Study On The Performance And Volatility Of The Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Method

Alexandre Reynaud, Alexandre Di Piazza, Jimmy Wilde
EPFL, Lausanne, Switzerland

***Abstract*—Most of the time, training a neural network involves solving a complex and non-convex optimization problem. Most of the optimization algorithms used are first order. While reducing greatly the computational cost, the convergence rate of said processes is considerably lower than for second-order methods. However, for large scale problems, the huge computational cost of the Hessian matrix is not viable and we have to resort to approximations, resulting in what we call the Quasi-Newton methods. In this paper we wish to study both the performance and the volatility of said methods on neural networks to understand better how the two interact.**

I. Introduction

In this paper, we compare Gradient Descent (GD) with the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) method, a variant of the quasi-Newton method, regarding the task of training neural networks. Gradient Descent is a common optimization algorithm in machine learning, it is a first-order optimization algorithm, has already proven its viability and serves here as a baseline. LBFGS is an alternative to Newton's method, it approximates the Hessian to reduce the memory cost while trying to keep the iteration-wise convergence speed of second-order methods.

II. Methodology

A. Constraints

In order to deal with the combinatorial complexity of our endeavor, we decided to constrain our research to fully-connected neural networks with between 1 and 20 hidden layers, 1 and 50 layer size and homogeneous activations, the cross-entropy loss and the RReLU and LogSigmoid activation functions. When it comes to the data, we opted to work with real-world information, the MNIST dataset, a large database of $28 * 28$ handwritten digits.

B. Data generation

To generate data in a timely fashion, we decided to subsample the MNIST dataset; the result contains 1000 training samples and 1000 testing samples. We then define a function that generates a neural network with the following variables: the number of layers in the NN, the number of nodes in each layer, the activation functions (shared by all layers), and the seed used to pseudo-randomly initialize the weights of the layers. Finally, we generate 10 random number of layers (between 1 and 20), 10 layer sizes (between 1 and 50), 10 random seeds and, for each of the resulting model configurations, we generate a model using the function mentioned above and run both our optimization algorithms for 100 iterations, computing the training loss as well as the test loss at each iteration. The produced data is written in a CSV file.

C. Data Preprocessing

Our first move, to have a usable dataset, was to remove all cases of certain divergence, i.e. all the runs where the loss explodes mid-run, resulting in NANs and INF losses. We registered the percentage of such cases to take them into account and we will discuss them in subsection III-A. Also, in our dataset, for each model configuration, iteration number, and optimization algorithm, we have 10 entries, one for each pseudo-random seed, our other move was to average these entries in order to get closer to the average runs. LBFGS performs better than GD most of the time when it comes to the training loss. It is not as clear of a cut when it comes to the test loss, this is obviously due to fitting and we, therefore, decided to focus exclusively on training data, as it is way more representative of the optimization capabilities of the studied algorithms.

D. Reproducibility

The random parameter generation (layer number, layer size, and seed numbers) is done after setting the seed to 1, furthermore, the weights of each generated model are set using the generated seeds, making our process easily reproducible.

III. Statistical analysis

A. Certain Divergence Cases

The LBFGS method certainly diverges 9.8% of the time, which we detect with the appearance of NAN and INF losses, the GD method never does. We suspect this could be solved by reducing the learning rate of the LBFGS optimization, but it is important to note that many of our runs converge anyway and that we are interested by this duality.

B. Configuration-wise variance

Regarding the other cases (non certain divergence ones), the standard deviation of the average training loss per seed for GD is 0.79 and the standard deviation of the average training loss per seed for LBFGS is 4.65×10^{22}. The results of LBFGS vary tremendously more from one random seed to the other than the ones of GD, which means that there are either divergence cases in the works still or that the algorithm oscillates around the local optima.

C. Studying the mean loss

The average loss per iteration increases way faster for LBFGS than GD (see Figure 1). For LBFGS, the average loss seems to be almost decorrelated from the layer size, with a correlation of -0.13 and no apparent pattern, and to be consistently higher and more unstable than the one of GD (see Figure 2). In comparison to the average loss of GD, the one of LBFGS is consistently higher and more variant but decreases in both these aspects as the number of layers increases (see Figure 3). Finally, considering only the cases using LogSigmoid, we get an average loss of 27752 in contrast with 1.06×10^{22} for RReLU. Due to the instability of LBFGS, GD brings better results regarding the mean loss. These results are due to incomplete cases of divergence and/or convergence instability for LBFGS. To study the impact of these outliers, it would be more interesting to focus on median and best performances.

