LEARNING TO OPTIMIZE QUASI-NEWTON METHOD BASED ON RECURRENT NEURAL NETWORK

A PREPRINT

Hongpei Li
ID:2021110408
SUFE

Han Zhang
ID:2021111452
SUFE

January 9, 2024

ABSTRACT

The learning-based optimization method has gained much attention. Traditional optimization problems usually use human empiricism to select optimization methods and tune parameters. Learning-based methods use the Machine Learning method to gain information not only from local gradients but also "global" things from previous steps information and features of the distribution of data. Our works focus on a specific and one of the most popular second-order methods called the Quasi-Newton method which is implemented by modifying an approximation of the Hessian matrix as a precondition. Instead of the traditional method using a Correction Matrix, we train RNN as a modifier of the preconditioned matrix in a Quasi-Method. We show numerical results in Logistic Regression.

Keywords Learning to optimization · Quasi-Newton · LSTM

1 Introduction

Quasi-Newton(QN) optimization methods like the Davidon-Fletcher-Powell (DFP) method, Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, and Symmetric Rank 1 (SR1) method provide computationally efficient approximations to the Hessian matrix used in Newton's method. These approaches are highly regarded for their effectiveness in solving smooth unconstrained optimization problems. Studies have showcased that, under specific assumptions, pairing quasi-Newton methods with practical line search techniques results in super-linear convergence rates. The efficiency of these techniques often lies in their swift estimation of the true Hessian matrix. Coupled with an exact or inexact line search that ensures the curvature condition, quasi-Newton methods demonstrate their efficiency across various optimization scenarios.
Consider at least twice continuously differentiable function f, the framework of QN's iteration is:

$$x_{k+1} = x_k - \alpha_k H_k \nabla f(x_k)$$

where α is step size(usually gained by linear-search), H_k is the inverse Hessian approximation. And Quasi-Newton Class method focuses on how to get more efficient H_k where BFGS, DFP, and SR1 use iterative schemes as:

$$H_{k+1}^{BFGS} = H_k - \frac{s_k y_k^T H_k + H_k y_k s_k^T}{y_k^T s_k} + \left(\frac{y_k^T H_k y_k}{y_k^T s_k} + 1\right)\frac{s_k s_k^T}{y_k^T s_k}$$

$$H_{k+1}^{DFP} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{y_k^T s_k}$$

$$H_{k+1}^{SR1} = H_k + \frac{(s_k - H_k y_k)(s_k - H_k y_k)^T}{(s_k - H_k y_k)^T y_k}$$

Our work belongs to Learning to Optimization(L2O), which is a new and popular topic these years. L2O focuses on using Machine Learning methods to accelerate traditional Optimization algorithms. Traditional optimization

