Hessian-free Second-order Optimization on Deep Neural Network

Yifeng Tao
Carnegie Mellon University
Pittsburgh, PA 15213
yifengt@andrew.cmu.edu

Chaojing Duan
Carnegie Mellon University
Pittsburgh, PA 15213
chaojind@andrew.cmu.edu

Rui Zhu
Carnegie Mellon University
Pittsburgh, PA 15213
rz1@andrew.cmu.edu

Abstract

Compared with first-order methods, second-order methods converge faster and take better care of the issues of vanishing gradient and pathological curvature in training deep neural networks (DNN). We implement the second-order Hessian-free (HF) method, which is intrinsically quasi-Newton, in training DNN for MNIST classification. We make thorough comparisons between HF and popular first-order methods as well as another commonly used second-order method: L-BFGS. Finally, to combine the advantages of both momentum and second-order methods, we develop momentum L-BFGS, which converges fastest among all the suitable methods for this task.

1 Introduction

The topic of updating weights in deep neural networks (DNN) with back-propagation [6] during training process has been receiving considerable amount of interest in the past decade. The most naive algorithm for this problem is gradient descent (GD) with mini-batches. However, GD suffers from "vanishing-gradient" issue (Figure1) – decay of error signals during back-propagation, as the depth of neural network surges [2], thus bringing about severe under-fitting problem.

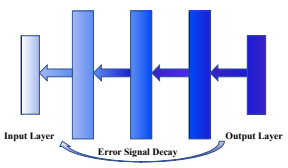

Figure 1: Vanishing gradients.

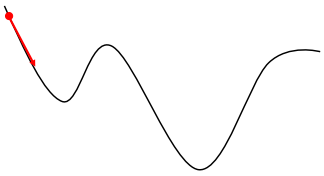

Figure 2: Local minima.

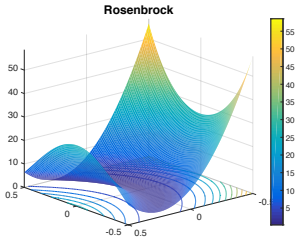

Figure 3: Pathological curvature.

In addition, GD takes steps according to gradients with no consideration of the local curvature.

There are two primary hypotheses to explain the drawbacks for gradient descent in training DNN. Firstly GD gets trapped in local minima or zigzagging as shown in Fig. 2. Second, it is widely accepted that the prevalence of pathological curvature [4] like the Rosenbrock function $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ poses serious threat for first-order methods to traverse through, as is shown in Fig. 3.

Second-order methods are developed to utilize information of local curvature to scale the updates [1]. It converges fast even with pathological Hessian of large condition number. Additionally,

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

during the back-propagation procedure, both the reduction along direction $\nabla f^T p$ and the curvature $p^T H p$ proximate to zero, which makes second-order algorithms applicable to overcome the vanishing gradient problem by pursuing direction $p_k = w_{ik}$ with a reasonable rate [10]. However, the calculation of Hessian matrix and its inverse are both time and space consuming, because the enormous amount of weights in modern neural networks [12].

Hessian-free (HF) optimization proposed by [11] aims to update weights of deep neural networks without calculating the Hessian matrix or its inverse explicitly. In this project, we would like to implement the Hessian-free optimization method for the feed-forward network in MNIST classification, and quantitatively compare their performance against classical GD and L-BFGS.

2 Background

For a DNN, we denote all the weight and bias parameters as θ, and the loss of network to be optimized as $f(\theta)$. The vanilla GD simply updates the parameters with a fixed step in the direction of steepest gradient descent:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t). \tag{1}$$

In modern DNN, adaptive first-order methods such as Momentum, Nesterov accelerated gradient, RMSprop, AdaDelta and Adam have been proposed to overcome the problems of pathological curvature and local minima by accumulating information from previous points in the optimization trajectory. Two ideas are in the core of these methods: momentum and scaled weight update.

2.1 First-order Method

2.1.1 Momentum

Traditional GD encounters problems when we have ravines (mixed large and small curvatures), common around local minima. In addition, optimization extremely slows down when it come to a plateau, an area where gradient is small [4]. To damp the harmful oscillations across the slopes of ravines and push the parameter out of plateau, the conception of momentum is introduced to damp the gradient in direction of large curvature by adding in historical gradients with opposite signs [16].