D. Studying the minimum loss

The minimum loss per iteration decreases way faster for LBFGS than GD, it seems somewhat more variant but said variance vanishes as the number of iterations increases (see Figure 4). For LBFGS, the min loss seems to decrease as the layer size increases and seems to be consistently lower than the one of GD (see Figure 5). In comparison to the min loss of GD, the one of LBFGS is lower for low amounts of layers and becomes similar as the number of layers increases (see Figure 6). Finally, for the LBFGS method, considering only the cases using LogSigmoid, we get a min loss of 4×10^{-4} in contrast with 2300 for RReLU.

E. Studying the median loss

The median loss per iteration decreases way faster for LBFGS than GD and is as stable (see Figure 7). For LBFGS, the median loss given the layer size seems to be quite stable and consistently lower than the one of GD (see Figure 8). In comparison to the median loss of GD, the one of LBFGS is lower for low amounts of layers and becomes similar to the number of layers increases (see Figure 9). Finally, LBFGS performs similarly when using LogSigmoid and RReLU with a respective median loss of 2303 and 2998, but contrast can still be drawn as LogSigmoid results in a loss variance of 229301 while RReLU in one of 5.82×10^{22}.

F. Studying the mean loss when LBFGS does better than GD

We consider here the cases where LBFGS performs better than GD on the last iteration. The mean loss per iteration starts with high volatility and an exponential growth (iterations 1 to 25), looking as it is going to diverge but ends-up decreasing as fast and becoming way less variant (iterations 25 to 75), reaching a point where it consistently outperforms the one of GD (iteration 75 to 100) (see Figure 10). Also, the mean loss for LBFGS is considerably lower than for GD when the number of layers is low and becomes more similar as that number increases (see Figure 11). As the layer size increases, so does the distance between the mean LBFGS and GD losses (see Figure 12). Furthermore, LogSigmoid proves again to be the best performing activation as the corresponding average loss is of 1827 against 2302 for RReLU. Strong parallels can be drawn with the min losses, showing that when LBFGS performs better than GD, it does it consistently around the level observed in the best case analysis. As a matter of fact, at the last iteration, in the cases of non-certain divergence, LBFGS outperforms GD 82% of the time, so around 74% of all cases.

G. Studying The Running time

Performing an iteration of the GD method took in average 0.07 second against 0.61 second for LBFGS. This means that gradient descent is almost 10 times quicker to run than LBFGS on average for the neural networks we built.

H. Variance Per Optimization Algorithm

The standard deviation of the training loss for GD is 30.9 against 9.27×10^{23}. The variance of LBFGS is humongous relative to the one of GD, even after taking the mean run per model configuration.

IV. Results

Here follow the take-aways of our analysis regarding LBFGS. Under all perspectives, the LogSigmoid activation brings better results than RReLU, it yields lower average and minimum losses but also lower loss variance than RReLU. Increasing the layer size seems to also yield better results as the mean loss is almost decorrelated from the layer size and both the median and min loss decrease drastically. Interestingly, as shown by both the median loss analysis and mean loss analysis (when LBFGS works better), LBFGS often beats GD. Increasing the number of hidden layers seems to reduce loss variance, but, as shown by the min and median plots, this is not necessarily desirable as we have seen that the instability of the algorithm leans towards over-performing GD.

V. Discussion

Our approach has 3 pitfalls, it focuses on a specific dataset and results could be biased by the corresponding specifics. Also, it covers only a subset (10 layer sizes, layer numbers, and seeds) of the studied problem, which reduces the reliability of our results. Finally, it is based on running the optimization algorithms using their default parameters, analysing the different impact of a wider array of learning rates could have brought interesting results and should be explored.

VI. Summary

Through our study, we have reaffirmed the volatility of LBFGS but also uncovered interesting aspects regarding its performance. While being slower and less stable than GD, there are significantly more runs that outperform GD than the opposite. Under the right conditions, we get median losses that outperform GD so drastically that it could compensate for the difference in running time. For example, one could run k parallel LBFGS optimizations and keep the best one, this would outperform the already excellent median runs with probability $1 - (1/2)^k$.

VII. Acknowledgements

The authors thank Martin Jaggi and Nicolas Flammarion and for their support and helpful suggestions.

References

[1] N. S. Keskar and A. S. Berahas, "adaqn: An adaptive quasi-newton algorithm for training rnns," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 1–16.

[2] R. H. Byrd, S. L. Hansen, J. Nocedal, and Y. Singer, "A stochastic quasi-newton method for large-scale optimization," *SIAM Journal on Optimization*, vol. 26, no. 2, pp. 1008–1031, 2016.