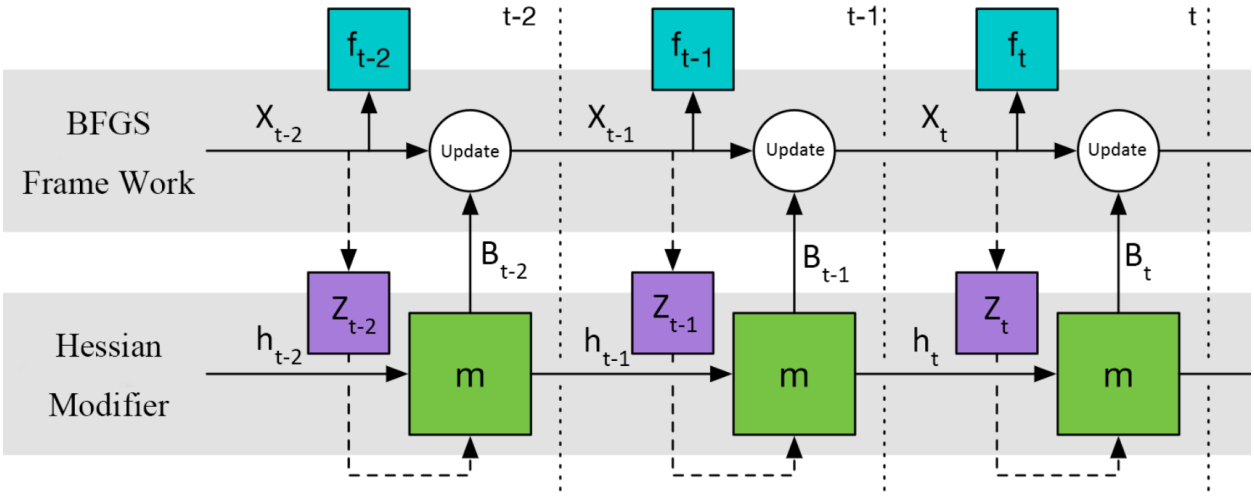

Figure 1: Flow diagram of LSTM-BFGS

methods build on components of basic methods, such as gradient descent, conjugate gradient, Newton step, simplex basis updating, and stochastic sampling, and are theoretically sound. Most traditional optimization methods can be represented in a few lines of code and their theory guarantees their performance. When solving optimization problems in practice, we can choose a method that is appropriate for a particular problem type and expect that the method will return no worse solution than the one it guarantees.

L2O is an alternative paradigm that develops optimization methods by training, i.e., by learning about the performance of sample problems. This method may lack a solid theoretical foundation, but the training process improves its performance. The training process is usually offline and time-consuming. However, online applications of the method aim to save time. For problems where it is difficult to obtain an objective solution, such as non-convex optimization and inverse problem applications, the solution of a well-trained L2O method may be of better quality than that of a classical method.

There have been several works that use the L2O framework in the Quasi-Newton method. WorkLi et al. [2020] uses the twin-LSTM framework to select step size and weight used in selecting a modification method in a convex combination. WorkTan et al. [2023] uses the PnP framework to optimize a regularised Quasi-Newton method which is relatively similar to the PnP-ADMM method. WorkLiao et al. [2023] train a special neural network designed for parameter-efficient representations of unitary matrices to approximate inverse Hessian matrix used in the Quasi-Newton method. Motivated by this work, we design an RNN-based Quasi-Newton(RQN) to approximate the inverse Hessian matrix.

2 Method Design

In our work, we train an LSTMnn to learn a mortification of an approximated inverse Hessian matrix inspired by the iterative scheme of the BFGS method. We use $\mathbf{z_k} = \begin{pmatrix} s_k \\ y_k \end{pmatrix}$, where $s_k = x_k - x_{k-1}, y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, as input of LSTM. And the LSTM returns a matrix $A_{k+1} = LSTM(z_k, \theta_k)$ where θ_k is the hidden state of LSTM, and the scheme of the inverse Hessian approximation matrix is $H_k = A_k A_k^T$ to guarantee the matrix we gain is symmetric and semi-definite. Then the iteration of our algorithm is shown in Algrithm 11, and the idea is shown as flow diagram in figure 11. Notice that in our algorithm, Line Search is not differentiable so we detach stepsize from gradient graph maintained by pytorch.

Algorithm 1 LSTM-based Quasi-Newton method

Given x_0 and f, initialize $k \leftarrow 0$, determine $\nabla f(x_0)$, initiate $\theta \leftarrow 0$
while x_k not converged **do**
  if $k = 0$ **then**
    $H_k \leftarrow I$
    $p_k \leftarrow \nabla f(x_k)$
    determine α_k with Wolfe line search(we show it in **Algorithm 2**)
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  else
    $\mathbf{z_k} = \begin{pmatrix} x_k - x_{k-1} \\ \nabla f(x_{k+1}) - \nabla f(x_k) \end{pmatrix}$
    $A_k \leftarrow \text{LSTM}(z_k; \theta)$
    $H_k \leftarrow H_{k-1} + A_k A_k^T$
    $p_k \leftarrow \nabla H_k f(x_k)$
    determine α_k with Wolfe line search
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $x_{0k+1} \leftarrow x_k + \alpha_{0k} p_{0k}$
  end if
  $k \leftarrow k + 1$
end while