The basic momentum algorithm is achieved by calculation of gradients and leaps in the direction of the updated accumulated gradient:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla f(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned} \tag{2}$$

Standard momentum method is improved by Nesterov accelerated gradient (NAG) [14], which can approximate the next-step position of the parameters θ by computing $f(\theta_t - \gamma v_{t-1})$ instead of $f(\theta_t)$. The update prediction keeps the update procedure steady and greatly improves the performance of RNNs on many different tasks [3].

Here the decaying coefficient γ controls how much momentum is conserved in the next step.

2.1.2 Scaled Weight Update: AdaGrad & AdaDelta & RMSprop

Momentum methods can be applied to adjust the updates according to historical updates and accelerate the GD process. Another stream of methods is to update each parameter in different scales based on its updating frequency.

AdaGrad algorithm updates the infrequent and frequent parameters respectively in large and small scales [5]. The update rule of AdaGrad algorithm is:

$$\theta_{t+1} = \theta_{t,i} - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f(\theta_t), \tag{3}$$

where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix and each diagonal element $G_{t,ii}$ is the sum of the squares of the gradients w.r.t. θ_i up to time step t.

However, G_t in the denominator keeps increasing since each added term in each step is positive, which leads to diminishing learning rate and $\theta_{t+1} \approx \theta_t$ eventually. RMSprop [17] (unpublished) and AdaDelta [19] offer solutions to monotonically decreasing learning rate by replacing it with a decaying average of all historical squared gradients $E[g^2]_t$:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2, \tag{4}$$

where RMS refer to the form of root mean squared in the update rule.

RMSprop performs well with on-line and non-stationary settings [7] and the update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla f(\theta_t). \tag{5}$$

To maintain the consistency of units in the update, AdaDelta [19] also defines a squared term as an exponentially decaying average:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2. \tag{6}$$

Hence, the update rule of AdaDelta is:

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} \nabla f(\theta_t). \tag{7}$$

2.1.3 Adam

Combining the advantages of all algorithms above, [7] proposed Adaptive Moment Estimation (Adam) algorithm, which is capable of computing adaptive learning rates for each parameter and works well in problems with a large set of parameters. It also keeps an exponentially decaying average of historical gradients m_t as well as squares of gradients v_t:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned} \tag{8}$$

The biases of m_t and v_t are neutralized with bias-corrected moment estimations:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{9}$$

The update rule of Adam is almost same with AdaDelta and RMSprop:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{10}$$

2.2 Second-order Methods

Second-order methods convergence fast when the error surface is quadratic or quasi-quadratic instead of planar; they also solve the issue of large condition number.

2.2.1 Newton's Method

Newton's method is the vanilla second-order method, which utilizes the second-order information of Hessian. The parameters in Newton's method are updated as:

$$\theta_{t+1} = \theta_t - \eta (\nabla^2 f(\theta_t))^{-1} \nabla f(\theta_t). \tag{11}$$

Therefore, the calculation of Hessian is required in each update, which makes Newton's method suffer from time-consuming calculation and space-consuming storage. For example, $\mathcal{O}(n^3)$ space and computation are necessary to solve a function with domain of $\mathbb{R}^n$ in Newton's method.

2.2.2 L-BFGS

Due to expensive Hessian computation when the problem scales, limited memory algorithms have been proposed to approximate the Hessian by saving only several vectors of length n instead of $n \times n$ matrices [18]. L-BFGS[15] is one of the quasi-Newton methods as a limited-memory version of BFGS[9], which uses limited space and does not need to compute the Hessian matrix explicitly.

3 Method

3.1 The Hessian-free Method

HF Newton method is applied to compute the search direction by solving the Newton equation below with conjugate gradient (CG) [18],

$$\nabla^2 f(\theta_t) p_t^n = -\nabla f(\theta_t) \tag{12}$$

In standard Newton's method, storage of $n \times n$ matrices and calculation of matrices inverse are required, which is inapplicable for cases of large number of parameters n models. In Quasi-Newton methods, low-rank or (block) diagonal matrices are preferable to approximate the inverse of Hessian, resulting in limited application in deep-learning problems. Compared with these algorithms above, computing the matrix-vector product Bv for an arbitrary vector v is easier to implement and more generally used. The product Bv can be calculated by finite difference method,

$$Bv = \lim_{\epsilon \to 0} \frac{\nabla f(\theta + \epsilon v) - \nabla f(\theta)}{\epsilon} \tag{13}$$

which is calculated for the exact value of matrix B without any additional requirement for B. Pseudo-code for standard HF optimization is shown in Algorithm 1