Appendix

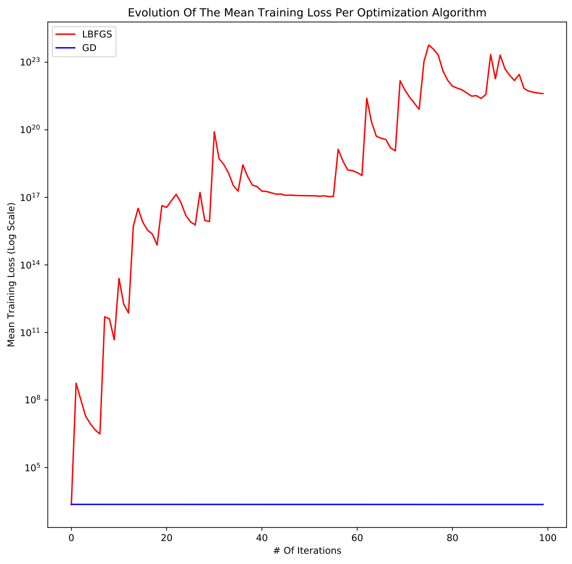

Fig. 1

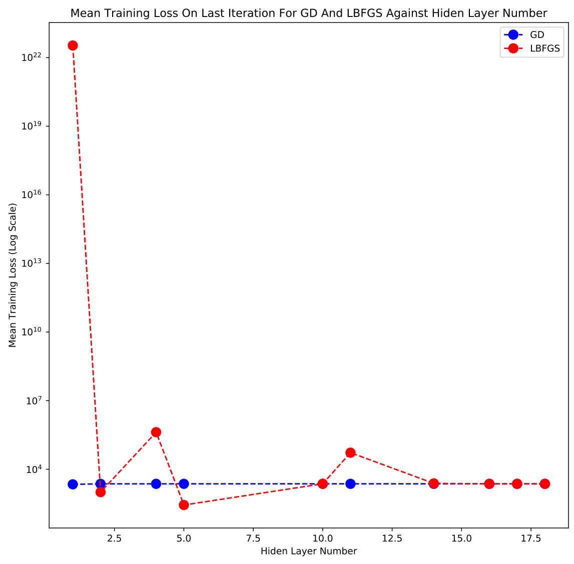

Fig. 3

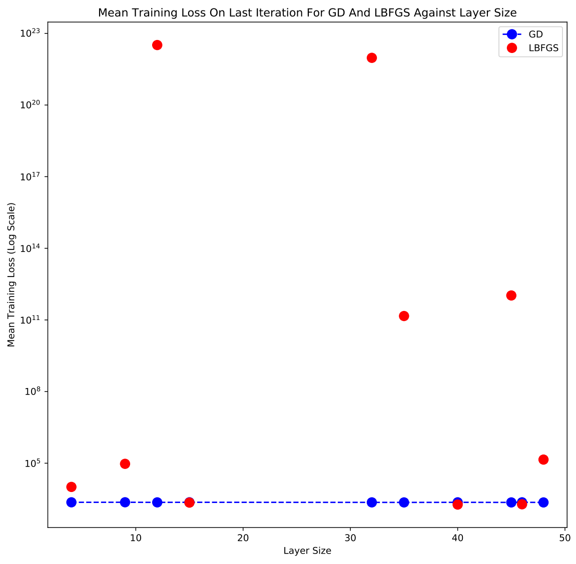

Fig. 2

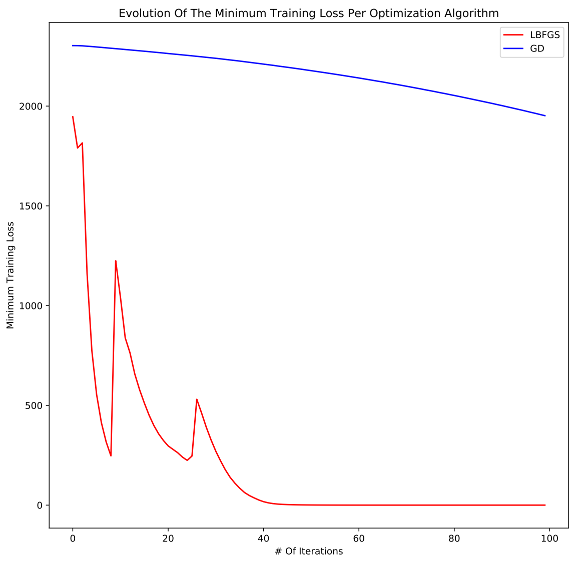