Algorithm 2 Strong Wolfe Line Search

Require: $f(x)$: Objective function
Require: $\nabla f(x)$: Gradient of the objective function
Require: x_0: Initial point
Require: α: Initial step length
Require: $\beta_1, \beta_2 \in (0, 1)$: Constants for Wolfe conditions
$x \leftarrow x_0$
$\alpha_{\text{min}} \leftarrow 0$
$\alpha_{\text{max}} \leftarrow \infty$
iteration $\leftarrow 0$
while iteration $<$ max_iterations **do**
  Compute $f(x)$ and $\nabla f(x)$
  Evaluate Wolfe conditions
  if Wolfe conditions are satisfied **then**
    return α
  else if Wolfe conditions not satisfied **then**
    if iteration $= 0$ **then**
      $\alpha \leftarrow$ initial step length
    else if Wolfe conditions violated in the previous step **then**
      $\alpha \leftarrow \text{interpolate}(\alpha_{\text{min}}, \alpha_{\text{max}})$
    else
      $\alpha \leftarrow \text{extrapolate}(\alpha_{\text{min}}, \alpha_{\text{max}})$
    end if
    if $f(x + \alpha \cdot \nabla f(x)) > f(x)$ **then**
      $\alpha_{\text{max}} \leftarrow \alpha$
    else
      $\alpha_{\text{min}} \leftarrow \alpha$
    end if
  end if
  Increment iteration
end while
return α (may not satisfy Wolfe conditions)

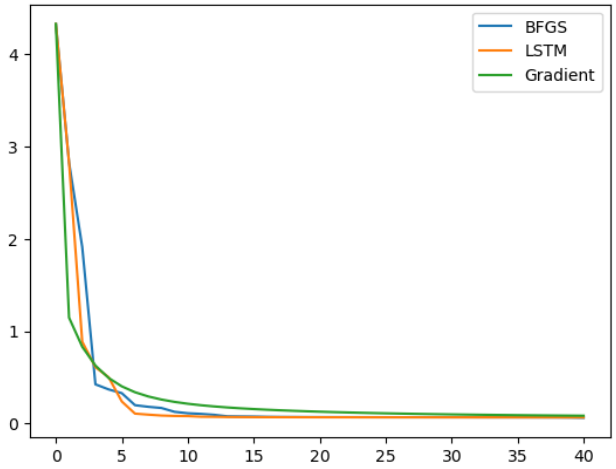

Figure 2: Least-Square problem

Algorithm 3 Simple Line Search for the whole batch

Require: Objective function f (a vector function)
Require: Current point x with shape (batch size,n)
Require: Descent direction d
Require: Loss and gradient at current point loss, grad
Require: Armijo and Curvature condition parameters c_1, c_2
Require: Maximum iterations max_iter
  Initialize α with a starting value
  for each iteration within max_iter **do**
    Evaluate the new loss at $x + \alpha d$
    Do the following job for all samples in this batch
    1. Check Armijo condition for sufficient decrease at new point
    2. Calculate the gradient at the new point
    3 . Check Curvature condition at new point
    if both conditions are satisfied for **any** sample in this batch **then**
      return the current α
    else if Armijo condition fails **then**
      Reduce α
    else if Curvature condition fails **then**
      Increase α
    end if
  end for
  return final α after maximum iterations