Algorithm 1 HF Method

for $t = 1, 2, ...$ **do**
  $g_t \leftarrow \nabla f(\theta_t)$
  formulate a local quadratic approximation around $f(\theta_t)$:
    $f(\theta_t + p) \approx f(\theta_t)^T p + \frac{1}{2} p^T \mathbf{B} p$
    where $\mathbf{B} = \mathbf{H} + \lambda \mathbf{I}$ is the damped Hessian (need not compute $\mathbf{H}$)
  $p_t = \arg\min_p f(\theta_t + p)$ with Conjugate Gradient Descent
  $\theta_{t+1} \leftarrow \theta_t + p_t$
end for

Theoretically, $n = \dim(\theta)$ iterations are required for convergence[18]. However, we do not always wait for CG to converge because CG in practice makes significant leaps in the first few iterations rather than n [11]. To determine the stopping condition, *residual* is defined as,

$$r_t = \mathbf{B} p_t + g_t. \tag{14}$$

Then in most versions of HF, the standard stopping condition is [11]

$$||r_t|| < \min(\frac{1}{2}, ||g_t||^{\frac{1}{2}}) ||g_t||. \tag{15}$$

HF algorithm is outstanding for several reasons. Firstly, it can calculate the product Bv for some vector v in each CG iteration accurately by Equation 13 instead of making approximations as Quasi-Newton methods do. Based on the stopping condition, far fewer iterations than n can be sufficient to obtain enough information to approximate the solution to the quadratic approximation. In addition, storage or computation of invert Hessian is no longer needed and only matrix-vector product is called to minimize positive definite quadratic cost functions in linear CG procedure. Furthermore, HF does not directly minimize the residual term $||r_t||^2$ but instead the quadratic term:

$$\frac{p_t^T \mathbf{B} p_t}{2} + g_t^T p_t,$$

which should be taken into consideration actually. Moreover, the steps for gradient and line-searches can be also calculated with larger mini-batches in large datasets because neither of them is required frequently in the procedure of HF [13].

3.2 Tweaks to HF

The paper[11] mentions several tweaks and tricks in HF which could possibly help in performance. Particularly two tweaks are highlighted: Hessian damping and CG back-tracking.

For Hessian damping, an damping parameter λ_damp is added to the Hessian approximation for conditioning Hessian toward an unit Hessian. The damping parameter is controlling how "conservative" the approximation is thus preventing overshooting in face of small curvature, and is updated using a Levenberg-Marquardt style heuristic.

For CG backtracking, the intuition is, the last few steps of CG may not be trustworthy thus we might need to consider prune some of the last steps of CG. This is done by "backtracking" the steps of CG after it has terminated, and prune the step which does not help reducing the objective.

4 Experiment and Analysis

We implement all the experiments in Matlab, code for the project is available on Github [1].

4.1 Dataset

We use the MNIST dataset [8] in all of our experiments. For efficient evaluation of multiple training methods, especially those relatively expensive second-order methods, we choose a subset of 5,000 samples for training, and 1,000 for testing.

4.2 Network Architecture

We design a fully-connected network for our experiments. We use *fc(dimin, dimout)* to denote a fully connected layer with input dimension of $dimin$ and output dimension of $dimout$. The network is arranged as follows: *fc(784, 200)*, *fc(200, 30)*, *fc(30, 10)*. Bias terms are included in each input layer. The total parameter dimension $n = 81,840$. We use sigmoid as activation for the first and second layer, and softmax for the third layer. The loss is computed as cross-entropy between the output and labels (both are one-hot vectors) , plus the weight-decay term $\frac{1}{2}\lambda||w||_2^2$ where λ is the hyper-parameter of weight-decay and w is the vectorized form of all weights in the network. Across all of our experiments we set λ to 2e-4.

Fig. 4 shows the neural network for the implementation of our algorithms.

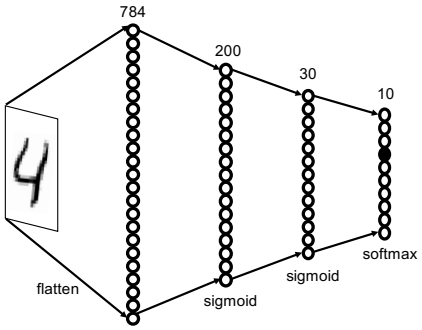

Figure 4: Architecture of neural network.

