So that the step size doesn't become infinitesimally small, there should be an additional tolerance parameter that specifies the minimum allowed step size. Theoretically, if the minimum is reached sooner than with a fixed step size, using the second derivative should be faster. However, this wasn't the case, as not only did the algorithm run slower, but a better minimum was found with a fixed step size. This could be due to the fact that this is an empirical gradient descent, which uses finite differences. There exist many other methods of creating a dynamic step size, such as Adagrad, Adam, RMSProp, Nadam, and AdaMax [3]. The Nesterov Accelerated Gradient was attempted but found to require a different implementation than the empirical gradient. This hinges on a similar idea as the previous second derivative idea, where the momentum and a look-ahead estimate are used to scale the step size [2]. The issue with the implementation was that to be used with the finite derivative present in the empirical gradient descent, there would have to be an initial momentum estimate. This is because each step of the gradient is not incremented by the step size, and is instead the initial vector subtracted from the look-ahead estimate. If there is no initial momentum estimate, the change will always be zero, meaning the gradient is always zero. A workaround could be to choose an arbitrary momentum value to start, but this might be unrealistic, and from initial testing, no improvement was found. Thinking about our start point next, it may be possible to use a process of "guided randomization" to determine our start point. Similar to the "neighborhood of points" approach, where an algorithm traverses over a neighborhood around a point. Imagine a point is randomly generated and a descent is run. For the next point, instead of using an entirely random point, either use that point as the center and create two points from adding and subtracting a fixed tolerance value, or, randomly generate another point that is anywhere outside of a given tolerance. An optimal number of iterations could likely be found through brute force with incremental step size since the optimization problems are fairly conservative on computation time. In terms of speeding up the code without using multiprocessing, there was a large speed-up by vectorizing the math of the given functions with NumPy. After comparing the vectorized, non-for-loop versions of the functions with the basic, for-loop versions, with a large number of iterations, computation times decreased by anywhere from about 0.6 seconds to almost 2.4 seconds for some of the more costly functions. This is a surprising increase, especially just from vectorizing the functions. The fastest computation time was achieved by keeping the math all on one line, and using entirely NumPy instead of the built-in math operators, although the time difference was fairly minimal. A final improvement that could be made is improving how the bounds are handled. In this implementation, the np.clip() function is used to keep the solution vector in the correct bounds. This function clips values larger or smaller than the given bounds to the respective edge point. This did not seem like a proper solution, so another method was implemented, which we named "backtrack", which resets the iteration with a smaller step size if the solution vector has a value over the bounds. Interestingly, both implementations consistently reach a similar minimum. However, computation time varies between the two for specific functions, and it seems random enough not to be significant.

6 Conclusion

In a future version, there could be many improvements, some of which were previously listed. The largest improvement would likely come from using an optimization algorithm different from gradient descent which is more robust to foxhole type formations. If the desire is solely to reach the global min-

ima using gradient descent, then the number of iterations should be greatly increased, and an adaptive step size should be implemented. The goal would then be to reach the global minima consistently at the lowest range of iterations possible. This implementation likely would have been slightly better with an adaptive step size, but from our testing, it seems like the largest problem is being trapped at local minimums. This could be solved through better optimization of the starting point so that more of the solution space is traversed. Increasing the number of iterations again is a simple solution, but it is unrealistic for large-scale, real-world problems.

References

[1] Kamalika Bhattacharjee and Sukanta Das. "A search for good pseudo-random number generators: Survey and empirical studies". In: *Computer Science Review* 45 (Aug. 2022), p. 100471. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2022.100471. URL: http://dx.doi.org/10.1016/j.cosrev.2022.100471.

[2] Aleksandar Botev, Guy Lever, and David Barber. *Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent*. 2016. arXiv: 1607.01981 [stat.ML].

[3] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG].

[4] Kevin Sheppard. *bashtage/randomgen*. original-date: 2018-02-20T09:59:34Z. Jan. 2024. URL: https://github.com/bashtage/randomgen (visited on 02/05/2024).