We set objective function as $\min_{\theta} \sum_{k=1}^{K} f(x_k)$, where K is steps of model training. As we always train a batch simultaneously, so we use a Wolfe Line search for a batch here. Besides, we use a low-accurate Wolfe Line Search in training instead of one with a huge number of iterations. Because if we search so many times, LSTM can not learn so much information as even H_k is an identical matrix, the function can descent just like a GD with line search, but if we don't use it and just set $\alpha_k = 1, \forall k$ it's easy to diverge and then the parameter of LSTM will become NaN soon. So we use small-iteration-line search to avoid $\inf$ in training, which we call Simple Line Search in **Algorithm 3**.

```
bfgs_time 6.688848257064819
lstm_time 1.8233678340911865

<matplotlib.legend.Legend at 0x19fbcef6d60>
```

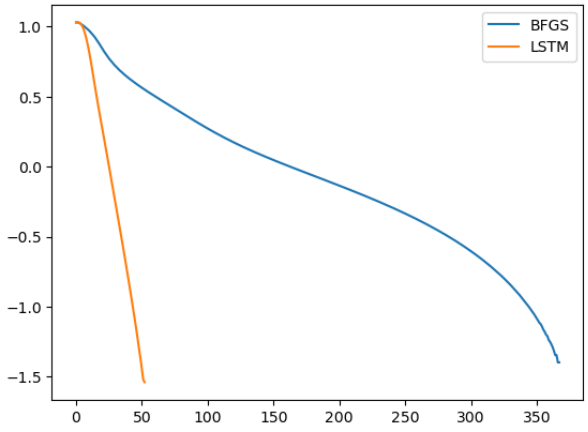

Figure 3: Random Logistic Regression

3 Numerical experiment

3.1 Least-Square Problem

Initially, we write BFGS-step by ourselves as baseline and run it for Least-Square problem $\min_x ||Wx - Y||^2$ with normal random sampling parameters W and Y. We compared our LSTM-BFGS with gradient descent with constant step and our implemented BFGS. We plot the function value and iteration number and show the result in Figure 2. We can find in this easy problem, that LSTM-BFGS isn't so powerful, We believe the reason is that we use Normal random sampling, which we know is sampling at the surface of a ball. So, the condition number $\frac{|\lambda_{min}|}{|\lambda_{max}|}$(where$\lambda_{max}, \lambda_{min}$ are highest and lowest eigenvalue of matrix W, and we always use it to ensure the efficiency of algorithm. The reason is when this number becomes relatively high, the matrix will be close to a ball in its space and the problem becomes easy for iteration) is surely close to 1, so all algorithms can be fast. By the way, our method surpasses Gradient Descent.

3.2 Logistic Regression (Random sample)

Then we consider Logistic Regression with the l2-norm problem, in this part, we compare our LSTM-BFGS method with our BFGS method. Similarly, we get W and Y by Random Sampling, where W is a normal random sample and Y is a uniform random sample. The result is shown as 3 We can find our LSTM-BFGS performs better than our BFGS. Linear Regression Problem is:

$$\min_x f(x) = -\frac{1}{m}\sum_{i=1}^{m}(Y_i \log \sigma(W_i x) + (1 - b_i)\log(1 - \sigma(W_i x))) + \frac{\lambda}{n}||x||^2$$

3.3 Logistic Regression (real data)

Recently, we sample from a real EMNIST dataset, a data set for hand-write recognition. And use it to generate parameters of the Linear Regression Problem with $\lambda = 0.7$. We choose an epoch=10, each epoch contains 100 batches with batch size 64 and set m=500,n=100 The steps we generate data are:

- **Prepossessing** For efficiency, we first reduce the size of each picture to 100x100.
- **Generate balanced classes** We choose the same size of data from each class of origin data set and then generate a balanced data set.
- **Generate W and Y** To generate a sample, we choose 2 classes randomly and choose $\frac{m}{2}$ for each selected class randomly. We flatten each picture to 1 1-dimensional vector and join them to generate a matrix W with