4.3 First-order Methods

We implement the HF algorithm based on the code [2] of [11], and tested several most popular choices of first-order methods in optimizing the network, including GD with/without back-tracking, accelerated GD with momentum, NAG, AdaGrad, RMSprop, and Adam. To make the comparison fair, we try to use the same hyperparameters for them, for example, we take $\eta = 0.001$ as the step size, $\epsilon = 1e-8$ to prevent zero denominator, decay of momentum $\gamma = 0.9$, decay parameter for Adam is set to $\beta_1 = 0.9$, $\beta_2 = 0.999$. [3] We run each algorithm for 10 times and compare the average performance

[1] `https://github.com/HessianFreeOptimization/HessianFreeOptimization`

[2] `http://www.cs.toronto.edu/~jmartens/research.html`

[3] We noticed that the different methods prefer different parameter settings, we make a compromise here to set them the same, which hopefully can give the overall order of convergence of each algorithm.

of difference between objective value f subtracted by the lowest value achieved $\hat{f}^*$ with regard to training epochs, number of evaluations of objective and gradient.

Fig. 5 shows performance of first-order methods against three measurements. Adam, which combines both momentum and scaled weight updates, performs best among all first-order methods.

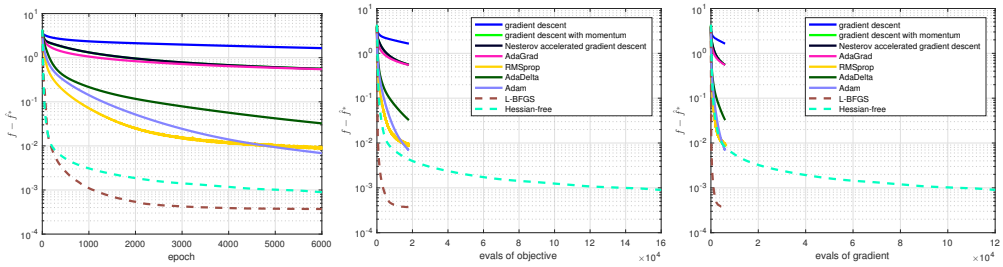

Figure 5: Convergence of first and second-order methods

Fig. 6 gives the training (solid lines) and test (dashed lines) error of first-order methods against training epochs, which is consistent with the results of loss function.

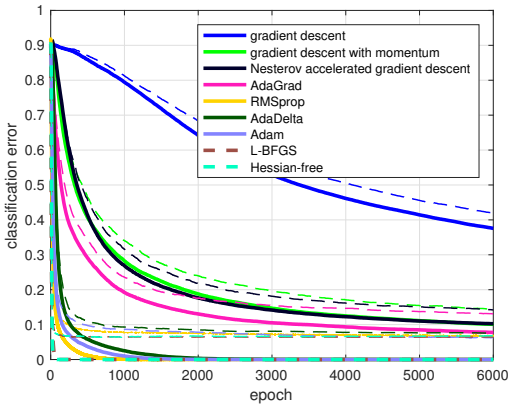

Figure 6: Training and test error of first and second-order methods.

4.4 Second-order Methods

We further implement second-order methods including L-BFGS and HF methods. As shown in Fig. 5 and Fig. 6, HF and L-BFGS easily beat all of the first-order methods in optimizing the network, even when we compare the number of objective function and gradient evaluations.

Notice that our second-order methods use backtracking line search to ensure convergence. To make fair comparison, gradient descent with backtracking is also implemented, but still falls far behind second-order methods with regard to convergence rate as shown in Fig. 7.

For comparison among second-order methods, we compare HF and L-BFGS ($m = 7$) in Fig. 8. Ten trials of both methods are plotted in the background with light color, and their mean values are plotted with dashed lines in foreground. We may observe that HF method converges fast at the beginning, but is quickly outperformed by L-BFGS afterwards. However as shown in Fig. 5, HF outnumbers all other methods by using 10 times more evaluations of objective and gradient, due to its expensive inner loops of conjugate gradient descent. Mathematically if we look at the time complexity of the CG procedure in HF, it would take n steps maximum, whose time complexity in one epoch is $\mathcal{O}(n^3\sqrt{k})$ where k is the condition number for the matrix, which is no better than Newton's Method ($\mathcal{O}(n^3)$) and much worse than L-BFGS ($\mathcal{O}(mn)$ where m is the number of past steps L-BFGS utilizes).