Fig. 4

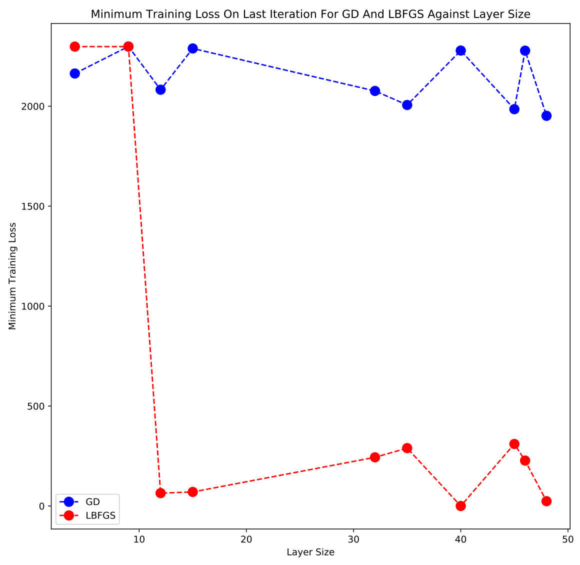

Fig. 5

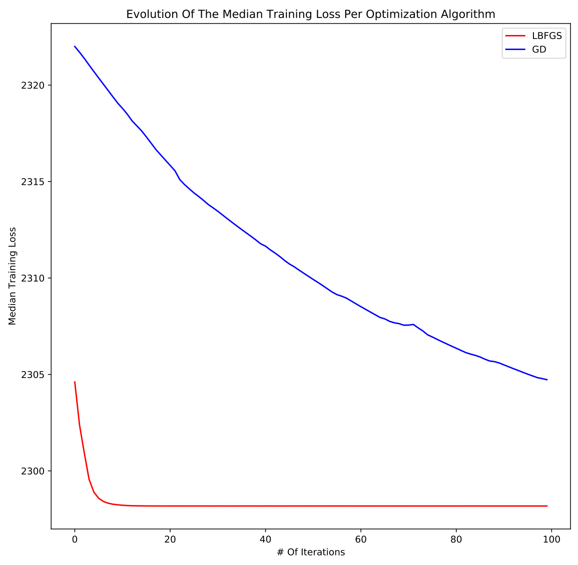

Fig. 7

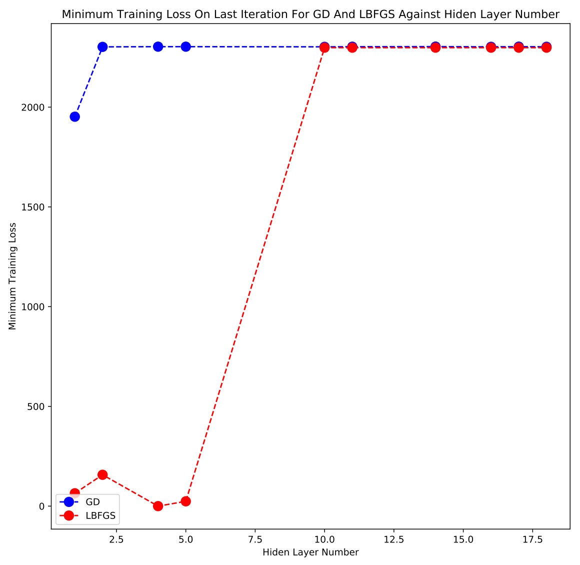

Fig. 6

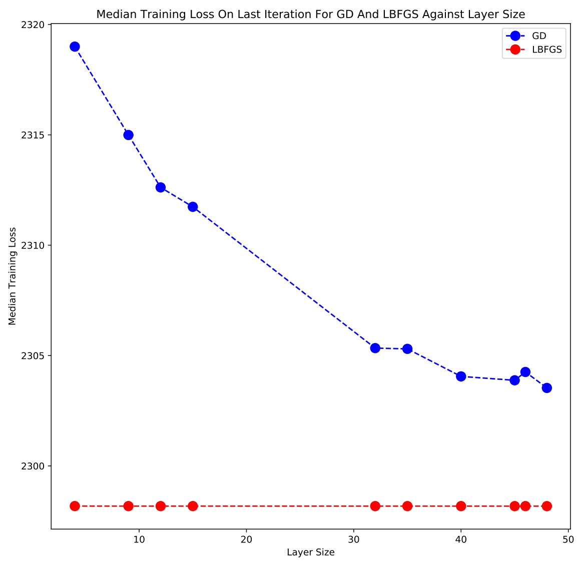

Fig. 8