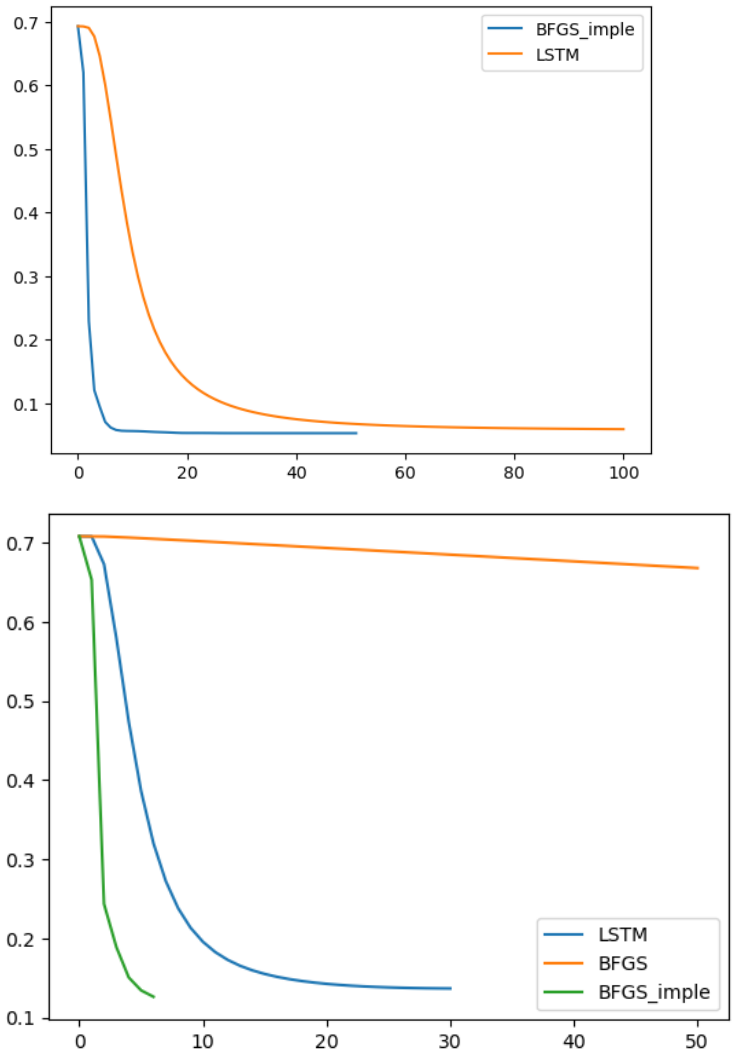

Figure 4: Random Logistic Regression

shape(m,n). We set the label of one of the selected classes as 1 and another one as 0 and joined these 500 labels to a vector. Finally, we joined W and Y to store in a matrix with shape(m,n+1)

- **Generate batch using Data Loader** We use "Shuffle" to avoid over-fitting, and then use Dataloader to generate W and Y according to batch size.

In this part, we use a well-implement BFGS algorithm in GitHub https://github.com/rfeinman/pytorch-minimize. The result is shown in4, we can find our LSTM-BFGS performs better than our LSTM but worse than BFGS in this package. One of the most important reasons we believe is our simple line search can't find a suitable step size efficiently, as it handles the whole batch simultaneously and stops if one sample's step size is suitable. But the advantage of batch version line search is that we can handle data in a batch with a single call line search function and can take advantage of efficient matrix multiplication in GPU as the result in the first picture of 5.Comparing with LSTM-BFGS before training(show in 5), we can find that our training is effective for optimization.