Table 1: Conjugate gradient steps for tweaks to HF

Algorithm	Damping tweaks	CG back-tracking tweaks	CG iterations
HF	with Hessian damping	with CG back-tracking	93,936
		without CG back-tracking	99,696
	without Hessian damping	with CG back-tracking	226,021
		without CG back-tracking	28,2803

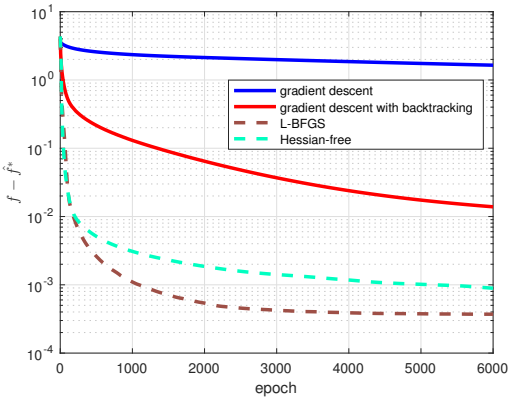

Figure 7: Comparisons of GD (w/ and w/o back-tracking) and second-order methods.

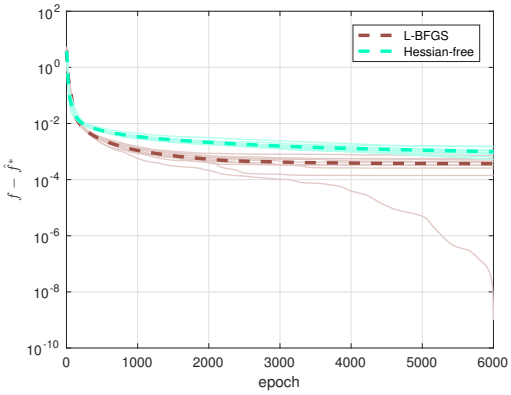

Figure 8: Comparisons of Hessian-free and L-BFGS methods.

We notice that the original Hessian-paper [12] paper mentions several tweaks to the method without justifying qualitatively how these tricks influence convergence rate and cost. We look into two key tweaks: Hessian damping and CG backtracking, and the results are shown in Fig. 9 and Table 1. We may conclude that, those tricks do not generally help much in convergence rate or cost. However Hessian-damping helps saving CG iterations and evaluations by avoiding adventurous steps when we have unusual curvatures, and there is no obvious evidence that CG-backtracking helps convergence by pruning the CG steps in our experiment. The possible explanation is that the stopping criterion of CG terminated CG in time before unnecessary steps are made.

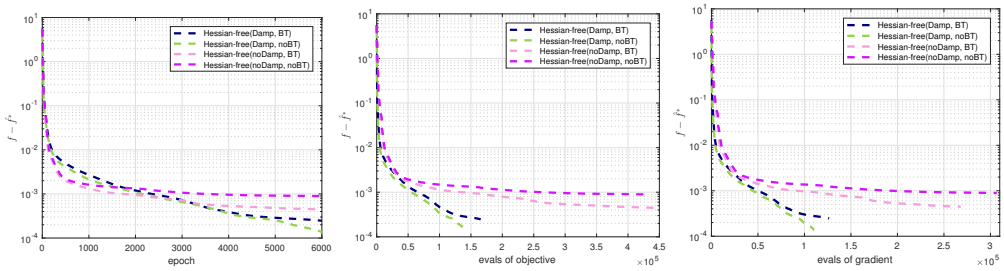

Figure 9: Comparisons of tweaks in HF.

4.5 Momentum L-BFGS

Between the second-orders methods we experimented on, L-BFGS outperforms HF in almost all situations. However due to the use of back-tracking and lacking momentum-style heuristics for escaping local minima and acceleration, we look into the issue of improving the current L-BFGS with momentum. Similar to what we have done in momentum method, we introduce the momentum v_t, which is dependent on the former momentum with a decaying parameter $\gamma = 0.9$, we tune the "step size"$\eta = 0.05$. But we replace the gradient $\nabla f(\theta)$ with the direction negative $-p$ calculated from L-BFGS:

Algorithm 2 Momentum HF Method

for $t = 1, 2, ...$ **do**
  $p_t \leftarrow \text{L-BFGS}(s, y)$
  $v_t \leftarrow \gamma v_{t-1} + \eta p_t$
  $\theta_{t+1} \leftarrow \theta_t + v_t$
  $s \leftarrow [s, \theta_{t+1} - \theta_t]$, $y \leftarrow [y, \nabla f(\theta_{t+1}) - \nabla f(\theta_t)]$
end for

As we can see from the comparison, momentum L-BFGS is inferior to L-BFGS with backtracking, especially at the beginning of training. This is because the step at the beginning can be as large as 1, which is much larger than 0.01 in momentum L-BFGS. To make the comparson fare, we implement L-BFGS with fixed step size of 0.01, momentum L-BFGS outperforms it. All the three L-BFGS methods beat the Adam for more than one order.