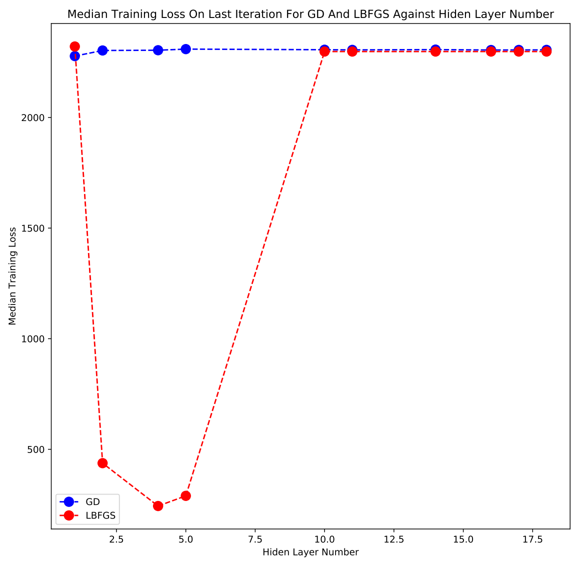

Fig. 9

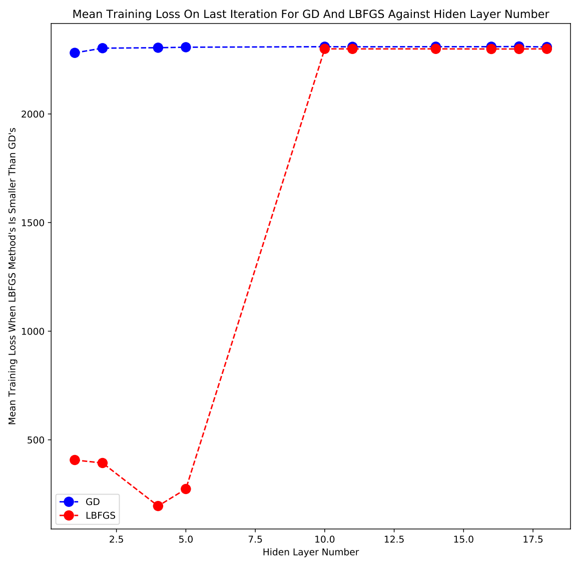

Fig. 11

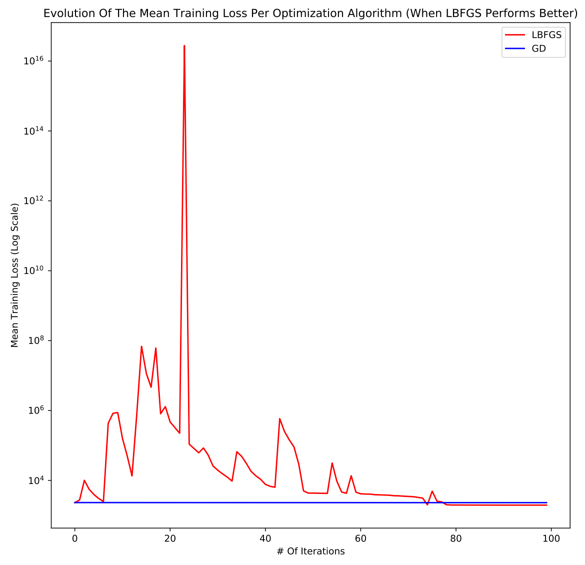

Fig. 10

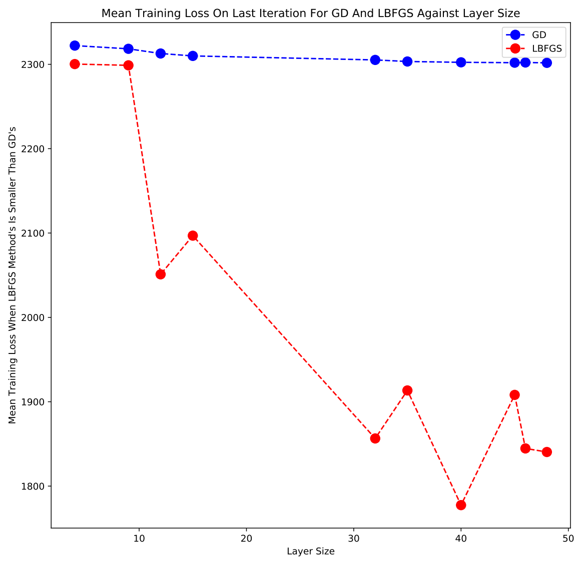

Fig. 12