4 Conclusion

Our work shows that the LSTM-based framework can improve optimization though it's not so outstanding right now, we still have kinds of jobs or ideas using this framework.

```
LSTM-BFGS running time:1.9574816226959229,BFGS running time:12.973588943481445
```

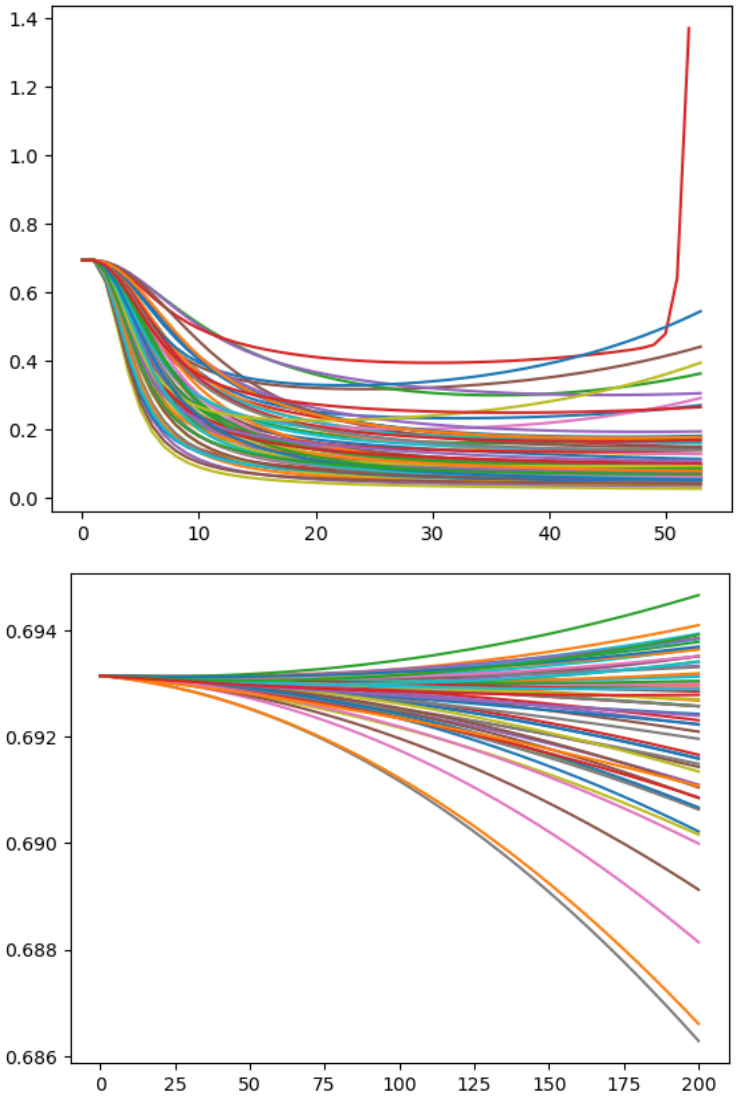

Figure 5: Run for batch 64, $\lambda = 0.1$
First picture is running time, and other 2 pictures are images of function values with iteration steps for LSTM-BFGS and well-implement BFGS respectively.

- We need suggest a more useful method to normalization.
- We need improve capacity of our Line Search or just use well-implemented Strong Wolfe Line Search such as function contained in package we downloaded from github.
- We can try different input and objective function for train LSTM or even other more efficient variants of LSTM suggested in other works(see Chen et al. [2021]).

That's same for the whole area of L2O Though several works show L2O's great performance in practical use. However, L2O still faces difficulties,for example, there are some questions for it: How to guarantee convergence? How to guarantee a converge rate? And the most important one is how can we train more efficiently and improve expandability as we cannot always guarantee huge scale of data to train L2O in practice.

References

Maojia Li, Jialin Liu, and Wotao Yin. Learning to combine quasi-newton methods. In *Proceedings of the 12th Annual Workshop on Optimization for Machine Learning*, Rochester, NY, USA, 2020. Workshop on Optimization for Machine Learning.

Hong Ye Tan, Subhadip Mukherjee, Junqi Tang, and Carola-Bibiane Schönlieb. Provably convergent plug-and-play quasi-newton methods, 2023.

Isaac Liao, Rumen R. Dangovski, Jakob N. Foerster, and Marin Soljačić. Learning to optimize quasi-newton methods, 2023.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark, 2021.