Fig. 10 shows the results of the momentum L-BFGS compared with L-BFGS with/without backtracking. The convergence rate with respect to the number of evaluation of objective function and gradients are similar to that based on the number of epoch, objective evaluation [4], gradient evaluation.

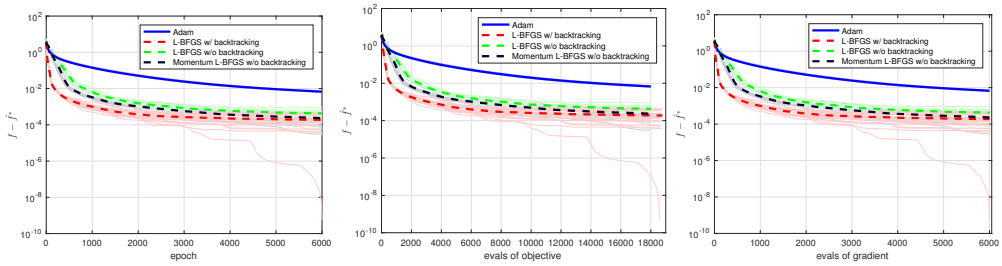

Figure 10: Comparisons of tweaks in L-BFGS.

In our experiments, step size in momentum L-BFGS should be carefully tuned, if it is too large (larger than 0.1), the loss would oscillate to infinity. When the step size is too small (smaller than 0.001), the objective value will also oscillate, but not likely go to infinity. This maybe due to the large proportion of momentum makes the move of parameters almost in a line in parameter space, forcing the parameters to go over hills and valleys without much consideration of local gradient.

We also tried to combine other tricks in adaptive first-order methods, but the system become unstable and hard to tune.

5 Conclusion and Future Work

We show empirically that HF and L-BFGS converge faster than all the modern first-order methods we have tried, including GD with backtracking and Adam, in the task of MNIST classification with our DNN architecture. The advantage holds in the sense of outer iterations and inner iterations (measured by number of evaluating the value and gradient of objective function). HF method fall behind L-BFGS later due to extensive iterations in the inner conjugate gradient loop. To make L-BFGS compatible with modern deep learning network framework, as well as improve the convergence rate, we applied momentum to the L-BFGS without backtracking. Our experiments show that the momentum version outperforms the L-BFGS without backtracking.

Acknowledgments

Thanks to the TA Hao Zhao for his help and constructive advice on the project. We also would like to give thanks to the anonymous reviewers for their helpful feedback on our midterm report.

[4] Momentum L-BFGS and other three methods without backtracking evaluates objective function for 18,769 and 18,000 times (train and test for 12,000 times) on average.

References

[1] R. Battiti. First-and second-order methods for learning: between steepest descent and newton's method. *Neural computation*, 4(2):141–166, 1992.

[2] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade.*, pages 437–478, 2012.

[3] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[4] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *NIPS 2014: Neural Information Processing Systems*, pages 2933–2941, 2014.

[5] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M.A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. *NIPS 2012: NEural Information Processing Systems*, pages 1–11, 2012.

[6] R. Hecht-Nielsen. Theory of the backpropagation neural network. *Proceedings of the International Joint Conference on Neural Networks*, pages 593–608, 1989.

[7] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv eprint arXiv:1412.6980*, 2014.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] D. Li and M. Fukushima. A modified bfgs method and its global convergence in nonconvex minimization. *Journal of Computational and Applied Mathematics*, 129(1):15–35, 2001.

[10] J. Martens. Deep learning via hessian-free optimization. *Lecture Notes*, 2010.

[11] J. Martens. Deep learning via hessian-free optimization. *Proceedings of the 27-th International Conference on Machine Learning*, pages 735–742, 2010.

[12] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040, 2011.

[13] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[14] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

[15] J. Ngiam, A. Coates, A. Lahiri, and et al. On optimization methods for deep learning. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272.

[16] S. Ruder. An overview of gradient descent optimization algorithm. *arXiv eprint arXiv:1609.04747*, 2016.

[17] T. Tieleman and G. Hinton. Rmsprop, coursera: Neural networks for machine learning. *Technical report, Lecture Notes 6.5*, 2012.

[18] S. J. Wright and J. Nocedal. Numerical optimization. *Springer*, pages 164–184.

[19] M. D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv eprint arXiv:1212.5701*, 2012.